BitTuner: A Toolbox for Automatically Configuring Learned Data Compressors

Qiyu Liu*, Yuxin Luo*, Mengke Cui*, Siyuan Han†, Jingshu Peng‡, Jin Li§, Lei Chen†¶
*Southwest University, †HKUST, ‡ByteDance, §Harvard University, ¶HKUST (GZ)
{qyliu.cs}@gmail.com, {luorainstar, tashimeng}@email.swu.edu.cn, {shanaj, leichen}@cse.ust.hk,
{jingshu.peng}@bytedance.com, {jinli}@g.harvard.edu

***Abstract*—Compressing sorted keys is a fundamental operation in data management and information retrieval. Inspired by the success of *learned index*, recent studies apply simple ML models to compress large-scale sorted keys, leading to the concept of *learned compressor*. Intuitively, learned compressors *losslessly* encode sorted keys by approximating them with an error-bounded ML model (e.g., a piecewise linear function) and a residual array to ensure lossless key restoration. However, determining the optimal configuration of underlying ML models to maximize compression efficacy is non-trivial.**

To address this, by analyzing the distribution characteristics of input keys, we propose BitTuner, a novel framework that automatically sets model hyper-parameters to *provably* achieve the best compression ratio. We demonstrate BitTuner on two real-world scenarios: inverted list compression and vectorDB codebook compression. The results show that BitTuner automates the parameter tuning procedure and achieves superior compression efficacy when compared to generic compressors such as LZ4 and LZMA.

***Index Terms*—learned index, data compression, automatic parameter tuning**

I. Introduction

With the rapid growth of data-intensive applications, vast volumes of data across different domains are being stored and queried by key-value (KV) databases [1]. For 10 billion sorted keys of type `uint64_t`, storing raw keys requires approximately 80 GiB of memory. Rather than disk-based solutions, a more efficient alternative is to *losslessly compress* the raw keys to fit within memory budgets, enabling fully in-memory query processing while incurring negligible overheads from in-situ decompression. Beyond KV stores, compressing sorted keys is also a fundamental operator in information retrieval (IR), vectorDB, DNA analytics, and more. The formal definition of *Sorted Keys Compression* is provided below.

Definition 1 (Sorted Keys Compression)**.** *Given a list of N sorted integer keys $\mathcal{K}$, the goal of compressing $\mathcal{K}$ is to find a compact representation $\mathcal{K}^c$ such that $\mathcal{K}^c[i] = \mathcal{K}[i]$ for $\forall i = 1, 2, \cdots, N$. The compression ratio is defined as* $\mathsf{size}(\mathcal{K}^c)/\mathsf{size}(\mathcal{K})$.

A recent experimental survey [2] summarizes the common list compressors, such as P4Delta, Elias-Fano index, entropy encoding family, etc. In contrast to these conventional methodologies, *learned compressors* [3] aim to directly fit

Co-primary Authors: Qiyu Liu, Yuxin Luo, and Mengke Cui. Correspondence to Qiyu Liu (Email: qyliu.cs@gmail.com).

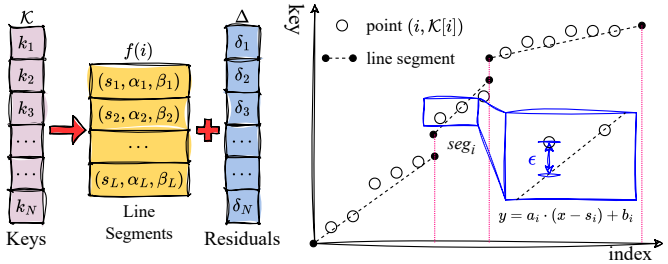


Fig. 1: Illustration of (left) the learned sorted list compressor and (right) an error-bounded piecewise linear model.

a projection function $f : \mathcal{I} \mapsto \mathcal{K}$ with a controllable error constraint ϵ, where $\mathcal{I} = \{1, 2, \cdots, |\mathcal{K}|\}$ is the index set. Then, a sorted list can be encoded by $\mathcal{K}^c = (f, \Delta)$, where Δ is an array of residuals between the model's predictions and the actual keys, i.e., $\Delta[i] = \mathcal{K}[i] - \lfloor f(i) \rfloor$. Since the mapping function f is error-bounded, i.e., $\Delta[i] \in [-\epsilon, +\epsilon]$, encoding each residual requires $\lceil \log_2(2\epsilon + 1) \rceil$ bits, and the total bits $\mathsf{size}(\mathcal{K}^c) = \mathsf{size}(f) + N \cdot \lceil \log_2(2\epsilon + 1) \rceil$, where $\mathsf{size}(f)$ is the bits required to encode a learned model f.

Intuitively, learning f is equivalent to learning the inverse cumulative distribution function (ICDF, a.k.a. quantile function) of $\mathcal{K}$. To balance the model's expressivity with inference efficiency, existing learned compressors often employ simple models like piecewise linear functions to approximate this mapping [3]. An error-bounded piecewise linear approximation (ϵ-PLA) is illustrated in Figure 1 and defined as follows.

Definition 2 (ϵ-PLA)**.** *Given a set of points in Cartesian space $\{(i, \mathcal{K}[i])\}_{i=1,\cdots,N} \subseteq \mathcal{I} \times \mathcal{K}$, an ϵ-PLA is defined as a piecewise linear function of L line segments,*

$$f(i) = \begin{cases} \alpha_1 \cdot (i - s_1) + \beta_1 & \text{if } s_1 \le i < s_2 \\ \alpha_2 \cdot (i - s_2) + \beta_2 & \text{if } s_2 \le i < s_3 \\ \cdots & \cdots \\ \alpha_L \cdot (i - s_L) + \beta_L & \text{if } s_L \le i \end{cases} \tag{1}$$

such that $|\mathcal{K}[i] - \lfloor f(i) \rfloor| \le \epsilon$ holds for $\forall i = 1, \cdots, N$. Each segment in f is a tuple (s_j, α_j, β_j), where s_j is the starting index, α_j is the slope, and β_j is the intercept.

In practice, the optimal online ϵ-PLA fitting algorithm [4] is adopted to minimize L, the number of required line segments. For `uint64` keys and `float32` slopes/intercepts, encoding

each segment requires 16 bytes, meaning that $\mathsf{size}(f) = 128L$ bits. Similar to conventional compressors, learned compressors need to tune hyper-parameters (i.e., ϵ in PLA) to achieve optimal performance. Intuitively, the value of ϵ can be neither too large nor too small: a large ϵ reduces the required segment count but increases the bits per residual; conversely, a small ϵ saves bits per residual at the cost of introducing more segments to meet the error constraint.

In this work, we showcase BitTuner, a fully data-driven framework that efficiently and effectively configures the error parameter ϵ for learned compressors based on ϵ-PLA. The key technical features of BitTuner are listed as follows.

❶ **Calibrated Cost Model**. We first establish an accurate space cost model, supported by rigorous theoretical analysis and extensive benchmark results.

❷ **Global Configuration**. For keys satisfying i.i.d. assumptions, BitTuner computes the *optimal* error parameter ϵ by minimizing the cost model developed in ❶, which turns out to be a *closed-form* solution.

❸ **Partition-based Configuration**. For arbitrarily distributed keys, a data partitioner is first invoked to split keys into consecutive and disjoint chunks that capture changes in distribution characteristics. BitTuner then applies the same configuration strategy from ❷ to each chunk, achieving an *adaptive* optimal configuration.

We propose to demonstrate BitTuner in two real-world list compression scenarios: vectorDB compression and inverted index compression. From our results, by adopting BitTuner, we can reduce the storage of the quantization codebook [5] constructed on a billion-scale vector set from 3.8 GiB to **0.88 GiB** ($\downarrow$ **425**%) without significant loss in efficiency. In addition to the provided scenarios, during the demonstration, participants can upload custom datasets (i.e., sorted keys) to BitTuner. Then, BitTuner will automatically analyze the data distribution and visually showcase how different choices of error parameters influence the compression ratio in an interactive manner.

II. Learned Compressor Cost Model

We first establish a space cost model based on theoretical analysis. Given an error parameter ϵ, previous studies [6] derive the expected segment count required for a PLA model to satisfy the error constraint.

Theorem 1 (Expected Segment Count [6])**.** *Given a set of N sorted keys $\{k_1, k_2, \cdots, k_N\}$, define the "gap" as $g_i = k_i - k_{i-1}$ for $i \geq 2$. Suppose g_i are i.i.d. random variables following an unknown distribution with mean μ and variance σ^2. For an error parameter satisfying $\epsilon \gg \sigma$, the expected segment count of the optimal ϵ-PLA is given by*

$$\mathbf{E}[L] = O\left(\frac{N \cdot \sigma^2}{\epsilon^2}\right). \tag{2}$$

According to Theorem 1, we reasonably model the segment count L as $L(\epsilon) = CN\sigma^2/\epsilon^2$, where C is a constant that can be fitted using some benchmark data (details are given in Section III). Then, the total number of bits required for a learned compressor based on ϵ-PLA is given by

$$B(\epsilon) = \underbrace{\frac{CN\sigma^2}{\epsilon^2} \cdot (K + 2F)}_{\text{segments}} + \underbrace{N \cdot \lceil \log_2(2\epsilon + 1) \rceil}_{\text{residual array}}, \tag{3}$$

where K and F are the number of bits to encode an integer (i.e., keys) and a float (i.e., slopes and intercepts). Based on the cost model derived in Eq. (3), when the "i.i.d." assumptions in Theorem 1 hold, the optimal error parameter ϵ can be derived by minimizing Eq. (3).

Theorem 2 (Optimal Global Error Parameter)**.** *Given a set of N sorted keys, the minimized space cost of a learned compressor based on ϵ-PLA (i.e., $B(\epsilon)$) is given by*

$$B^{\mathsf{opt}} \approx \frac{N}{2 \ln 2} + N \cdot \left\lceil 1 + \log_2 \epsilon^{\mathsf{opt}} \right\rceil, \tag{4}$$

with the corresponding ϵ^{opt} as

$$\epsilon^{\mathsf{opt}} = \sqrt{2 \ln 2 \cdot \sigma^2 \cdot C \cdot (K + 2F)}. \tag{5}$$

A detailed proof of Theorem 2 can be found in our technical report [7]. Theorem 2 implies that the optimal ϵ can be set by a simple *closed-form* solution, and the corresponding maximized compression ratio obtained by BitTuner is *fully provable*, i.e.,

$$\begin{aligned} \text{Compression Ratio} &\approx \frac{N \cdot K}{\frac{N}{2 \ln 2} + N \cdot \lceil 1 + \log_2 \epsilon^{\mathsf{opt}} \rceil} \\ &\approx \frac{K}{1.72 + \lceil \log_2 \epsilon^{\mathsf{opt}} \rceil}. \end{aligned} \tag{6}$$

Eq. (6) implies that by setting $\epsilon = \epsilon^{\mathsf{opt}}$, a learned compressor encodes each key with an average of $1.72 + \lceil \log_2 \epsilon^{\mathsf{opt}} \rceil$ bits.

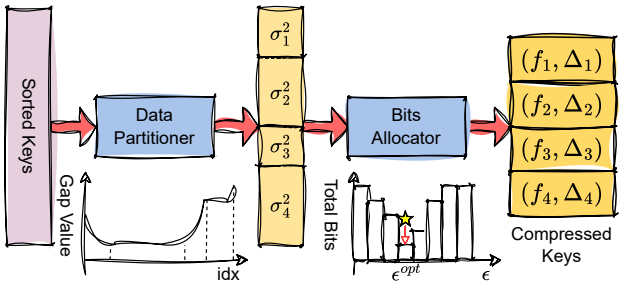


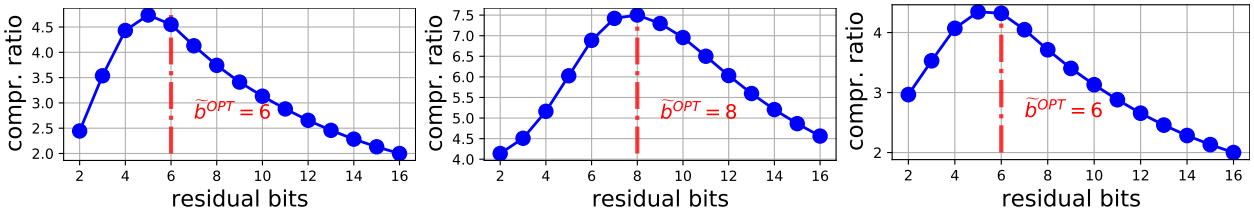
Fig. 2: Framework overview of BitTuner.

III. System Architecture

Based on the established space cost model of learned compressors, this section provides an overview of the architecture and main workflow of BitTuner, as illustrated in Figure 2.

A. Preparation

According to Eq. (3) and Theorem 2, the constants σ^2, K, F can be easily determined. For constant C, we fit a simple yet effective linear regression model on collected probe datasets. The procedure is as follows. We collect 100 sets of sorted keys with different sizes and gap distributions. Then, we physically construct ϵ-PLA models by varying ϵ across $\{2^1, 2^2, \cdots, 2^{10}\}$.



(a) SIFT (b) AUDIO (c) Deep1B

Fig. 3: Compression ratio w.r.t. different residual bits (b) to demonstrate the effectiveness of BitTuner's optimal parameter setting strategy. Note that $b = \lceil \log_2(2\epsilon + 1) \rceil$ and $\widetilde{b}^{\mathsf{OPT}}$ is the estimated residual bits using Eq. (10).

For each ϵ, we record the actual segment count L and fit it to the equation $L(\epsilon) = CN\sigma^2/\epsilon^2$. By extensive testing, we adopt $\widetilde{C} \approx 1.734$ as the estimated value for C, which achieves consistently high accuracy across different datasets.

B. Data Partitioning

Given a sorted list $\mathcal{K}$, Theorem 2 reveals that the only data-aware factor for determining ϵ^{opt} is σ^2, the *gap variance* within $\mathcal{K}$. Additionally, Theorem 2 relies on the "i.i.d." assumption on gap distribution. In practice, real-world datasets often exhibit shifts in gap variance, making the "i.i.d." assumption unrealistic. As a result, a globally configured ϵ can lead to sub-optimal compression efficacy.

To capture the change of gap distribution characteristics, we adopt PELT [8], an efficient change-point detection algorithm based on dynamic programming, on a random sample[1] over the full key set $\mathcal{K}$. PELT partitions the input keys $\mathcal{K}$ into disjoint chunks $\mathcal{P} = \{P_1, P_2, \cdots, P_m\}$, where $\cup_{i=1,\cdots,m} P_i = \mathcal{K}$ and $P_i \cap P_j = \phi$ for $\forall i \neq j$, such that the distribution characteristics (e.g., mean and variance) significantly differ for consecutive chunks P_i and P_{i+1}. During the partition phase, the gap variance σ_i^2 within chunk P_i is estimated for the subsequent residual bits allocation.

C. Residual Bits Allocation

We then construct learned compressors for partitioned key chunks $P_1, \cdots, P_m$. For each chunk P_i, we apply Theorem 2 to derive the optimal error constraint $\epsilon_i^{\mathsf{opt}}$. Then, the compressed representation for sorted list $\mathcal{K}$ is $\{(f_i, \Delta_i) \mid i = 1, \cdots, m\}$ where f_i is an $\epsilon_i^{\mathsf{opt}}$-PLA trained on P_i and Δ_i is the corresponding residual array. In practice, the residual array Δ_i is implemented as a compact bits array, and the number of bits to be allocated is $N \cdot \lceil \log_2(2\epsilon_i^{\mathsf{opt}} + 1) \rceil$.

Figure 3 illustrates the actual compression ratio on one selected chunk w.r.t. different settings of bits per residual on three real datasets. From the results, the estimated residual bits are close to their corresponding observed optimal values, revealing that, ❶ our established cost model can well depict the actual space overhead of learned compressors based on PLA, and ❷ the optimal parameter configuration strategy (i.e., Theorem 2) is theoretically correct.

[1]The decision to apply PELT on a sample of the key set $\mathcal{K}$, rather than the entire set, is based on efficiency consideration. While PELT is a linear-time algorithm, running it on billion-scale keys remains computationally expensive. In practice, the sample ratio is set to 10%, which is an empirical value to balance the estimation accuracy and efficiency loss.

TABLE I: Statistics of pre-loaded datasets in BitTuner.

Type	Dataset	Size	Gap Variance
PQ codewords	SIFT1B	1 Billion	11386.35
PQ codewords	Deep1B	1 Billion	4948.87
PQ codewords	AUDIO	438 Million	2763.58
docIDs	CCNews	42 Million	211.67
docIDs	Clueweb	35 Million	762.07
synthetic	Uniform	1 Billion	Varying
synthetic	Normal	1 Billion	Varying

IV. Demonstration Scenarios

We propose to demonstrate two real data compression applications using BitTuner: vectorDB codebook compression and inverted list compression. A screenshot of BitTuner's user interface is provided in Figure 4. Both the system and datasets are publicly available at [7].

Dataset Description. The statistics of datasets pre-loaded by BitTuner are summarized in Table I. We categorize these datasets into three types. (A) *Vector Codewords*. We adopt product quantization (PQ), a widely used technique to reduce the size of vectorDB [5], to encode high dimensional dense vectors into integer codewords. We adopt two billion-scale codebooks, SIFT1B and Deep1B, generated with a PQ4×8 quantizer using the `faiss` library [9]. (B) *Document IDs*. Another integer compression application is inverted index compression in information retrieval. We construct inverted indexes from two large corpora, CCNews and Clueweb [2], and pick the *largest* inverted list (i.e., document IDs) among all terms as the sorted keys. (C) *Synthetic*. We also provide two synthetic key sets sampled from uniform and normal distributions. Beyond the above pre-loaded datasets, users can upload their custom key sets via BitTuner's interface.

Scenario ❶: Data Exploration. The first scenario will allow users to interactively explore the raw keys, using either pre-loaded datasets or custom datasets. As shown in the left part of Figure 4, for a sorted list $\mathcal{K}$, BitTuner plots the relationship between key $\mathcal{K}[i]$ and index i, i.e., the mapping to be learned later. During data loading, BitTuner also computes the gaps of input keys and visualizes the gap distribution.

Scenario ❷: Keys Partitioning. By clicking the "Partition" button, BitTuner will sample the key gaps and apply the distri-

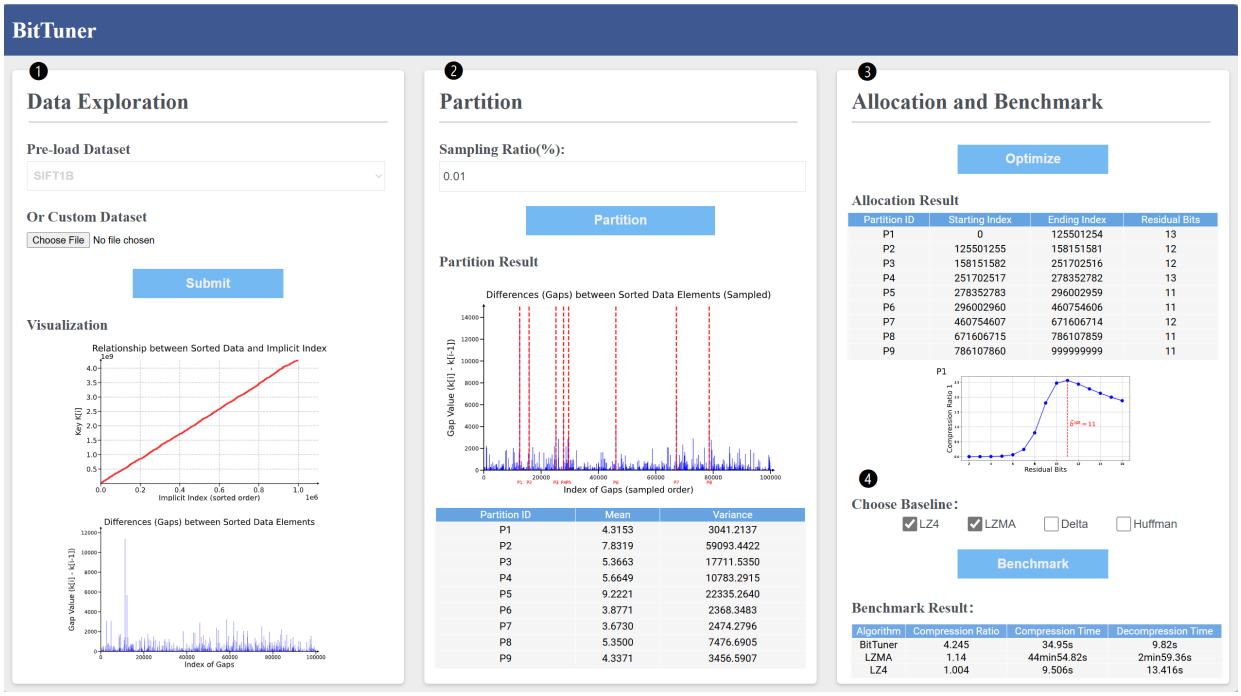


Fig. 4: Interface of BitTuner: ❶ upload and explore the raw key set; ❷ partition keys and compute necessary statistics; ❸ estimate error parameters and allocate error bits; ❹ construct learned compressors and visualize benchmark results.

bution change-point detection algorithm PELT [8] to partition raw keys into consecutive and disjoint chunks. The partitioning results are displayed in the middle of Figure 4, with red-dashed lines indicating partition boundaries. A summary of the gap distribution characteristics (e.g., mean and variance) for each chunk is shown in the right panel.

Scenario ❸: Residual Bits Allocation. In this scenario, users can click the "Optimize" button to automatically estimate the required residual bits for each chunk based on the space cost model (Section II). Once the estimation is complete, a result table will be displayed in the top-right corner of Figure 4, showing the specific residual bits configured by BitTuner for each partitioned key chunk. By further clicking on a chunk ID, a plot depicting the relationship between the *estimated* compression ratio and various residual bits settings is provided for users, allowing for a visual inspection of parameter tuning effects.

Scenario ❹: Benchmark. By clicking the "Benchmark" button, BitTuner will physically construct learned compressors for each partitioned key chunk. A micro-benchmark is then performed to evaluate the actual compression ratio, compression throughput, and decompression throughput. Through the "Baseline" checkboxes, users can choose to compare against different baseline methods, including generic compressors like LZ4 and LZMA, as well as entropy encoding methods like Huffman encoding. The benchmark results, once available, will appear in the bottom-right corner of Figure 4.

Performance Takeaways. From our benchmark results, on various integer compression tasks, our BitTuner is able to achieve the highest compression ratio with a minor efficiency loss. For example, BitTuner can reduce the raw size of SIFT's codebook from 3.8 GiB to 0.88 GiB ($\downarrow$ **425**%). More evaluation results are available in our technical report [7].

V. Conclusion

In this work, we demonstrate a novel framework BitTuner to address the challenges of learned compressor configuration. BitTuner is designed on the top of rigorous theoretical foundations, ensuring that the hyper-parameter tuned by BitTuner is *near optimal*. We showcase the effectiveness of BitTuner in two practical application scenarios: vectorDB compression and inverted index compression. Our demonstration provides insights into *why and how* hyper-parameters influence the overall compression efficacy of learned compressors.

References

[1] "RocksDB," https://rocksdb.org/, accessed: 2024-10-20.

[2] G. E. Pibiri and R. Venturini, "Techniques for inverted index compression," *ACM Computing Surveys (CSUR)*, vol. 53, no. 6, pp. 1–36, 2020.

[3] A. Boffa, P. Ferragina, and G. Vinciguerra, "A learned approach to design compressed rank/select data structures," *ACM Transactions on Algorithms (TALG)*, vol. 18, no. 3, pp. 1–28, 2022.

[4] J. O'Rourke, "An on-line algorithm for fitting straight lines between data ranges," *Commun. ACM*, vol. 24, no. 9, pp. 574–578, 1981.

[5] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[6] P. Ferragina, F. Lillo, and G. Vinciguerra, "Why are learned indexes so effective?" in *International Conference on Machine Learning*. PMLR, 2020, pp. 3123–3132.

[7] "BitTuner," https://github.com/qyliu-hkust/BitTuner, accessed: 2024-10-20.

[8] R. Killick, P. Fearnhead, and I. A. Eckley, "Optimal detection of changepoints with a linear computational cost," *Journal of the American Statistical Association*, vol. 107, no. 500, pp. 1590–1598, 2012.

[9] "Faiss," https://faiss.ai/, accessed: 2024-10-20.

APPENDIX

PROOF OF THEOREM 2

Proof. To simplify the final results, we assume that $2\epsilon \neq 2^k$, implying that $\log_2(2\epsilon+1) = 1+\log_2\epsilon$. Thus, Eq. (3) can be rewritten as

$$B(\epsilon) = \frac{CN\sigma^2}{\epsilon^2}\cdot(K+2F) + N\cdot\lceil 1+\log_2\epsilon\rceil. \tag{7}$$

We then define two auxiliary functions $B_\ell(\epsilon)$ and $B_h(\epsilon)$ as follows,

$$\begin{aligned} B_\ell(\epsilon) &= \frac{CN\sigma^2}{\epsilon^2}\cdot(K+2F) + N\cdot(1+\log_2\epsilon), \\ B_h(\epsilon) &= \frac{CN\sigma^2}{\epsilon^2}\cdot(K+2F) + N\cdot(2+\log_2\epsilon). \end{aligned} \tag{8}$$

The derivatives of $B_\ell\epsilon$ and $B_h(\epsilon)$ are

$$\frac{\partial B_\ell}{\partial\epsilon} = \frac{\partial B_h}{\partial\epsilon} = -\frac{2CN\sigma^2}{\epsilon^3}\cdot(K+2F) + \frac{N}{\epsilon\cdot\ln 2}. \tag{9}$$

By setting Eq. (9) to 0, ϵ^{opt} can be solved as follows,

$$\epsilon^{\mathsf{opt}} = \sqrt{2\ln 2\cdot\sigma^2\cdot C\cdot(K+2F)}. \tag{10}$$

Thus, the maximized $B_\ell(\epsilon)$ and $B_h(\epsilon)$ can be solved as follows,

$$B_\ell(\epsilon^{\mathsf{opt}}) = B_h(\epsilon^{\mathsf{opt}}) = \frac{N}{2\ln 2} + N\cdot\lceil 1+\log_2\epsilon^{\mathsf{opt}}\rceil \tag{11}$$

As $B_\ell(\epsilon) \le B(\epsilon) \le B_h(\epsilon)$, Eq. (11) is also the optima of $B(\epsilon)$ by taking $\epsilon = \epsilon^{\mathsf{opt}}$. Thus we complete the proof. □