PCF Learned Sort: a Learning Augmented Sort Algorithm with $\mathcal{O}(n \log \log n)$ Expected Complexity

Anonymous authors
Paper under double-blind review

Abstract

Sorting is one of the most fundamental algorithms in computer science. Recently, Learned Sorts, which use machine learning to improve sorting speed, have attracted attention. While existing studies show that Learned Sort is empirically faster than classical sorting algorithms, they do not provide theoretical guarantees about its computational complexity. We propose PCF Learned Sort, a theoretically guaranteed Learned Sort algorithm. We prove that the expected complexity of PCF Learned Sort is $\mathcal{O}(n \log \log n)$ under mild assumptions on the data distribution. We also confirm empirically that PCF Learned Sort has a computational complexity of $\mathcal{O}(n \log \log n)$ on both synthetic and real datasets. This is the first study to theoretically support the empirical success of Learned Sort, and provides evidence for why Learned Sort is fast.

1 Introduction

Sorting is one of the most fundamental algorithms in computer science and has been extensively studied for many years. Recently, a novel sorting method called Learned Sort has been proposed (Kraska et al., 2019). In Learned Sort, a machine learning model is trained to estimate the distribution of elements in the input array, specifically, the cumulative distribution function (CDF). The model's predictions are then used to rearrange the elements, followed by a minor refinement step to complete the sorting process. Empirical results show that Learned Sort is faster than classical sorting algorithms, including highly optimized counting-based sorting algorithms, comparison sorting algorithms, and hybrid sorting algorithms.

On the other hand, there are few theoretical guarantees regarding the computational complexity of Learned Sort. The first proposed Learned Sort algorithm (Kraska et al., 2019) has a best-case complexity of $\mathcal{O}(n)$, but its expected or worst-case complexity is not discussed. The more efficient Learned Sort algorithms proposed later (Kristo et al., 2020; 2021a) also have $\mathcal{O}(n)$ best-case complexity, but $\mathcal{O}(n^2)$ worst-case complexity (or $\mathcal{O}(n \log n)$ with some modifications). The goal of this paper is to develop a Learned Sort that is theoretically guaranteed to be computationally efficient.

We propose PCF Learned Sort, which can sort with an expected complexity $\mathcal{O}(n \log \log n)$ under mild assumptions on the data distribution. Furthermore, we prove that PCF Learned Sort can sort with a worst-case computational complexity $\mathcal{O}(n \log n)$. We then empirically confirm that our Learned Sort can sort with an average complexity of $\mathcal{O}(n \log \log n)$ on both synthetic and real datasets.

Independently and concurrently, Zeighami & Shahabi (2024) explored complexity-guaranteed Learned Sort. While our PCF Learned Sort is motivated by similar principles and incorporates comparable design choices, several key distinctions exist between our approach and theirs. A comprehensive comparison between our method and that of Zeighami & Shahabi (2024) is provided in Section 5.

This paper is organized as follows. In Section 2, we first present related work. Then, in Section 3, we introduce our PCF Learned Sort and provide theorems along with brief explanations of their proofs, establishing the computational complexity guarantees. Then, in Section 4, we empirically verify our theorems. We then discuss our findings, limitations, and potential future directions in Section 5. Finally, in Section 6, we summarize our results and conclusions.

2 Related Work

Our research is in the context of algorithms with machine learning (Section 2.1). There are two types of sorting algorithms, comparison sorts (Section 2.2) and non-comparison sorts (Section 2.3), and our proposed method is a non-comparison sort. However, the idea, implementation, and proof of computational complexity of our method are similar to those of sample sort, which is a type of comparison sort. Furthermore, our proposed algorithm and proof of computational complexity are based on those of Learned Index (Section 2.4).

2.1 Algorithms with Machine Learning

Our research lies in the emerging field of Learned Sort, which uses machine learning techniques to improve the performance of classical sorting algorithms. Kraska et al. (2019) proposed a method for fast sorting that employs a model $\tilde{F}(q)$ trained to approximate the CDF $F(q)$. This first Learned Sort roughly sorts the array by placing each element x in the array at the $n\tilde{F}(x)$-th position of the output array. It then sorts the array completely by refining the roughly sorted array using insertion sort. This algorithm has a best-case complexity of $\mathcal{O}(n)$, but a worst-case complexity of $\mathcal{O}(n^2)$ because of the exception handling for collisions and final refinement.

Subsequently, implementations have been proposed that improve cache efficiency (Kristo et al., 2020) and are robust against key duplication (Kristo et al., 2021a). Despite these improvements, these approaches still employ insertion sort during the final refinement, which leads to a worst-case complexity of $\mathcal{O}(n^2)$. The worst-case complexity can be reduced to $\mathcal{O}(n \log n)$ by using a sorting algorithm with small worst-case complexity, such as Introsort (Musser, 1997) or TimSort (McIlroy, 1993). However, this complexity is the same as that of many classical comparison sorting algorithms and thus does not explain the empirical success of Learned Sort.

A related research area, known as algorithms with predictions, also aims to improve algorithm performance by incorporating predictions from machine learning models (Mitzenmacher & Vassilvitskii, 2022). In this context, machine learning predictions are typically assumed to be available at no cost, and the models are treated as an opaque box. Algorithms with predictions have been successfully applied to a wide range of problems, including caching (Narayanan et al., 2018; Rohatgi, 2020; Lykouris & Vassilvitskii, 2021; Im et al., 2022), ski rental (Purohit et al., 2018; Gollapudi & Panigrahi, 2019; Shin et al., 2023), scheduling (Gollapudi & Panigrahi, 2019; Lattanzi et al., 2020; Lassota et al., 2023), and matching (Antoniadis et al., 2023; Dinitz et al., 2021; Sakaue & Oki, 2022).

As part of the study of algorithms with predictions, there has been research on sorting with predictions Lu et al. (2021); Chan et al. (2023); Erlebach et al. (2023). In particular, Bai & Coester (2023) proposed a sorting algorithm with predictions with tight performance guarantees. It is important to note, however, that their analysis excludes the cost of training and inference of the machine learning model from the computational complexity guarantee. This exclusion contrasts with our problem setting, where we ensure that the computational complexity covers the entire process from receiving an unsorted array to returning a sorted array.

2.2 Comparison Sorts

Sorting algorithms that use comparisons between keys and require no other information about the keys are called comparison sorts. It is well-known that the worst-case complexity of a comparison sort is at least $\Omega(n \log n)$. Commonly used comparison sorting algorithms include quick sort, heap sort, merge sort, and insertion sort. The GNU Standard Template Library in C++ uses Introsort (Musser, 1997), an algorithm that combines quick sort, heap sort, and insertion sort. Java (Java, 2023) and Python up to version 3.10 (Peters, 2002) use TimSort (McIlroy, 1993), an improved version of merge sort. Python 3.11 and later use Powersort (Munro & Wild, 2018), a merge sort that determines a near-optimal merge order.

Sample sort (Frazer & McKellar, 1970) is an extension of quick sort that uses multiple pivots, whereas quick sort uses only one pivot. Sample sort samples a small number of keys from the array, determines multiple pivots, and uses them to partition the array into multiple buckets. The partitioning is repeated recursively

until the array is sufficiently small. The in-place parallel superscalar sample sort (Axtmann et al., 2022) is one of the most efficient sample sort implementations. Its computational and cache efficiency is theoretically guaranteed by a theorem about the probability that the pivots partition the array (nearly) equally.

2.3 Non-Comparison Sorts

Non-comparison sorts use information other than key comparison. Radix sort and counting sort are the most common types of non-comparison sorts. Radix sort uses counting sort as a subroutine for each digit. When the number of digits in the array element is w, the computational complexity of radix sort is $\mathcal{O}(wn)$. Thus, radix sort is particularly effective when the number of digits is small. There are several variants of radix sort, such as Spreadsort (Ross, 2002), which integrates the advantages of comparison sort into radix sort and is implemented in the Boost C++ Libraries, and RegionSort (Obeya et al., 2019), which enables efficient parallelization by modeling and resolving dependencies among the element swaps.

In addition, non-comparison sorting algorithms tailored for specific data types have been developed. For integer arrays, a deterministic algorithm with worst-case complexity of $\mathcal{O}(n \log\log n)$ (Han, 2002) and a randomized algorithm with expected complexity of $\mathcal{O}(n\sqrt{\log\log n})$ (Han & Thorup, 2002) have been proposed. For real-valued arrays, recent advances have led to the development of a sorting algorithm with a worst-case complexity of $\mathcal{O}(n\sqrt{\log n})$ (Han, 2020). Our PCF Learned Sort also targets real-valued arrays and, under mild assumptions on the distribution, achieves an expected complexity of $\mathcal{O}(n \log\log n)$, which is smaller than (Han, 2020).

2.4 Learned Index

Kraska et al. (2018) showed that index data structures such as B-trees and Bloom filters can be made faster or more memory efficient by combining them with machine learning models and named such novel data structures Learned Index. Since then, various learning augmented B-trees (Wang et al., 2020; Kipf et al., 2020) and learning augmented Bloom filters (Mitzenmacher, 2018; Vaidya et al., 2021; Sato & Matsui, 2023) have been proposed. There are several works on learning augmented B-trees whose performance is theoretically guaranteed. PGM-index (Ferragina & Vinciguerra, 2020) is a learning augmented B-tree that is guaranteed to have the same worst-case query complexity as the classical B-tree, i.e., $\mathcal{O}(\log n)$. Zeighami & Shahabi (2023) proposed a learning augmented B-tree with an expected query complexity of $\mathcal{O}(\log\log n)$ under mild assumptions on the distribution.

3 Methods

This section describes our PCF Learned Sort. First, in Section 3.1, we give an overview of the algorithm and the key theorems used to guarantee the expected and worst-case computational complexity. Next, in Section 3.2, we introduce the specific algorithm for PCF Learned Sort and guarantee its computational complexity.

3.1 Method Overview

Our algorithm repeats recursive model-based bucketing until the array is small enough or the bucketing "fails." In model-based bucketing, a CDF model is trained and then used to partition an array of length n into (approximately) γ buckets, satisfying the following two conditions: all elements in the i-th bucket are smaller than all elements in the $(i+1)$-th bucket, and with high probability, all buckets are approximately the same size. If we set $\gamma = n^c$ (where $0 < c < 1$ is a constant) and the partitioning into buckets is done well, the size of each bucket will be approximately n^{1-c}, so the size of the bucket in the k-th recurrence depth will be approximately $n^{(1-c)^k}$. Here,

$$k \geq -\frac{\log\log n}{\log(1-c)} \Rightarrow n^{(1-c)^k} \leq n^{(1-c)^{-\frac{\log\log n}{\log(1-c)}}} = e \tag{1}$$

(where log is the natural logarithm, so the base is e). That is, with a recursion depth of $k = \Theta(\log\log n)$, the size of the bucket is small enough, i.e., the max recursion depth is $\mathcal{O}(\log\log n)$. Thus, if the expected

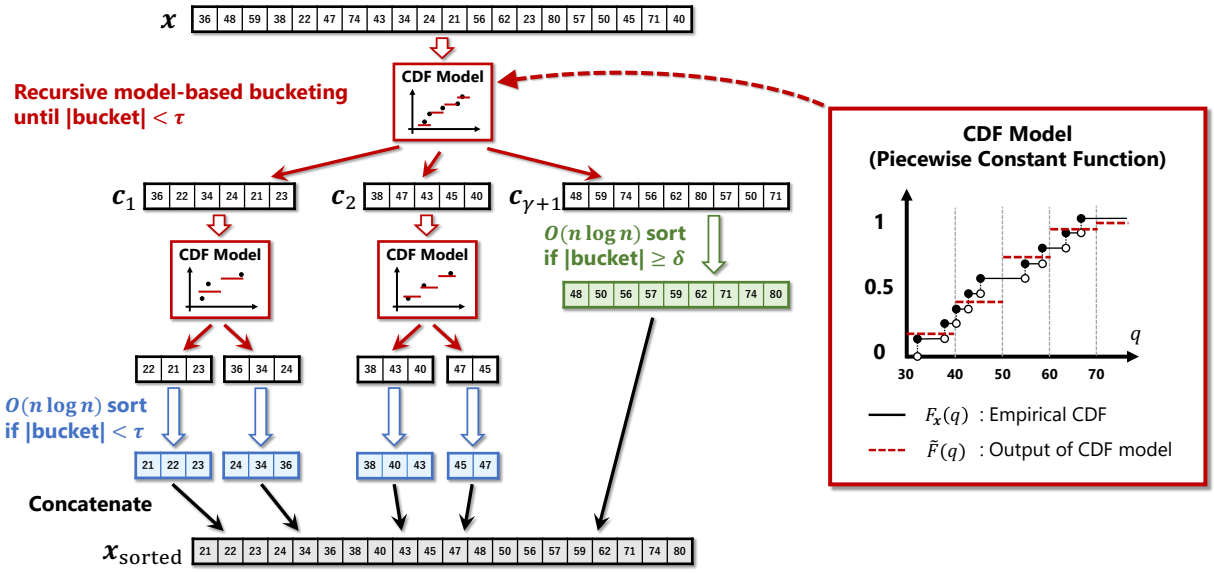

Figure 1: PCF Learned Sort: First, the input array is partitioned into $\gamma+1$ buckets using a CDF model-based method. Buckets larger than δ or smaller than τ are sorted with a "standard" sort algorithm of complexity $\mathcal{O}(n \log n)$ (e.g., IntroSort). Otherwise, the recursive model-based bucketing is repeated. Finally, the sorted arrays are concatenated. The CDF model used for bucketing is a Piecewise Constant Function (PCF). The function is constant within each interval, and the interval widths are constant.

computational complexity of the partition is $\mathcal{O}(n)$ and the probability that the bucketing "fails" is sufficiently small, the total computational complexity is $\mathcal{O}(n \log \log n)$. Under mild assumptions on the distribution, we can prove that this fast and accurate partition can be achieved by using the Piecewise Constant Function (PCF) as the CDF model.

Let $\mathcal{D}$ ($\subseteq \mathbb{R}$) be the range of possible values of the input array and $\boldsymbol{x}$ ($\in \mathcal{D}^n$) be the input array. If the length of the input array, denoted as $|\boldsymbol{x}|$, is less than τ, our algorithm sorts the input array using a "standard" sort algorithm with $\mathcal{O}(n \log n)$ complexity, such as IntroSort, where τ is a predetermined constant. Otherwise, if $|\boldsymbol{x}| \geq \tau$, model-based bucketing is performed.

The model-based bucketing method $\mathcal{M}$ takes an input array $\boldsymbol{x}$ and partitions it into several buckets. First, it sets the parameter γ ($\in \mathbb{N}$), which determines the number of buckets. The parameter γ is determined as a function of the array length n. The number of buckets returned by $\mathcal{M}$ is $\gamma + 1$. Next, all or some elements of $\boldsymbol{x}$ are used to train the CDF model $\tilde{F}\colon \mathcal{D} \to [0, 1]$. The $\tilde{F}(q)$ is trained to predict $F_{\boldsymbol{x}}(q)$, the empirical CDF of $\boldsymbol{x}$ defined as:

$$F_{\boldsymbol{x}}(q) \coloneqq \frac{|\{i \in \{1, \ldots, n\} \mid x_i \leq q\}|}{|\boldsymbol{x}|}. \tag{2}$$

The specific model and training method of $\tilde{F}(q)$ are explained in Section 3.2. Finally, the CDF model is used to partition the input array $\boldsymbol{x}$ into $\gamma+1$ buckets. All $\gamma+1$ buckets, $\{\boldsymbol{c}_j\}_{j=1}^{\gamma+1}$, are initialized to be empty, and then for each $i \in \{1, \ldots, n\}$, x_i is appended to $\boldsymbol{c}_{\lfloor \tilde{F}(x_i)\gamma \rfloor + 1}$. This is based on the intuition that the number of elements less than or equal to x_i in the array $\boldsymbol{x}$ (i.e., $|\{j \in \{1, \ldots, n\} \mid x_j \leq x_i\}|$) is approximately equal to $n\tilde{F}(x_i)$.

We restrict the CDF model $\tilde{F}$ to non-decreasing functions to ensure that the bucket with the larger ID gets the larger value, i.e.,

$$p \in \boldsymbol{c}_j \ \wedge \ q \in \boldsymbol{c}_k \ \wedge \ j < k \Rightarrow p < q. \tag{3}$$

Algorithm 1 The Learned Sort algorithm

Input:
  $\boldsymbol{x} \in \mathbb{R}^n$: The array to be sorted
Output:
  $\boldsymbol{x}_{\text{sorted}} \in \mathbb{R}^n$: The sorted version of array $\boldsymbol{x}$
Algorithm:
  STANDARD-SORT($\boldsymbol{x}$) : The sort algorithm with computational complexity $\mathcal{O}(n \log n)$
  CDF-MODEL($\boldsymbol{x}$) : Instantiate a CDF Model $\tilde{F}(q)$ that estimates $F_{\boldsymbol{x}}(q)$

function LEARNED-SORT($\boldsymbol{x}$)
  $n \leftarrow |\boldsymbol{x}|$
  if $n < \tau$ **then**
    return STANDARD-SORT($\boldsymbol{x}$)

  // Model-based bucketing
  $\tilde{F}(q) \leftarrow$ CDF-MODEL($\boldsymbol{x}$)
  $\boldsymbol{c}_1 \leftarrow [\,],\ \boldsymbol{c}_2 \leftarrow [\,],\ \ldots\ ,\ \boldsymbol{c}_{\gamma+1} \leftarrow [\,]$
  for $i = 1, 2, \ldots, n$
    $j \leftarrow \lfloor \tilde{F}(x_i)\gamma \rfloor + 1$
    $\boldsymbol{c}_j$.APPEND(x_i)

  // Recursively sort and concatenate
  for $j = 1, 2, \ldots, \gamma + 1$
    if $|\boldsymbol{c}_j| \geq \delta$ **then**
      $\boldsymbol{c}_j \leftarrow$ STANDARD-SORT($\boldsymbol{c}_j$)
    else
      $\boldsymbol{c}_j \leftarrow$ LEARNED-SORT($\boldsymbol{c}_j$)
  $\boldsymbol{x}_{\text{sorted}} \leftarrow$ CONCATENATE($\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_{\gamma+1}$)
  return $\boldsymbol{x}_{\text{sorted}}$

This means that each bucket is responsible for a disjoint and continuous interval. Let $t_j = \min_{x \in \boldsymbol{c}_j} x$ $(j = 1, \ldots, \gamma + 1)$, $t_{\gamma+2} = \infty$, then the j-th bucket $\boldsymbol{c}_j$ $(j = 1, \ldots, \gamma + 1)$ is responsible for a continuous interval $\mathcal{I}_j \coloneqq [t_j, t_{j+1})$.

After model-based bucketing, our algorithm determines for each bucket whether the bucketing "succeeds" or "fails." For each $j \in \{1, \ldots, \gamma+1\}$, we check whether the size of bucket $\boldsymbol{c}_j$ is less than δ, where δ is an integer determined by n. If $|\boldsymbol{c}_j| \geq \delta$ (which means the bucketing "fails"), the bucket is sorted using the "standard" sort algorithm (e.g., IntroSort). If $|\boldsymbol{c}_j| < \delta$ (which means the bucketing "succeeds"), the bucket is sorted by recursively calling our Learned Sort algorithm. Note that the parameters such as γ and δ are redetermined for each recursion according to the size of the bucket (i.e., the input array in the next recursion step), and the CDF model is retrained for each bucket. After each bucket is sorted, the buckets are concatenated. The input array $\boldsymbol{x}$ is sorted by the above procedure. The basic idea of our algorithm is visualized in Figure 1, and the pseudocode is given in Algorithm 1.

The following is a lemma about the worst-case complexity of our Learned Sort as defined above.

Lemma 3.1. *Assume that there exists a model-based bucketing algorithm $\mathcal{M}$ such that $\mathcal{M}$ can perform bucketing (including model training and inferences) an array of length n into $\gamma + 1 = \mathcal{O}(n)$ buckets with a worst-case complexity of $\mathcal{O}(n)$. Also, assume that the "standard" sort algorithm has a worst-case complexity of $\mathcal{O}(n \log n)$. Then, the worst-case complexity of our Learned Sort with such $\mathcal{M}$ and $\delta = \lfloor n^d \rfloor$ (where d is a constant satisfying $0 < d < 1$) is $\mathcal{O}(n \log n)$.*

This lemma can be intuitively shown from the following two points: (i) the maximum recursion depth is $\mathcal{O}(\log \log n)$, and (ii) each element of the input array $\boldsymbol{x}$ undergoes several bucketing and only one "standard"

sort. (i) can be shown from the fact that the size of the bucket in the i-th recursion depth is less than n^{d^i}, and (ii) is evident from the algorithm's design since the buckets sorted by "standard" sort are now left only to be concatenated. The exact proof is given in Appendix A.1.

Next, we introduce an important lemma about the expected computational complexity of our Learned Sort. The following assumption is necessary to guarantee the expected computational complexity.

Assumption 3.2. The input array $\boldsymbol{x} \in \mathcal{D}^n$ is formed by independent sampling according to a probability density distribution $f(x) \colon \mathcal{D} \to \mathbb{R}_{\geq 0}$.

We define $f_{\mathcal{I}}(x) \colon \mathcal{I} \to \mathbb{R}_{\geq 0}$ to be the conditional probability density distribution of $f(x)$ under the condition that $x \in \mathcal{I}$ for a interval $\mathcal{I} \subseteq \mathcal{D}$, i.e.,

$$f_{\mathcal{I}}(x) \coloneqq \frac{f(x)}{\int_{\mathcal{I}} f(y)dy}. \tag{4}$$

The expected computational complexity of our proposed Learned Sort is guaranteed by the following lemma.

Lemma 3.3. *Let $\boldsymbol{x}_{\mathcal{I}}$ $(\in \mathcal{I}^n)$ be the array formed by sampling n times independently according to $f_{\mathcal{I}}(x)$. Assume that there exist a model-based bucketing algorithm $\mathcal{M}$ and a constant d $(\in (0,1))$ that satisfy the following for any interval $\mathcal{I}$ $(\subseteq \mathcal{D})$:*

- *$\mathcal{M}$ can perform bucketing (including model training and inferences) an array of length n, with an expected complexity of $\mathcal{O}(n)$. That is, $\mathcal{M}$ can take the array $\boldsymbol{x}_{\mathcal{I}}$ $(\in \mathcal{I}^n)$ as input and divide it into $\gamma + 1$ buckets, $\{\boldsymbol{c}_j\}_{j=1}^{j=\gamma+1}$ satisfying Equation (3), with an expected complexity of $\mathcal{O}(n)$.*
- *$\gamma + 1 = \mathcal{O}(n)$.*
- *$\Pr[\exists j, |\boldsymbol{c}_j| \geq \lfloor n^d \rfloor] = \mathcal{O}\left(\frac{1}{\log n}\right)$.*

Also, assume that the "standard" sort algorithm has an expected complexity of $\mathcal{O}(n \log n)$. Then, the expected complexity of our Learned Sort with such $\mathcal{M}$ and $\delta = \lfloor n^d \rfloor$ is $\mathcal{O}(n \log \log n)$.

This lemma can be proved intuitively by the following two points: (i) the maximum recursion depth is $\mathcal{O}(\log \log n)$, and (ii) the expected total computational complexity from the i-th to the $(i+1)$-th recursion depth is $\mathcal{O}(n)$. (i) is the same as in the explanation of the proof of Lemma 3.1. (ii) can be shown from the fact that the expected computational complexity from the i-th to the $(i+1)$-th recursion depth is $\mathcal{O}(n \log n)$ with probability $\mathcal{O}(\frac{1}{\log n})$, and $\mathcal{O}(n)$ in other cases. See Appendix A.2 for the exact proof.

Note that the assumption of Lemma 3.3 includes "$\mathcal{M}$ works well with high probability for any $\mathcal{I}$ $(\subseteq \mathcal{D})$." This is because our Learned Sort algorithm recursively repeats the model-based bucketing. The range of elements in the bucket, i.e., the input array in the next recursion step, can be any interval $\mathcal{I}$ $(\subseteq \mathcal{D})$.

3.2 PCF Learned Sort

We propose PCF Learned Sort as an implementation that satisfies the assumptions of Lemma 3.1 and Lemma 3.3, and thus has a guarantee that the worst-case complexity is $\mathcal{O}(n \log n)$ and the expected complexity is $\mathcal{O}(n \log \log n)$. PCF Learned Sort approximates the CDF by a Piecewise Constant Function (PCF). The PCF is a function that has intervals of equal width and outputs the same value in each interval (the right side of Figure 1). The study that develops a Learned Index with a theoretical guarantee on its complexity (Zeighami & Shahabi, 2023) also used PCF as a CDF model.

The model-based bucketing method in PCF Learned Sort $\mathcal{M}_{\mathrm{PCF}}$ trains the CDF model $\tilde{F}$ in the following way. First, the parameters $\alpha \in \{1, \ldots, n\}$ and $\beta \in \mathbb{N}$ are determined by n, the length of the input array. The parameter α is the number of samples to train the CDF model, and β is the number of intervals in the PCF. Next, the PCF is trained by counting the number of samples in each interval. Using $x_{\min} = \min_i x_i$ and $x_{\max} = \max_i x_i$, let $i(x)$ be a function defined as follows:

$$i(x) = \left\lfloor \frac{x - x_{\min}}{x_{\max} - x_{\min}} \beta \right\rfloor + 1. \tag{5}$$

α samples are taken at random from $\boldsymbol{x}$ to form $\boldsymbol{a}$ ($\in \mathcal{D}^{\alpha}$), and then $i(x)$ is used to form the array $\boldsymbol{b}$ ($\in {\mathbb{Z}_{\geq 0}}^{\beta+1}$) defined as follows:

$$b_i = |\{j \in \{1, \ldots, \alpha\} \mid i(a_j) \leq i\}|. \tag{6}$$

This counting corresponds to the training of the PCF. Note that $\boldsymbol{b}$ is an non-decreasing non-negative array and $b_{\beta+1} = \alpha$, i.e., $0 \leq b_1 \leq b_2 \leq \cdots \leq b_{\beta+1} = \alpha$.

The inference for the CDF model $\tilde{F}(x)$ is performed using $i(x)$, $\boldsymbol{b}$, and the following equation:

$$\tilde{F}(x) = \frac{b_{i(x)}}{\alpha}. \tag{7}$$

Since $i(x)$ is a non-decreasing function and $\boldsymbol{b}$ is also a non-decreasing array, $\tilde{F}(x)$ is a non-decreasing function. Also, $0 \leq \tilde{F}(x) \leq 1$ because $0 \leq b_i \leq \alpha$ for every i.

The following is a lemma to bound the probability that $\mathcal{M}_{\mathrm{PCF}}$ will "fail" bucketing. This lemma is important to guarantee the expected computational complexity of PCF Learned Sort.

Lemma 3.4. *Let σ_1 and σ_2 be respectively the lower and upper bounds of the probability density distribution $f(x)$ in $\mathcal{D}$, and assume that $0 < \sigma_1 \leq \sigma_2 < \infty$. That is, $x \in \mathcal{D} \Rightarrow \sigma_1 \leq f(x) \leq \sigma_2$.*

Then, in model-based bucketing of $\boldsymbol{x}_{\mathcal{I}}$ ($\in \mathcal{I}^n$) to $\{\boldsymbol{c}_j\}_{j=1}^{\gamma+1}$ using $\mathcal{M}_{\mathrm{PCF}}$, the following holds for any interval $\mathcal{I}$ ($\subseteq \mathcal{D}$):

$$K \geq 1 \Rightarrow \Pr[\exists j, |\boldsymbol{c}_j| > \delta] \leq \frac{2n}{\delta} \exp\left\{-\frac{\alpha K}{2\gamma}\left(1 - \frac{1}{K}\right)^2\right\}, \tag{8}$$

where

$$K \coloneqq \frac{\gamma\delta}{2n} - \frac{2\sigma_2\gamma}{\sigma_1\beta}. \tag{9}$$

The proof of this lemma is based on and combines proofs from two existing studies. The first is Lemma 5.2. from a study of IPS^{4}o (Axtmann et al., 2022), an efficient sample sort. This lemma guarantees the probability of a "successful recursion step" when selecting pivots from samples and using them to perform a partition. This lemma is for the method that does not use the CDF model, so the proof cannot be applied directly to our case. Another proof we refer to is the proof of Lemma 4.5. from a study that addressed the computational complexity guarantee of the Learned Index (Zeighami & Shahabi, 2023). This lemma provides a probabilistic guarantee for the error between the output of the PCF and the empirical CDF. Some modifications are required to adapt it to the context of sorting and to attribute it to the probability of bucketing failure, i.e., $\Pr[\exists j, |\boldsymbol{c}_j| > \delta]$. By appropriately combining the proofs of these two lemmas, Lemma 3.4 is proved. The exact proof is given in Appendix A.3.

Here, we emphasize that the assumption of this lemma, $0 < \sigma_1 \leq \sigma_2 < \infty$, is sufficiently reasonable and "mild" as described in (Zeighami & Shahabi, 2023). It asserts that the probability density function $f(x)$ is both bounded and nonzero over its domain $\mathcal{D}$. This class of distributions covers the majority of real-world scenarios because real-world data is commonly derived from bounded and continuous phenomena, e.g., heights, weights, and prices.

Using Lemma 3.1, Lemma 3.3, and Lemma 3.4, we can prove the following theorems.

Theorem 3.5. *If $\mathcal{M}_{\mathrm{PCF}}$ is the bucketing method, the sort algorithm with worst-case complexity of $\mathcal{O}(n \log n)$ is "standard" sort, and $\alpha = \beta = \gamma = \delta = \lfloor n^{3/4} \rfloor$, then the worst-case complexity of PCF Learned Sort is $\mathcal{O}(n \log n)$.*

Proof. When $\alpha = \beta = \gamma = \lfloor n^{3/4} \rfloor$, the computational complexity for model-based bucketing is $\mathcal{O}(n)$ because (i) the PCF is trained in $\mathcal{O}(\alpha + \beta) = \mathcal{O}(n^{3/4})$, and (ii) the total complexity of inference for n elements is $\mathcal{O}(n)$, since the inference is performed in $\mathcal{O}(1)$ per element. Therefore, since $\gamma + 1 = \mathcal{O}(n)$, the worst-case complexity of "standard" sort is $\mathcal{O}(n \log n)$, and $\delta = \lfloor n^{3/4} \rfloor$, we can prove the worst-case complexity of PCF Learned Sort is $\mathcal{O}(n \log n)$ by Lemma 3.1. □

Theorem 3.6. *Let σ_1 and σ_2 be the lower and upper bounds, respectively, of the probability density distribution $f(x)$ in $\mathcal{D}$, and assume that $0 < \sigma_1 \leq \sigma_2 < \infty$. Then, if $\mathcal{M}_{\mathrm{PCF}}$ is the bucketing method, the sort algorithm with expected complexity of $\mathcal{O}(n \log n)$ is "standard" sort, and $\alpha = \beta = \gamma = \delta = \lfloor n^{3/4} \rfloor$, then the expected complexity of PCF Learned Sort is $\mathcal{O}(n \log \log n)$.*

Proof. When $\alpha = \beta = \gamma = \lfloor n^{3/4} \rfloor$, the computational complexity for model-based bucketing is $\mathcal{O}(n)$. Since $K = \Omega(\sqrt{n})$ when $\alpha = \beta = \gamma = \delta = \lfloor n^{3/4} \rfloor$, $K \geq 1$ for sufficiently large n, and

$$\frac{2n}{\delta} \exp\left\{ -\frac{\alpha K}{2\gamma} \left(1 - \frac{1}{K}\right)^2 \right\} = \mathcal{O}(n^{\frac{1}{4}} \exp(-\sqrt{n})) \leq \mathcal{O}\left(\frac{1}{\log n}\right). \tag{10}$$

Given that $\gamma + 1 = \mathcal{O}(n)$, the expected complexity of "standard" sort is $\mathcal{O}(n \log n)$, and $\delta = \lfloor n^{3/4} \rfloor$, it follows from Lemma 3.3 and Lemma 3.4 that the expected complexity of PCF Learned Sort is $\mathcal{O}(n \log \log n)$. □

Note that the exact value of σ_1 and σ_2 is not required to run PCF Learned Sort since the parameters for this algorithm, i.e., α, β, γ, and δ, are determined without any prior knowledge. In other words, PCF Learned Sort can sort in expected $\mathcal{O}(n \log \log n)$ complexity as long as $0 < \sigma_1 \leq f(x) \leq \sigma_2 < \infty$, even if it does not know the exact value of σ_1 and σ_2. If σ_1 and σ_2 do not satisfy the assumption of Theorem 3.6, i.e., $\sigma_1 = 0$ or $\sigma_2 = \infty$, then the expected complexity of PCF Learned Sort is increased to $\mathcal{O}(n \log n)$, but from Theorem 3.5, it cannot be greater than $\mathcal{O}(n \log n)$.

4 Experiments

In this section, we confirm our theorems empirically. First, in Section 4.1, we confirm that the computational complexity of PCF Learned Sort is $\mathcal{O}(n \log \log n)$ for both synthetic and real data. Then, in Section 4.2, we conduct experiments with various parameter settings and empirically confirm Lemma 3.4, a lemma that bounds the probability of bucketing failure and is an important lemma to guarantee the expected computational complexity of PCF Learned Sort. Finally, in Section 4.3, we present the results of an experiment measuring sorting time.

4.1 Computational Complexity of PCF Learned Sort

We experimented with synthetic datasets created from the following four types of distributions: uniform distribution ($\mathsf{min} = 0, \mathsf{max} = 1$), normal distribution ($\mu = 0, \sigma = 1$), exponential distribution ($\lambda = 1$), lognormal distribution ($\mu = 0, \sigma = 1$). The input array was generated by independently taking $n = 10^4, \ldots, 10^8$ samples from each distribution. We also used four real datasets, NYC, Wiki, OSM, and Books, which are described below. NYC: pick-up datetimes in the yellow taxi trip records (Kristo, 2021). Wiki: Wikipedia article edit timestamps (Marcus et al., 2020). OSM: uniformly sampled OpenStreetMap locations represented as Google S2 CellIds (Marcus et al., 2020). Books: book sale popularity data from Amazon (Marcus et al., 2020). For each dataset, we randomly sample $n = 10^4, \ldots, 10^8$ elements, shuffle the sampled elements, and then use them as an input array to examine the computational complexity of the sort algorithms. We meticulously counted the total number of basic operations for sorting the input array to observe the computational complexity of each sorting algorithm. Here, the basic operations consist of four arithmetic operations, powers, comparisons, logical operations, assignments, and memory access. We chose this metric, which counts basic operations, to mitigate the environmental dependencies observed in other metrics, such as CPU instructions and CPU time, which are heavily influenced by compiler optimizations and the underlying hardware. This is the same idea as the metric selection in the experiment of (Zeighami & Shahabi, 2023).

The parameters of PCF Learned Sort are set as in Theorem 3.5 and Theorem 3.6, $\alpha = \beta = \gamma = \delta = \lfloor n^{3/4} \rfloor$ and $\tau = 100$. As the "standard" sort algorithm used in PCF Learned Sort, we used quick sort, which has an expected computational complexity of $\mathcal{O}(n \log n)$. We compared our PCF Learned Sort with (plain) quick sort, radix sort, and one of the state-of-the-art learning augmented sort algorithms (Kristo et al., 2021b).

Figure 2 shows the number of operations divided by the length of the input array, n. It shows the mean and standard deviation of the 10 measurements for each condition. The proposed PCF Learned Sort has the

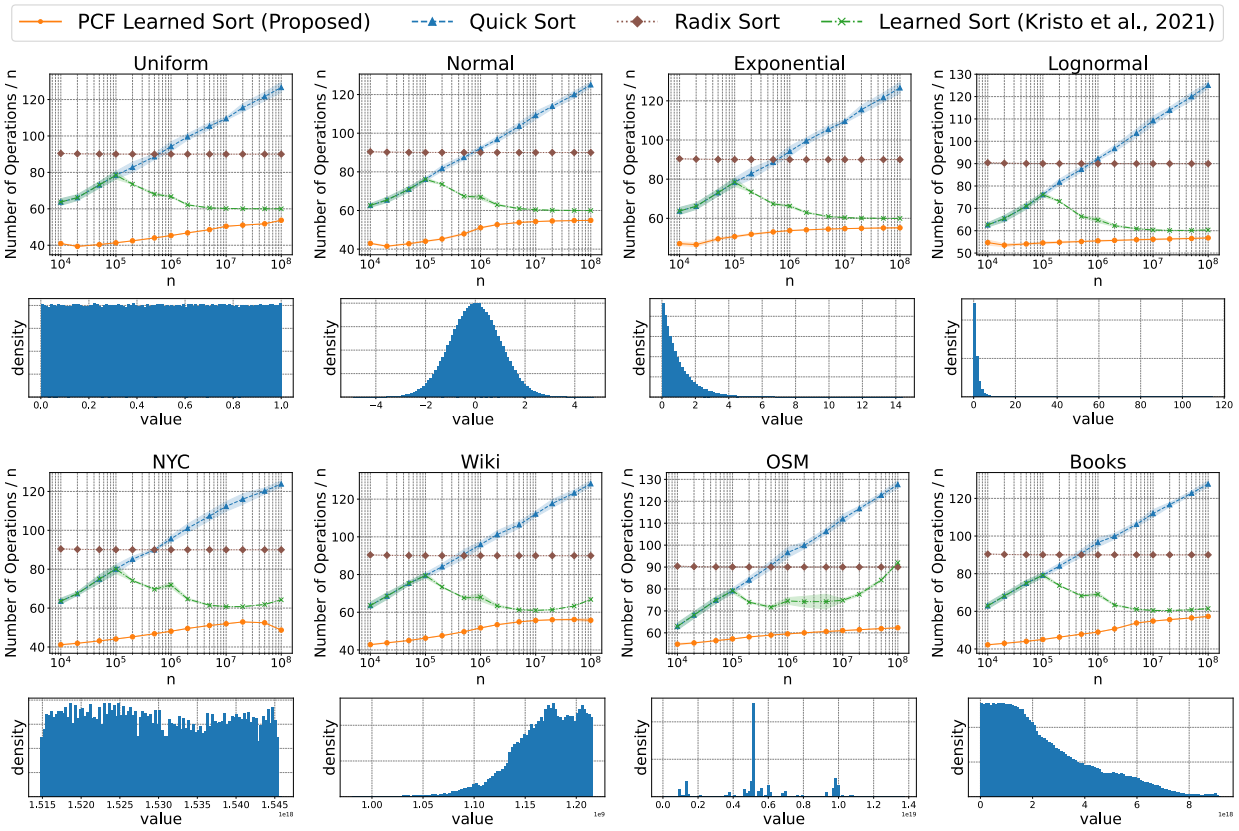

Figure 2: Number of operations to sort the array. Below each graph is a histogram that visualizes the distribution of each dataset. The standard deviation of the 10 measurements is represented by the shaded area.

lowest number of operations for all condition settings. We see that PCF Learned Sort has up to 2.8 times fewer operations than quick sort. Also, while the graph of quick sort is almost linear, the graph of PCF Learned Sort is almost flat. This suggests that PCF Learned Sort has a computational complexity much smaller than $\mathcal{O}(n \log n)$.

The graph of the radix sort is almost perfectly flat, but it always requires more operations than PCF Learned Sort. This difference is due to the different partitioning methods of the two algorithms. While the radix sort performs partitioning at a predetermined granularity, our PCF Learned Sort performs partitioning using intervals that are adaptively set by the learning model. Kristo et al.'s Learned Sort 2021a uses a more complex regression model than ours and focuses on cache efficiency, so it is harder to guarantee computational efficiency and requires more operations to sort than PCF Learned Sort. In particular, for the OSM dataset, the difference between Kristo et al.'s Learned Sort 2021a and PCF Learned Sort is more pronounced. The OSM dataset has a histogram with multiple sharp peaks, which makes it difficult for the CDF model to regress. Therefore, the number of operations for (Kristo et al., 2021a) increases rapidly as n increases. On the other hand, PCF Learned Sort, which has theoretical guarantees on expected and worst-case computations, shows an almost flat graph.

4.2 Confirmation of Lemma 3.4

Lemma 3.4 bounds the probability that a bucket of size greater than δ exists. This is an important lemma that allows us to guarantee the expected computational complexity of PCF Learned Sort. Here, we empirically confirm that this upper bound is appropriate.

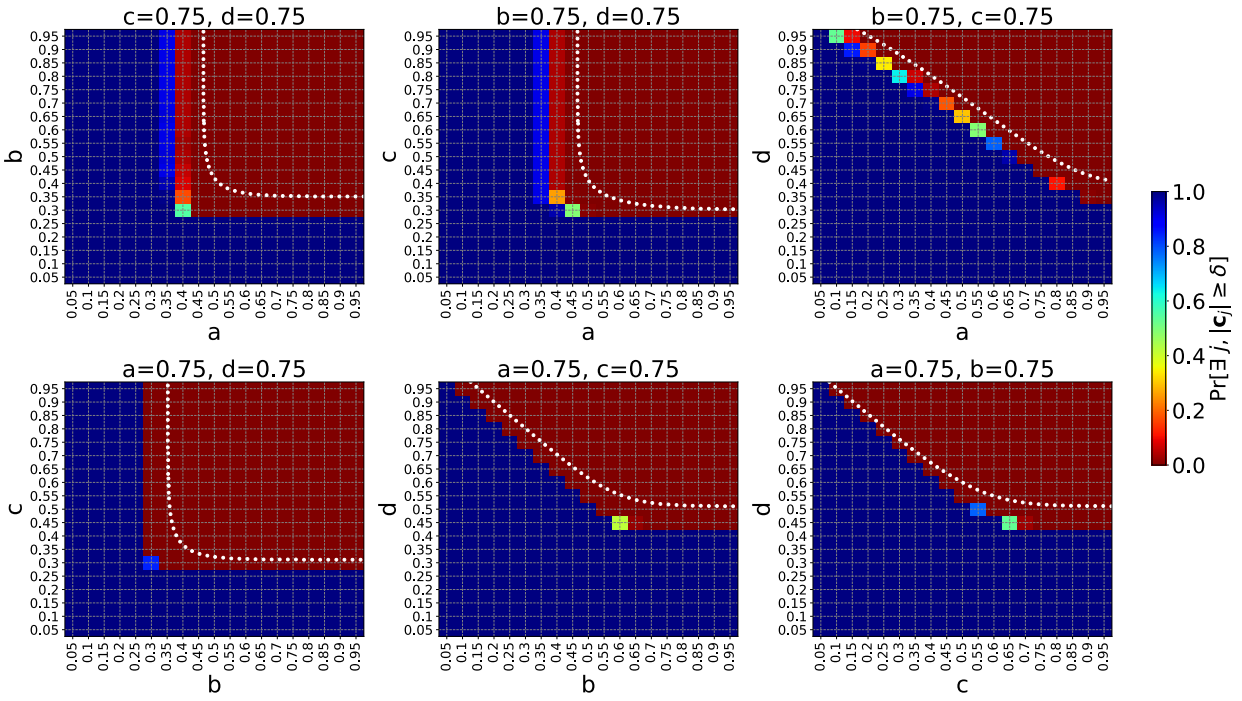

Figure 3: Heatmap showing the empirical frequency of bucketing failure, i.e., $\exists j, |\boldsymbol{c}_j| > \delta$. The variables a, b, c, d, except those on the x- and y-axes, were set to 0.75. The white dotted line represents the parameters that make the right side of Equation (8) equal to 0.5.

We have experimented with $\alpha = \lfloor n^a \rfloor, \beta = \lfloor n^b \rfloor, \gamma = \lfloor n^c \rfloor, \delta = \lfloor n^d \rfloor$, varying a, b, c, d from 0.05 to 0.95 at 0.05 intervals. For each a, b, c, d setting, the following was repeated 100 times: we took $n = 10^6$ elements from the uniform distribution to form the input array and divided the array into $\gamma + 1$ buckets by $\mathcal{M}_{\mathrm{PCF}}$, and checked whether or not $\exists j, |\boldsymbol{c}_j| > \delta$. Thus, for each $a, b, c, d \in \{0.05, 0.10, \ldots, 0.95\}$, we obtained the empirical frequency at which bucketing "fails."

Heat maps in Figure 3 show the empirical frequency of bucketing failures when two of the a, b, c, d parameters are fixed, and the other two parameters are varied. The values of the two fixed variables are set to 0.75, e.g., in the upper left heap map of Figure 3 (horizontal axis is a and vertical axis is b), $c = d = 0.75$. The white dotted line represents the parameter so that the right side of Equation (8) is 0.5. That is, Lemma 3.4 asserts that "in the region upper right of the white dotted line, the probability of bucketing failure is less than 0.5."

We can see that the white dotted line is close to (or slightly to the upper right of) the actual bound of whether bucketing "succeeds" or "fails" more often. In other words, we can see that the theoretical upper bound from Lemma 3.4 agrees well (to some extent) with the actual probability. We can also confirm that, as Lemma 3.4 claims, the probability of bucketing failure is indeed small in the region upper right of the white line.

4.3 Experiments on Sorting Time

Here, we empirically compare the time each algorithm takes to sort. Note that the metric used in Section 4.1, the number of operations, does not change depending on the machine or compilation method, but the sorting time does. All experiments were performed on a Linux machine equipped with an Intel® Core™ i9-11900H CPU @ 2.50GHz and 62GB of memory. GCC version 9.4.0 was used for compilation, employing the `-O3` optimization flag. As the "standard" sort algorithm used in PCF Learned Sort, we used `std::sort`, which has a worst-case computational complexity of $\mathcal{O}(n \log n)$. We compared our PCF Learned Sort with

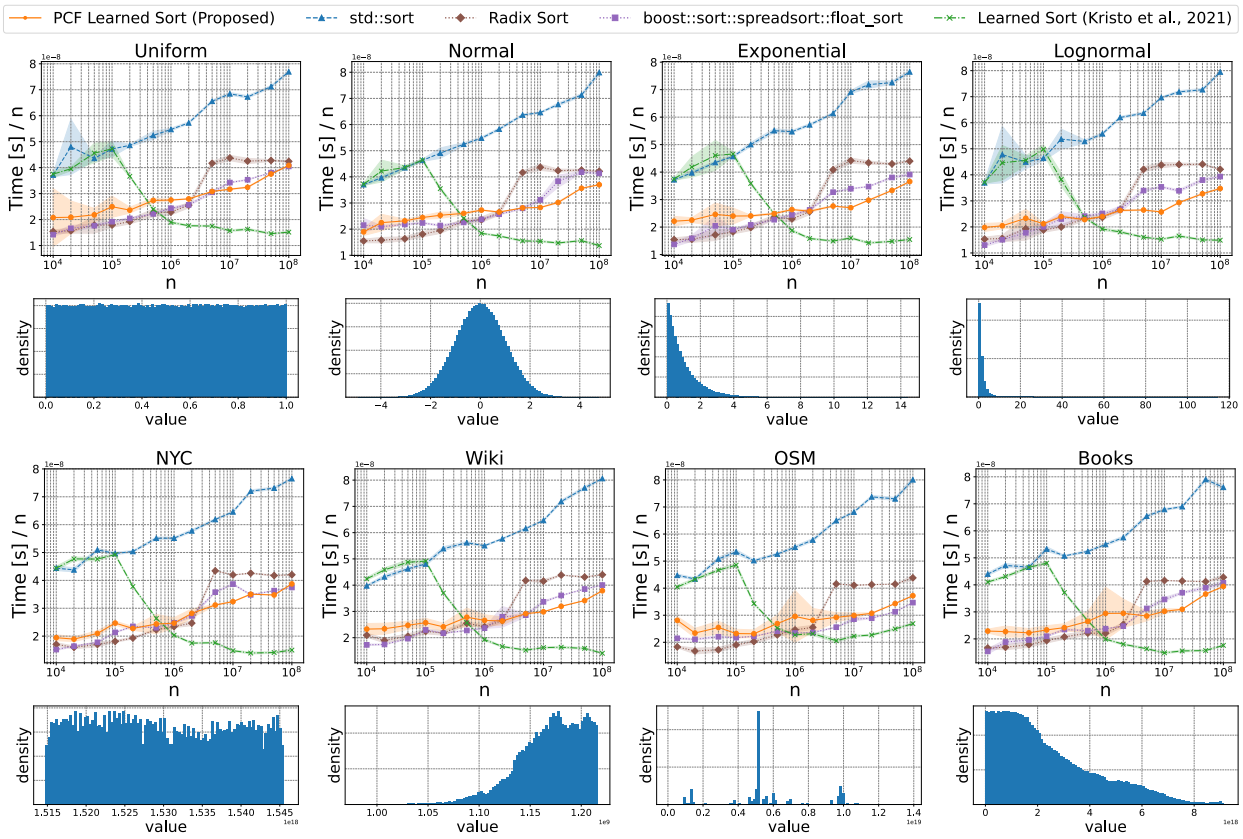

Figure 4: Time to sort the array. Below each graph is a histogram that visualizes the distribution of each dataset. The standard deviation of the 10 measurements is represented by the shaded area.

`std::sort`, radix sort, `boost::sort::spreadsort::float_sort` (Boost C++ implementation of Spreadsort (Ross, 2002)), and one of the state-of-the-art learning augmented sort algorithms (Kristo et al., 2021b).

Figure 4 shows the sorting time divided by the length of the input array, n. It shows the mean and standard deviation of the 10 measurements for each condition. We see that our PCF Learned Sort is up to 2.5 times faster than `std::sort`. Also, while the graph of `std::sort` is almost linear, the PCF Learned Sort graph shows a relatively slow increase, suggesting that PCF Learned Sort has a computational complexity much smaller than $\mathcal{O}(n \log n)$. Furthermore, we find that for relatively large n ($n > 10^6$), our PCF Learned Sort usually outperforms not only radix sort but also Spreadsort, an algorithm that cleverly incorporates the advantage of comparison sort into radix sort.

Kristo et al.'s Learned Sort 2021a turns out to be much faster than PCF Learned Sort. This is because (Kristo et al., 2021a) is implemented with an emphasis on cache efficiency, while we have not done a highly optimized implementation that carefully considers cache. Another reason is that (Kristo et al., 2021a) uses a more complex and accurate (but harder to guarantee complexity and accuracy theoretically) regression model than PCF Learned Sort. Developing a Learned Sort that is as fast as (Kristo et al., 2021a) and can be theoretically guaranteed is a future work.

5 Discussion

Comparison with (Zeighami & Shahabi, 2024). Most recently, a concurrent work (Zeighami & Shahabi, 2024) introduced a theoretical framework for learned database operations, including sorting, indexing,

and cardinality estimation. A key strength of their approach is the formal definition of "distribution learnability" and its applicability to a broad class of distributions, including those subject to distribution shifts. Using this framework, they developed a Learned Sort algorithm with an expected running time of $O(n \log \log n)$ under certain distributional assumptions. While their approach employs a bucketing-based sorting algorithm similar to ours, there are several key differences between their method and ours.

First, their algorithm relies on detailed prior knowledge about the distribution. Specifically, their algorithm explicitly requires a parameter $\varkappa_2$, which represents the "learning possibility" of the distribution (see Definition 3.2 in (Zeighami & Shahabi, 2024) for details). This means that their algorithm cannot be executed without knowing the parameter $\varkappa_2$. Since the parameter $\varkappa_2$ depends on ρ_1 and ρ_2, their algorithm must know the exact values of ρ_1 and ρ_2 or at least their lower and upper bounds. In contrast, our algorithm does not require these values, making it more widely applicable.

Second, they do not provide experimental results. Since estimating ρ_1 and ρ_2 from real data is challenging, empirical evaluation of their method is difficult. On the other hand, since our algorithm does not depend on these specific values, it is easy to implement and evaluate experimentally.

Finally, their algorithm lacks a worst-case complexity guarantee. In particular, there are cases where the algorithm may not terminate, making it impossible to give an upper bound on its worst-case complexity. In contrast, we provide a formal worst-case complexity guarantee of $\mathcal{O}(n \log n)$.

Limitations of the Current Theoretical Framework. The proof of Theorem 3.6, which establishes the expected complexity of PCF Learned Sort as $\mathcal{O}(n \log \log n)$, does not extend to distributions where there exists an $x \in \mathcal{D}$ such that $f(x) = 0$ or $f(x) = \infty$. A similar limitation is observed in Learned Indexes Zeighami & Shahabi (2023). Addressing these constraints and developing a theory or algorithm applicable to a broader class of distributions remains an important direction for future research on Learned Indexes and Learned Sorts.

Future Research Directions. Future work could include integrating a more advanced CDF approximation method with theoretical guarantees into our Learned Sort algorithm. By adopting a refined CDF model and a bucketing algorithm that satisfies the conditions of Lemma 3.3, we may achieve a sorting algorithm with even stronger theoretical guarantees. In other words, improving the accuracy and speed of CDF approximation can lead to further advances in Learned Sort.

Another important research direction is achieving an implementation that not only maintains theoretical guarantees but also matches or surpasses the empirical performance of the state-of-the-art Learned Sorts. One potential approach to improving real-world efficiency is optimizing memory access patterns to enhance cache efficiency, similar to existing Learned Sort (Kristo et al., 2021a). By carefully structuring memory accesses, reducing cache misses, and leveraging cache-aware data layouts, the performance of PCF Learned Sort could be improved without compromising its theoretical guarantees. Another promising direction is dynamic parameter tuning. Currently, the parameters of PCF Learned Sort are set as $\alpha = \beta = \gamma = \delta = \lfloor n^{3/4} \rfloor$. Dynamically adjusting these parameters based on the input distribution and array size could lead to further performance gains while maintaining theoretical guarantees.

6 Conclusion

We proposed PCF Learned Sort and proved theoretically that its worst-case computational complexity is $\mathcal{O}(n \log n)$ without assumptions and its expected computational complexity is $\mathcal{O}(n \log \log n)$ under mild assumptions on the distribution. We then confirm this computational complexity empirically on both synthetic and real data. This is the first study to support the empirical success of Learned Sort theoretically and provides insight into why Learned Sort is fast.

References

Antonios Antoniadis, Christian Coester, Marek Eliáš, Adam Polak, and Bertrand Simon. Online metric algorithms with untrusted predictions. *ACM Transactions on Algorithms*, 2023.

Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. *ACM Transactions on Parallel Computing*, 2022.

Xingjian Bai and Christian Coester. Sorting with predictions. *Advances in Neural Information Processing Systems*, 2023.

T.-H. Hubert Chan, Enze Sun, and Bo Wang. Generalized sorting with predictions revisited. In *Frontiers of Algorithmics*, 2023.

Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Faster matchings via learned duals. *Advances in Neural Information Processing Systems*, 2021.

Thomas Erlebach, Murilo de Lima, Nicole Megow, and Jens Schlöter. Sorting and hypergraph orientation under uncertainty with predictions. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2023.

Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the Very Large Data Bases Endowment*, 2020.

W Donald Frazer and Archie C McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the Association for Computing Machinery*, 1970.

Sreenivas Gollapudi and Debmalya Panigrahi. Online algorithms for rent-or-buy with expert advice. In *Proceedings of the International Conference on Machine Learning*, 2019.

Yijie Han. Deterministic sorting in o(n log log n) time and linear space. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, 2002.

Yijie Han. Sorting real numbers in o(n sqrt(log n)) time and linear space. *Algorithmica*, 2020.

Yijie Han and Mikkel Thorup. Integer sorting in o(n sqrt(log log n)) expected time and linear space. In *Proceedings of the Symposium on Foundations of Computer Science*, 2002.

Sungjin Im, Ravi Kumar, Aditya Petety, and Manish Purohit. Parsimonious learning-augmented caching. In *Proceedings of the International Conference on Machine Learning*, 2022.

Java. List (java se 21 & jdk 21). URL: `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html#sort(java.util.Comparator)`, 2023. Accessed on 2024-01-18.

Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: a single-pass learned index. In *Proceedings of the international workshop on exploiting artificial intelligence techniques for data management*, 2020.

Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the International Conference on Management of Data*, 2018.

Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *Proceedings of the Conference on Innovative Data Systems Research*, 2019.

Ani Kristo. NYC Yellow Taxi Trips Dataset, Version 2.0, License: CC0 1.0 Universal. URL: `https://doi.org/10.7910/DVN/SSDV7O`, 2021.

Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The case for a learned sorting algorithm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020.

Ani Kristo, Kapil Vaidya, and Tim Kraska. Defeating duplicates: A re-design of the learnedsort algorithm. *arXiv preprint arXiv:2107.03290*, 2021a.

Ani Kristo, Kapil Vaidya, and Tim Kraska. LearnedSort, License: GPL 3.0. URL: `https://github.com/anikristo/LearnedSort`, 2021b. Accessed on 2024-01-18.

Alexandra Anna Lassota, Alexander Lindermayr, Nicole Megow, and Jens Schlöter. Minimalistic predictions to schedule jobs with online precedence constraints. In *Proceedings of the International Conference on Machine Learning*, 2023.

Silvio Lattanzi, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Online scheduling via learned weights. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2020.

Pinyan Lu, Xuandi Ren, Enze Sun, and Yubo Zhang. Generalized sorting with predictions. In *Symposium on Simplicity in Algorithms (SOSA)*, 2021.

Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *Journal of the Association for Computing Machinery*, 2021.

Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. SOSD, License: GPL 3.0. URL: `https://github.com/learnedsystems/SOSD`, 2020. Accessed on 2024-01-18.

Peter McIlroy. Optimistic sorting and information theoretic complexity. In *Proceedings of the ACM-SIAM Symposium on Discrete algorithms*, 1993.

Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. *Advances in Neural Information Processing Systems*, 2018.

Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Communications of the ACM*, 2022.

J Ian Munro and Sebastian Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. In *European Symposium on Algorithms*, 2018.

David R Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 1997.

Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. Deepcache: A deep learning based framework for content caching. In *Proceedings of the Workshop on Network Meets AI & ML*, 2018.

Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. Theoretically-efficient and practical parallel in-place radix sorting. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, 2019.

Tim Peters. Python: list.sort. URL: `https://github.com/python/cpython/blob/main/Objects/listsort.txt`, 2002. Accessed on 2024-01-18.

Manish Purohit, Zoya Svitkina, and Ravi Kumar. Improving online algorithms via ml predictions. *Advances in Neural Information Processing Systems*, 2018.

Dhruv Rohatgi. Near-optimal bounds for online caching with machine learned advice. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2020.

Steven J Ross. The spreadsort high-performance general-case sorting algorithm. In *PDPTA*, 2002.

Shinsaku Sakaue and Taihei Oki. Discrete-convex-analysis-based framework for warm-starting algorithms with predictions. *Advances in Neural Information Processing Systems*, 2022.

Atsuki Sato and Yusuke Matsui. Fast partitioned learned bloom filter. *Advances in Neural Information Processing Systems*, 2023.

Yongho Shin, Changyeol Lee, Gukryeol Lee, and Hyung-Chan An. Improved learning-augmented algorithms for the multi-option ski rental problem via best-possible competitive analysis. In *Proceedings of the International Conference on Machine Learning*, 2023.

Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. Partitioned learned bloom filter. In *International Conference on Learning Representations*, 2021.

Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. Sindex: a scalable learned index for string keys. In *Proceedings of the ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020.

Sepanta Zeighami and Cyrus Shahabi. On distribution dependent sub-logarithmic query time of learned indexing. In *Proceedings of the International Conference on Machine Learning*, 2023.

Sepanta Zeighami and Cyrus Shahabi. Theoretical analysis of learned database operations under distribution shift through distribution learnability. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

A Proofs

Here, we give the proofs omitted in the main paper. In Appendix A.1, we give the proof of Lemma 3.1, which is important for proving the worst-case complexity of PCF Learned Sort. Appendix A.2 and Appendix A.3 give proofs of Lemma 3.3 and Lemma 3.4, respectively, which are important for proving the expected computational complexity of PCF Learned Sort. The quantization-aware version of Lemma 3.4 and Theorem 3.6 is defined and the proof is given in the Appendix A.4.

A.1 Proof of Lemma 3.1

Proof. Let $P(n)$ be the worst-case complexity of our Learned Sort when using the model-based bucketing algorithm $\mathcal{M}$ as assumed in Lemma 3.1 and $\delta = \lfloor n^d \rfloor$. Let $S(n)$ be the worst-case complexity of the "standard" sort algorithm and $R(n)$ be the worst-case complexity of model-based bucketing (including model training and inferences). Since $S(n) = \mathcal{O}(n \log n)$ and the "standard" sort algorithm terminates after a finite number of operations,

$$\exists\, C_1, l_1\ (>0), \quad n \geq 0 \Rightarrow S(n) \leq C_1 + l_1 n \log n. \tag{11}$$

Since $R(n) = \mathcal{O}(n)$ and $\gamma + 1 = \mathcal{O}(n)$,

$$\exists\, n_2, l_2\ (>0), \ \ n \geq n_2 \Rightarrow R(n) \leq l_2 n. \tag{12}$$

$$\exists\, n_3, l_3\ (>0), \ \ n \geq n_3 \Rightarrow \gamma + 1 \leq l_3 n. \tag{13}$$

In the following, we prove $T(n) = \mathcal{O}(n \log\log n)$ by mathematical induction.

First, for $n < \max(n_2, n_3, \tau) =: n_0$, there exists a constant $C\ (>0)$ such that $P(n) \leq C$. That is, for $n < n_0$, our Learned Sort terminates in a finite number of operations. This is because, since $\delta < n$, the bucket will either be smaller than the original array length n, or the bucket will be immediately sorted by the "standard" sort algorithm.

Next, assume that there exists a constant $C\ (>0)$ and $l\ (>0)$ such that $P(n) \leq C + ln \log n$ for all $n < k$, where k is an integer such that $k \geq n_0$. Let $\mathcal{S}_{\gamma,k}$ be the set consisting of all $(\gamma+1)$-dimensional vectors of positive integers whose sum is k, i.e., $\mathcal{S}_{\gamma,k} \coloneqq \left\{ \boldsymbol{s} \in \mathbb{Z}^{\gamma}_{\geq 0} \mid \sum_{i=1}^{\gamma+1} s_i = k \right\}$. Then,

$$\begin{aligned}
P(k) &\leq R(k) + \max_{\boldsymbol{s}\in\mathcal{S}_{\gamma,k}} \sum_{i=1}^{\gamma+1} \left\{ \mathbb{1}[s_i \geq \lfloor k^d \rfloor] \cdot S(s_i) + \mathbb{1}[s_i < \lfloor k^d \rfloor] \cdot P(s_i) \right\} \\
&\leq l_2 k + \max_{\boldsymbol{s}\in\mathcal{S}_{\gamma,k}} \sum_{i=1}^{\gamma+1} \left\{ \mathbb{1}[s_i \geq \lfloor k^d \rfloor] \cdot (C_1 + l_1 s_i \log s_i) + \mathbb{1}[s_i < \lfloor k^d \rfloor] \cdot (C + l s_i \log s_i) \right\} \\
&\leq l_2 k + \max_{\boldsymbol{s}\in\mathcal{S}_{\gamma,k}} \sum_{i=1}^{\gamma+1} \left\{ (C_1 + l_1 s_i \log s_i) + \left(C + l s_i \log k^d\right) \right\} \\
&\leq l_2 k + C_1(\gamma+1) + l_1 k \log k + C(\gamma+1) + lk \log k^d \\
&\leq l_2 k + C_1 l_3 k + l_1 k \log k + C l_3 k + lkd \log k \\
&\leq \{l_2 + C_1 l_3 + l_1 + C l_3 - l(1-d)\}\, k \log k + (C + lk \log k).
\end{aligned} \tag{14}$$

Therefore, if we take l such that

$$\frac{l_2 + C_1 l_3 + l_1 + C l_3}{1-d} \leq l, \tag{15}$$

then $P(k) \leq C + lk \log k$ (note that the left side of the following equation is a constant independent of k).

Hence, by mathematical induction, it is proved that there exists a constant $C\ (>0)$ and $l\ (>0)$ such that $P(n) \leq C + ln \log n$ for all $n \in \mathbb{N}$. □

A.2 Proof of Lemma 3.3

Proof. The proof approach is the same as in Lemma 3.1, but in Lemma 3.3, the "expected" computational complexity is bounded. The following two randomnesses are considered for computing the "expected" computational complexity: (i) the randomness with which n elements are independently sampled according to the probability density function $f(x)$ in the process of forming the input array $\boldsymbol{x}$, and (ii) the randomness of the PCF Learned Sort algorithm sampling α elements from the input array $\boldsymbol{x}$ for training the PCF.

Let $T(n)$ be the expected complexity of our Learned Sort when using the model-based bucketing algorithm $\mathcal{M}$ as assumed in Lemma 3.3 and $\delta = \lfloor n^d \rfloor$. Let $S(n)$ be the expected complexity of the "standard" sort algorithm and $R(n)$ be the expected complexity of model-based bucketing (including model training and inferences). Since $S(n) = \mathcal{O}(n \log n)$ and the "standard" sort algorithm terminates after a finite number of operations,

$$\exists\, C_1, l_1\ (>0),\quad n \geq 0 \Rightarrow S(n) \leq C_1 + l_1 n \log n. \tag{16}$$

Since $R(n) = \mathcal{O}(n)$, $\Pr[\exists j, |\boldsymbol{c}_j| \geq \lfloor n^d \rfloor] = \mathcal{O}(1/\log n)$, and $\gamma + 1 = \mathcal{O}(n)$,

$$\exists\, n_2, l_2\ (>0),\ \ n \geq n_2 \Rightarrow R(n) \leq l_2 n, \tag{17}$$

$$\exists\, n_3, l_3\ (>0),\ \ n \geq n_3 \Rightarrow \Pr[\exists j, |\boldsymbol{c}_j| \geq \lfloor n^d \rfloor] \leq \frac{l_3}{\log n}, \tag{18}$$

$$\exists\, n_4, l_4\ (>0),\ \ n \geq n_4 \Rightarrow \gamma + 1 \leq l_4 n. \tag{19}$$

In the following, we prove $T(n) = \mathcal{O}(n \log\log n)$ by mathematical induction.

First, for $n < \max(n_2, n_3, n_4, \tau) =: n_0$, there exists a constant $C\ (>0)$ such that $T(n) \leq C$. That is, for $n < n_0$, our Learned Sort terminates in a finite number of operations.

Next, assume that there exists a constant $C\ (>0)$ and $l\ (>0)$ such that $T(n) \leq C + ln \log\log n$ for all $n < k$, where k is an integer such that $k \geq n_0$. Then, from $k \geq n_2$,

$$\begin{aligned} T(k) &\leq R(k) + \mathbb{E}\left[\sum_{j=1}^{\gamma+1} \mathbb{1}[|\boldsymbol{c}_j| \geq \lfloor k^d \rfloor] \cdot S(|\boldsymbol{c}_j|) + \mathbb{1}[|\boldsymbol{c}_j| < \lfloor k^d \rfloor] \cdot T(|\boldsymbol{c}_j|)\right] \\ &\leq l_2 k + \Pr\left[\exists j, |\boldsymbol{c}_j| \geq \lfloor k^d \rfloor\right] \cdot \mathbb{E}\left[\sum_{j=1}^{\gamma+1} S(|\boldsymbol{c}_j|)\right] + \mathbb{E}\left[\sum_{j=1}^{\gamma+1} \mathbb{1}[|\boldsymbol{c}_j| < \lfloor k^d \rfloor] \cdot T(|\boldsymbol{c}_j|)\right] \\ &\leq l_2 k + \Pr[\exists j, |\boldsymbol{c}_j| \geq \lfloor k^d \rfloor] \cdot \{C_1(\gamma+1) + l_1 k \log k\} + \mathbb{E}\left[\sum_{i=1}^{\gamma+1} T(\min(\lfloor k^d \rfloor, |\boldsymbol{c}_j|))\right]. \end{aligned} \tag{20}$$

Here, from $k \geq n_3$ and $k \geq n_4$,

$$\begin{aligned} \Pr[\exists j, |\boldsymbol{c}_j| \geq \lfloor k^d \rfloor] \cdot \{C_1(\gamma+1) + l_1 k \log k\} &\leq \frac{l_3}{\log k} \cdot (C_1 l_4 k + l_1 k \log k) \\ &\leq (C_1 l_3 l_4 + l_1 l_3) k. \end{aligned} \tag{21}$$

Also, from the assumption of induction and $k \geq n_4$,

$$\begin{aligned} \mathbb{E}\left[\sum_{i=1}^{\gamma+1} T(\min(\lfloor k^d \rfloor, |\boldsymbol{c}_j|))\right] &\leq \mathbb{E}\left[\sum_{i=1}^{\gamma+1} \left\{C + l \cdot \min(\lfloor k^d \rfloor, |\boldsymbol{c}_j|) \log\log \min(\lfloor k^d \rfloor, |\boldsymbol{c}_j|)\right\}\right] \\ &\leq \mathbb{E}\left[C(\gamma+1) + \sum_{i=1}^{\gamma+1} l \cdot |\boldsymbol{c}_j| \log\log \lfloor k^d \rfloor\right] \\ &\leq C l_4 k + l k \log\log \lfloor k^d \rfloor \\ &\leq C l_4 k + l k \log d + l k \log\log k. \end{aligned} \tag{22}$$

Therefore,

$$\begin{aligned} T(k) &\leq l_2 k + (C_1 l_3 l_4 + l_1 l_3)k + C l_4 k + lk \log d + lk \log\log k \\ &\leq \left\{ l_2 + C_1 l_3 l_4 + l_1 l_3 + C l_4 - l \log \frac{1}{d} \right\} k + (C + lk \log\log k). \end{aligned} \tag{23}$$

Therefore, if we take l such that

$$\frac{l_2 + C_1 l_3 l_4 + l_1 l_3 + C l_4}{\log \frac{1}{d}} \leq l, \tag{24}$$

then $T(k) \leq C + lk \log\log k$ (note that the left side of Equation (24) is a constant independent of k).

Hence, by mathematical induction, it is proved that there exists a constant C (> 0) and l (> 0) such that $T(n) \leq C + ln \log\log n$ for all $n \in \mathbb{N}$. □

A.3 Proof of Lemma 3.4

We first present the following lemma to prove Lemma 3.4.

Lemma A.1. *Let $\boldsymbol{e}$ ($\in \mathcal{I}^n$) be a sorted version of $\boldsymbol{x}_\mathcal{I}$ ($\in \mathcal{I}^n$) and $\Delta \coloneqq (x_{\max} - x_{\min})/\beta$ (where $x_{\min}$ and $x_{\max}$ are the minimum and maximum values of $\boldsymbol{x}_\mathcal{I}$, respectively).*

Also, define the set $\mathcal{S}_r$ and $\mathcal{T}_r$ as follows ($r = 1, \ldots, n$):

$$\mathcal{S}_r = \{k \mid e_{\max(1, r-\delta/2)} + \Delta < e_k \leq e_r - \Delta\}, \qquad \mathcal{T}_r = \{k \mid e_r + \Delta \leq e_k < e_{\min(r+\delta/2, n)} - \Delta\}. \tag{25}$$

Using this definition, define Y_{jr}, Z_{jr}, Y_r, Z_r as follows ($j = 1, \ldots, \alpha,\ r = 1, \ldots, n$):

$$Y_{jr} = \begin{cases} 1 & (j \in \mathcal{S}_r) \\ 0 & (\text{else}) \end{cases}, \qquad Z_{jr} = \begin{cases} 1 & (j \in \mathcal{T}_r) \\ 0 & (\text{else}) \end{cases}, \tag{26}$$

$$Y_r = \sum_{j=1}^{\alpha} Y_{jr}, \qquad Z_r = \sum_{j=1}^{\alpha} Z_{jr}. \tag{27}$$

If the size of the bucket to which e_r is allocated is greater than or equal to δ, then the following holds:

$$\left(r \geq \frac{\delta}{2} + 1 \ \wedge\ Y_r \leq \left\lfloor \frac{\alpha}{\gamma} \right\rfloor \right) \ \vee\ \left(r \leq n - \frac{\delta}{2} \ \wedge\ Z_r \leq \left\lfloor \frac{\alpha}{\gamma} \right\rfloor \right). \tag{28}$$

Proof. We prove the contraposition of the lemma. That is, we prove that e_r is allocated to a bucket smaller than δ under the assumption that $\left(r < \frac{\delta}{2} + 1 \ \vee\ Y_r > \left\lfloor \frac{\alpha}{\gamma} \right\rfloor \right) \ \wedge\ \left(r > n - \frac{\delta}{2} \ \vee\ Z_r > \left\lfloor \frac{\alpha}{\gamma} \right\rfloor \right)$.

For convenience, we hypothetically define $e_0 = -\infty, e_{n+1} = \infty$, and assign e_0 to the 0th bucket and e_{n+1} to the $(\gamma + 2)$-th bucket. The size of the 0th bucket and the $(\gamma + 2)$-th bucket are both 1.

First, we prove that e_r and $e_{\min(r+\delta/2, n+1)}$ are assigned to different buckets. When $n - \delta/2 < r \leq n$, e_r and $e_{\min(r+\delta/2, n+1)} = e_{n+1}$ are obviously assigned to different buckets. When $r \leq n - \delta/2$, the ID of the bucket to which e_r is assigned is

$$\begin{aligned} \lfloor \tilde{F}(e_r)\gamma \rfloor + 1 &= \left\lfloor \frac{\gamma}{\alpha} b_{i(e_r)} \right\rfloor + 1 \\ &\leq \frac{\gamma}{\alpha} b_{i(e_r)} + 1 \\ &= \frac{\gamma}{\alpha} |\{j \mid i(a_j) \leq i(e_r)\}| + 1 \\ &\leq \frac{\gamma}{\alpha} |\{j \mid a_j \leq e_r + \Delta\}| + 1. \end{aligned} \tag{29}$$

The ID of the bucket to which $e_{\min(r+\delta/2,n+1)} = e_{r+\delta/2}$ is assigned is

$$\begin{aligned}\lfloor \tilde{F}(e_{r+\delta/2})\gamma \rfloor + 1 &= \left\lfloor \frac{\gamma}{\alpha} b_{i(e_{r+\delta/2})} \right\rfloor + 1 \\ &> \frac{\gamma}{\alpha} b_{i(e_{r+\delta/2})} \\ &= \frac{\gamma}{\alpha} \left| \{j \mid i(a_j) \leq i(e_{r+\delta/2})\} \right| \\ &\geq \frac{\gamma}{\alpha} \left| \{j \mid a_j \leq e_{r+\delta/2} - \Delta\} \right|.\end{aligned} \tag{30}$$

Thus, taking the difference between these two bucket IDs,

$$\begin{aligned}\left(\lfloor \tilde{F}(e_{r+\delta/2})\gamma \rfloor + 1\right) - \left(\lfloor \tilde{F}(e_r)\gamma \rfloor + 1\right) &> \frac{\gamma}{\alpha} \left| \{j \mid a_j \leq e_{r+\delta/2} - \Delta\} \right| - \left(\frac{\gamma}{\alpha} \left| \{j \mid a_j \leq e_r + \Delta\} \right| + 1 \right) \\ &= \frac{\gamma}{\alpha} \left| \{j \mid e_r + \Delta < a_j \leq e_{r+\delta/2} - \Delta\} \right| - 1 \\ &= \frac{\gamma}{\alpha} |\mathcal{T}_r| - 1 \\ &= \frac{\gamma}{\alpha} \sum_{j=1}^{\alpha} Z_{jr} - 1 \\ &= \frac{\gamma}{\alpha} Z_r - 1 \\ &\geq 0.\end{aligned} \tag{31}$$

Therefore, e_r and $e_{\min(r+\delta/2,n+1)}$ are assigned to different buckets.

In the same way, we can prove that $e_{\max(0,r-\delta/2)}$ and e_r are also assigned to different buckets. Thus, the size of the bucket to which e_r is assigned is at most $\delta - 1$ (at most from $e_{\max(0,r-\delta/2)+1}$ to $e_{\min(r+\delta/2,n+1)-1}$), and the contraposition of the lemma is proved. □

Using Lemma A.1, we can prove Lemma 3.4.

Proof. Let $q = \max_y \int_y^{y+\Delta} f_{\mathcal{I}}(x)dx$ (where y is a value such that $(y, y+\Delta) \subseteq \mathcal{I}$). Then, from $\sigma_1 \leq f(x) \leq \sigma_2$ for all $x \in \mathcal{I}$,

$$\begin{aligned}q &\leq \frac{\max_y \int_y^{y+\Delta} f(y)dy}{\int_{\mathcal{I}} f(x)dx} \\ &\leq \frac{\max_y \int_y^{y+\Delta} \sigma_2 dy}{\int_{\mathcal{I}} \sigma_1 dx} \\ &\leq \frac{\sigma_2 \Delta}{\sigma_1 (x_{\max} - x_{\min})} \\ &= \frac{\sigma_2}{\sigma_1 \beta}.\end{aligned} \tag{32}$$

Thus, when $r \geq \frac{\delta}{2} + 1$,

$$\begin{aligned}\mathbb{E}\left[\frac{\delta}{2} - |\mathcal{S}_r|\right] &= \mathbb{E}\left[\frac{\delta}{2} - \left|\{k \mid e_{r-\delta/2} + \Delta < e_k \leq e_r - \Delta\}\right|\right] \\ &= \mathbb{E}\left[\left|\{k \mid e_{r-\delta/2} < e_k \leq e_{r-\delta/2} + \Delta\}\right|\right] + \mathbb{E}\left[|\{k \mid e_r - \Delta < e_k \leq e_r\}|\right] \\ &\leq nq + nq \\ &\leq \frac{2\sigma_2 n}{\sigma_1 \beta}.\end{aligned} \tag{33}$$

Thus, when $r \geq \frac{\delta}{2} + 1$,

$$\begin{aligned}
\mathbb{E}[Y_r] &= \frac{\alpha}{n}\mathbb{E}\left[|\mathcal{S}_r|\right] \\
&= \frac{\alpha}{n}\left(\frac{\delta}{2} - \mathbb{E}\left[\frac{\delta}{2} - |\mathcal{S}_r|\right]\right) \\
&\geq \frac{\alpha\delta}{2n} - \frac{2\sigma_2\alpha}{\sigma_1\beta} \\
&= \frac{\alpha K}{\gamma}.
\end{aligned} \tag{34}$$

Here, when $K \geq 1$, we have

$$0 \leq 1 - \frac{\alpha}{\gamma\mathbb{E}[Y_r]} < 1. \tag{35}$$

Therefore, from the Chernoff bound,

$$\begin{aligned}
\Pr\left[Y_r \leq \frac{\alpha}{\gamma}\right] &= \Pr\left[Y_r \leq \left\{1 - \left(1 - \frac{\alpha}{\gamma\mathbb{E}[Y_r]}\right)\right\}\mathbb{E}\left[Y_r\right]\right] \\
&\leq \exp\left\{-\frac{1}{2}\left(1 - \frac{\alpha}{\gamma\mathbb{E}[Y_r]}\right)^2 \mathbb{E}\left[Y_r\right]\right\} \\
&\leq \exp\left\{-\frac{\alpha K}{2\gamma}\left(1 - \frac{1}{K}\right)^2\right\}.
\end{aligned} \tag{36}$$

In the same way, we can prove that when $r \leq n - \frac{\delta}{2}$,

$$\Pr\left[Z_r \leq \frac{\alpha}{\gamma}\right] \leq \exp\left\{-\frac{\alpha K}{2\gamma}\left(1 - \frac{1}{K}\right)^2\right\}. \tag{37}$$

Thus, by defining E_r to be the event "e_r is allocated to a bucket with size greater than or equal to δ," from Lemma A.1,

$$\begin{aligned}
\Pr[E_r] &\leq \Pr\left[\left(r \geq \frac{\delta}{2} + 1 \ \wedge \ Y_r \leq \left\lfloor\frac{\alpha}{\gamma}\right\rfloor\right) \ \vee \ \left(r \leq n - \frac{\delta}{2} \ \wedge \ Z_r \leq \left\lfloor\frac{\alpha}{\gamma}\right\rfloor\right)\right] \\
&\leq \Pr\left[\left(r \geq \frac{\delta}{2} + 1 \ \wedge \ Y_r \leq \left\lfloor\frac{\alpha}{\gamma}\right\rfloor\right)\right] + \Pr\left[\left(r \leq n - \frac{\delta}{2} \ \wedge \ Z_r \leq \left\lfloor\frac{\alpha}{\gamma}\right\rfloor\right)\right] \\
&\leq \begin{cases} \exp\left\{-\frac{\alpha K}{2\gamma}\left(1 - \frac{1}{K}\right)^2\right\}, & \left(r < \frac{\delta}{2} + 1 \ \vee \ r > n - \frac{\delta}{2}\right) \\ 2\exp\left\{-\frac{\alpha K}{2\gamma}\left(1 - \frac{1}{K}\right)^2\right\}, & \text{(else)} \end{cases} \\
&\leq 2\exp\left\{-\frac{\alpha K}{2\gamma}\left(1 - \frac{1}{K}\right)^2\right\}.
\end{aligned} \tag{38}$$

Therefore,

$$\mathbb{E}\left[\sum_{r=1}^{n} \mathbb{1}[E_r]\right] \leq 2n\exp\left\{-\frac{\alpha K}{2\gamma}\left(1 - \frac{1}{K}\right)^2\right\} \tag{39}$$

Then, noting that the number of buckets with size greater than or equal to δ is less than or equal to $\sum_{r=1}^{n} \mathbb{1}[E_r]/\delta$,

$$\mathbb{E}\left[|\{j \mid |\boldsymbol{c}_j| > \delta\}|\right] \leq \frac{2n}{\delta}\exp\left\{-\frac{\alpha K}{2\gamma}\left(1 - \frac{1}{K}\right)^2\right\}. \tag{40}$$

Then, from Markov's inequality, we have

$$\begin{aligned}\Pr[\exists j, |\boldsymbol{c}_j| > \delta] &= \Pr[|\{j \mid |\boldsymbol{c}_j| > \delta\}| \geq 1] \\ &\leq \frac{2n}{\delta} \exp\left\{-\frac{\alpha K}{2\gamma}\left(1 - \frac{1}{K}\right)^2\right\}. \end{aligned} \tag{41}$$

□

A.4 The Quantization-Aware Version of Lemma 3.4 and Theorem 3.6

In general, computers represent numbers in a finite number of bits, so the numbers they handle are inherently discrete. However, Theorem 3.6, which states that the expected computational complexity of PCF Learned Sort is $\mathcal{O}(n \log\log n)$, does not cover discrete distributions. Here, we define the quantization process by which a computer represents numbers in finite bits and then show that the expected computational complexity of PCF Learned Sort is still $\mathcal{O}(n \log\log n)$, under the assumption that "the quantization is fine enough."

First, we assume the sampling and quantization process is as follows:

Assumption A.2. For a range of values $\mathcal{D}$ ($\subseteq \mathbb{R}$), define m ($\in \mathbb{N}$) contiguous regions $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_m$ such that (i) they are disjoint from each other and (ii) together they form $\mathcal{D}$. For each region, determine representative values $r_1, r_2, \ldots, r_m$. Here, $r_1, r_2, \ldots, r_m$ are values contained in $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_m$, respectively. For a value x ($\in \mathcal{D}$), the quantized value of x, x', is obtained as $x' = r_i$, where i is the (only) i such that $x \in \mathcal{D}_i$. The value x is sampled according to the probability density function $f(x)$, but a computer keeps x' (instead of x) in $\log_2 m$ bits with some quantization error.

Also, for the interval $\mathcal{I} \subseteq \mathcal{D}$, we define $\eta(\mathcal{I})$ as follows.

Definition A.3. $\eta(\mathcal{I})$ is the maximum width of $\mathcal{D}_i$ that intersects with interval $\mathcal{I}$, i.e.,

$$\eta(\mathcal{I}) \coloneqq \max\left\{|\mathcal{D}_i| \mid \mathcal{D}_i \cup \mathcal{I} \neq \varnothing, i = 1, \ldots, m\right\}, \tag{42}$$

where $|\mathcal{D}_i|$ is the width of the range $\mathcal{D}_i$.

We can prove that $\eta(\mathcal{I})$ is the upper bound of the quantization error of x in $\mathcal{I}$, i.e., $|x - x'| \leq \eta(\mathcal{I})$ when $x \in \mathcal{I}$.

Now, the assumption that the quantization is "fine enough" is specifically defined as follows.

Assumption A.4. (Recall that our PCF Learned Sort recursively calls its own algorithm. Each time of recursion, the range of values of interest $\mathcal{I}$ changes, and the length of the array of interest n also changes.) For all $\mathcal{I}$ and n that appear in the algorithm, the following holds:

$$\frac{\beta\eta(\mathcal{I})}{|\mathcal{I}|} \leq \frac{1}{2}. \tag{43}$$

We show intuitively and empirically that this is a satisfactory assumption.

First, to show intuitively that this assumption is satisfactory, we give an example. Let $\mathcal{I}$ be a range of values that can be represented by a 64-bit double (1 bit for the sign, 11 bits for the exponent part, and 52 bits for the mantissa part), that is, $\mathcal{I} = [-1.79 \times 10^{308}, 1.79 \times 10^{308}]$. The quantization is performed by mapping each value to the nearest number that can be represented by a 64-bit double. In this setting, $\eta(\mathcal{I}) = 1.99 \times 10^{292}$, $|\mathcal{I}| = 3.59 \times 10^{308}$. $\eta(\mathcal{I})/|\mathcal{I}| = 5.55 \times 10^{-17}$. Thus, for usual β, $\beta \leq 9 \times 10^{15}$, Equation (43) holds.

Second, we show that Equation (43) is a satisfactory assumption empirically. In the 1,280 measurements, where 10 measurements each for 16 different n($\in \{10^3, 2 \times 10^3, 5 \times 10^3, \ldots, 10^8\}$) on 8 different datasets, 5.23×10^7 pairs of $(\mathcal{I}, \beta)$ appeared (we set $\beta = \lfloor n^{3/4} \rfloor$), and the left side of Equation (43) is at most 8.11×10^{-9}, indicating that Equation (43) is always true with a margin.

Under this definition and assumption about quantization, we can prove the quantization-aware version of Lemma 3.4.

Lemma A.5 (Quantization-aware version of Lemma 3.4)**.** *Let σ_1 and σ_2 be respectively the lower and upper bounds of the probability density distribution $f(x)$ in $\mathcal{D}$, and assume that $0 < \sigma_1 \leq \sigma_2 < \infty$. That is, $x \in \mathcal{D} \Rightarrow \sigma_1 \leq f(x) \leq \sigma_2$. Also, let $\boldsymbol{x}'_{\mathcal{I}}$ be the array created by the quantization of $\boldsymbol{x}$ ($\in \mathcal{I}^n$) in the manner defined in Assumption A.2.*

Then, in model-based bucketing of $\boldsymbol{x}'_{\mathcal{I}}$ ($\in \mathcal{I}^n$) to $\{\boldsymbol{c}_j\}_{j=1}^{\gamma+1}$ using $\mathcal{M}_{\mathrm{PCF}}$, the following holds for any interval $\mathcal{I}$ ($\subseteq \mathcal{D}$) under Assumption A.4:

$$K \geq 1 \Rightarrow \Pr[\exists j, |\boldsymbol{c}_j| > \delta] \leq \frac{2n}{\delta} \exp\left\{-\frac{\alpha K'}{2\gamma}\left(1 - \frac{1}{K'}\right)^2\right\}, \tag{44}$$

where

$$K' \coloneqq \frac{\gamma\delta}{2n} - \frac{4\sigma_2\gamma}{\sigma_1\beta}. \tag{45}$$

The proof of Lemma A.5 is done in the same way as Lemma 3.4. That is, we first prove the following lemma.

Lemma A.6 (Quantization-aware version of Lemma A.6)**.** *Let $\boldsymbol{e}'$ ($\in \mathcal{I}^n$) be a sorted version of $\boldsymbol{x}'_{\mathcal{I}}$ ($\in \mathcal{I}^n$) and $\Delta \coloneqq (x'_{\max} - x'_{\min})/\beta$ (where $x'_{\min}$ and $x'_{\max}$ are the minimum and maximum values of $\boldsymbol{x}'_{\mathcal{I}}$, respectively). Also, define the set $\mathcal{S}'_r, \mathcal{T}'_r$ as follows ($r = 1, \ldots, n$):*

$$\mathcal{S}'_r = \{k \mid e_{\max(1, r-\delta/2)} + \Delta + 2\eta(\mathcal{I}) < e_k \leq e_r - \Delta - 2\eta(\mathcal{I})\}, \tag{46}$$

$$\mathcal{T}'_r = \{k \mid e_r + \Delta + 2\eta(\mathcal{I}) \leq e_k < e_{\min(r+\delta/2, n)} - \Delta - 2\eta(\mathcal{I})\}. \tag{47}$$

Using this definition, define $Y'_{jr}, Z'_{jr}, Y'_r, Z'_r$ as follows ($j = 1, \ldots, \alpha,\ r = 1, \ldots, n$):

$$Y'_{jr} = \begin{cases} 1 & (j \in \mathcal{S}'_r) \\ 0 & (\text{else}) \end{cases}, \qquad Z'_{jr} = \begin{cases} 1 & (j \in \mathcal{T}'_r) \\ 0 & (\text{else}) \end{cases}, \tag{48}$$

$$Y'_r = \sum_{j=1}^{\alpha} Y'_{jr}, \qquad Z'_r = \sum_{j=1}^{\alpha} Z'_{jr}. \tag{49}$$

If the size of the bucket to which e'_r is allocated is greater than or equal to δ, then the following holds:

$$\left(r \geq \frac{\delta}{2} + 1 \ \wedge\ Y'_r \leq \left\lfloor \frac{\alpha}{\gamma} \right\rfloor\right) \ \vee\ \left(r \leq n - \frac{\delta}{2} \ \wedge\ Z'_r \leq \left\lfloor \frac{\alpha}{\gamma} \right\rfloor\right). \tag{50}$$

Proof. The proof method is exactly the same as for Lemma A.1. That is, by taking the difference between the IDs of the buckets to which $e'_{r+\delta/2}$ and e'_r are assigned,

$$\begin{aligned} &\left(\lfloor \tilde{F}(e'_{r+\delta/2})\gamma \rfloor + 1\right) - \left(\lfloor \tilde{F}(e'_r)\gamma \rfloor + 1\right) \\ >&\frac{\gamma}{\alpha}\left|\{j \mid a_j \leq e_{r+\delta/2} - \Delta - 2\eta(\mathcal{I})\}\right| - \left(\frac{\gamma}{\alpha}\left|\{j \mid a_j \leq e_r + \Delta + 2\eta(\mathcal{I})\}\right| + 1\right) \\ =&\frac{\gamma}{\alpha}\left|\{j \mid e_r + \Delta + 2\eta(\mathcal{I}) < a_j \leq e_{r+\delta/2} - \Delta - 2\eta(\mathcal{I})\}\right| - 1 \\ =&\frac{\gamma}{\alpha} Z'_r - 1 \\ \geq&0, \end{aligned} \tag{51}$$

when $Z'_r > \left\lfloor \frac{\alpha}{\gamma} \right\rfloor$. Thus, we can prove that when

$$\left(r < \frac{\delta}{2} + 1 \ \vee\ Y'_r > \left\lfloor \frac{\alpha}{\gamma} \right\rfloor\right) \ \wedge\ \left(r > n - \frac{\delta}{2} \ \vee\ Z'_r > \left\lfloor \frac{\alpha}{\gamma} \right\rfloor\right) \tag{52}$$

holds, $e'_{r+\delta/2}$ and e'_r are assigned to the different bucket and $e'_{r-\delta/2}$ and e'_r are assigned to the different bucket. Then, the contraposition of the lemma is proved. □

Using Lemma A.6, we can prove Lemma A.5.

Proof. Let $q' = \max_y \int_y^{y+\Delta 2\eta(\mathcal{I})} f_{\mathcal{I}}(x)dx$ (where y is a value such that $(y, y+\Delta+2\eta(\mathcal{I}) \subseteq \mathcal{I}$). Then, from $\sigma_1 \leq f(x) \leq \sigma_2$ for all $x \in \mathcal{I}$,

$$\begin{aligned} q' &\leq \frac{\max_y \int_y^{y+\Delta+2\eta(\mathcal{I})} f(y)dy}{\int_{\mathcal{I}} f(x)dx} \\ &\leq \frac{\max_y \int_y^{y+\Delta+2\eta(\mathcal{I})} \sigma_2 dy}{\int_{\mathcal{I}} \sigma_1 dx} \\ &= \frac{\sigma_2(\Delta+2\eta(\mathcal{I}))}{\sigma_1(x'_{\max}-x'_{\min})} \\ &\leq \frac{\sigma_2}{\sigma_1\beta} \cdot \left(1+\frac{2\beta\eta(\mathcal{I})}{|\mathcal{I}|}\right) \\ &\leq \frac{2\sigma_2}{\sigma_1\beta}. \end{aligned} \tag{53}$$

The last inequality is obtained by Assumption A.4.

Thus, when $r \geq \frac{\delta}{2}+1$,

$$\begin{aligned} \mathbb{E}\left[\frac{\delta}{2}-|\mathcal{S}'_r|\right] &= \mathbb{E}\left[\frac{\delta}{2}-\left|\{k \mid e_{r-\delta/2}+\Delta+2\eta(\mathcal{I}) < e_k \leq e_r-\Delta-2\eta(\mathcal{I})\}\right|\right] \\ &= \mathbb{E}\left[\left|\{k \mid e_{r-\delta/2} < e_k \leq e_{r-\delta/2}+\Delta+2\eta(\mathcal{I})\}\right|\right] + \mathbb{E}\left[|\{k \mid e_r-\Delta-2\eta(\mathcal{I}) < e_k \leq e_r\}|\right] \\ &\leq nq' + nq' \\ &\leq \frac{4\sigma_2 n}{\sigma_1\beta}. \end{aligned} \tag{54}$$

Thus, when $r \geq \frac{\delta}{2}+1$,

$$\begin{aligned} \mathbb{E}[Y'_r] &= \frac{\alpha}{n}\mathbb{E}\left[|\mathcal{S}'_r|\right] \\ &= \frac{\alpha}{n}\left(\frac{\delta}{2}-\mathbb{E}\left[\frac{\delta}{2}-|\mathcal{S}'_r|\right]\right) \\ &\geq \frac{\alpha\delta}{2n}-\frac{4\sigma_2\alpha}{\sigma_1\beta} \\ &= \frac{\alpha K'}{\gamma}. \end{aligned} \tag{55}$$

From this point forward, by proceeding in exactly the same way as the proof of Lemma 3.4, we can prove the following using Lemma A.1:

$$K' \geq 1 \Rightarrow \Pr[\exists j, |\mathbf{c}_j| > \delta] \leq \frac{2n}{\delta}\exp\left\{-\frac{\alpha K'}{2\gamma}\left(1-\frac{1}{K'}\right)^2\right\}. \tag{56}$$

□

Using Lemma A.5, we can prove the following theorem.

Theorem A.7 (Quantization-aware version of Theorem 3.6)**.** *Let σ_1 and σ_2 be the lower and upper bounds, respectively, of the probability density distribution $f(x)$ in $\mathcal{D}$, and assume that $0 < \sigma_1 \leq \sigma_2 < \infty$. Also, assume that the input array is quantized in a way that satisfies Assumption A.2 and Assumption A.4. Then, the expected complexity of PCF Learned Sort with $\mathcal{M}_{\text{PCF}}$ as the bucketing method and $\alpha = \beta = \gamma = \delta = \lfloor n^{3/4} \rfloor$ is $\mathcal{O}(n \log\log n)$.*

Proof. When $\alpha = \beta = \gamma = \lfloor n^{3/4} \rfloor$, the computational complexity for model-based bucketing is $\mathcal{O}(n)$. Since $K' = \Omega(\sqrt{n})$ when $\alpha = \beta = \gamma = \delta = \lfloor n^{3/4} \rfloor$, $K' \geq 1$ for sufficiently large n, and

$$\frac{2n}{\delta} \exp\left\{ -\frac{\alpha K'}{2\gamma} \left(1 - \frac{1}{K'}\right)^2 \right\} = \mathcal{O}(n^{\frac{1}{4}} \exp(-\sqrt{n})) \leq \mathcal{O}\left(\frac{1}{\log n}\right). \tag{57}$$

Therefore, from Lemma 3.3 and Lemma A.5, the expected computational complexity of PCF Learned Sort is $\mathcal{O}(n \log \log n)$. □