BLISS: A Billion scale Index using Iterative Re-partitioning

Gaurav Gupta*
Rice University, Texas, USA

Tharun Medini
ThirdAI Corp., Texas, USA

Anshumali Shrivastava
Rice University & ThirdAI Corp., Texas, USA

Alexander J. Smola
Amazon Web Services, California, USA

ABSTRACT

Representation learning has transformed the problem of information retrieval into one of finding the approximate set of nearest neighbors in a high dimensional vector space. With limited hardware resources and time-critical queries, the retrieval engines face an inherent tension between latency, accuracy, scalability, compactness, and the ability to load balance in distributed settings. To improve the trade-off, we propose a new algorithm, called **BaLanced Index for Scalable Search (BLISS)**, a highly tunable indexing algorithm with enviably small index sizes, making it easy to scale to billions of vectors. It iteratively refines partitions of items by learning the relevant buckets directly from the query-item relevance data. To ensure that the buckets are balanced, BLISS uses the power-of-K choices strategy. We show that BLISS provides superior load balancing with high probability (and under very benign assumptions). Due to its design, BLISS can be employed for both near-neighbor retrieval (ANN problem) and extreme classification (XML problem). For the case of ANN, we train and index **4 datasets with billion vectors each**. We compare the recall, inference time, indexing time, and index size for BLISS with the two most popular and well-optimized libraries- Hierarchical Navigable Small World (HNSW) graph and Facebook's FAISS. BLISS requires **100**× lesser RAM than HNSW, making it fit in memory on commodity machines while taking a similar inference time as HNSW for the same recall. Against FAISS-IVF, BLISS achieves similar performance with **3-4**× less memory requirement. BLISS is both data and model parallel, making it ideal for distributed implementation for training and inference. For the case of XML, BLISS surpasses the best baselines' precision while being **5**× faster for inference on popular multi-label datasets with half a million classes.

CCS CONCEPTS

• **Information systems** → **Nearest-neighbor search**; *Top-k retrieval in databases*; • **Computing methodologies** → *Supervised learning by classification.*

*gaurav.gupta@rice.edu

KDD '22, August 14–18, 2022, Washington, DC, USA

ACM ISBN 978-1-4503-9385-0/22/08…$15.00
https://doi.org/10.1145/3534678.3539414

KEYWORDS

learning-to-index, billion-scale, search, classification, load-balance

ACM Reference Format:
Gaurav Gupta, Tharun Medini, Anshumali Shrivastava, and Alexander J. Smola. 2022. BLISS: A Billion scale Index using Iterative Re-partitioning. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22), August 14–18, 2022, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3534678.3539414

1 INTRODUCTION

Information Retrieval (IR) focuses on mapping a given query q to one or a few items out of an extensive set (millions to billions) of candidates. Beyond classical approaches that use techniques such as an inverted index [36], recent approaches have shifted towards retrieving nearest neighbors embedded in a dense embedding vector space as a first stage. Examples of this can be found in algorithms such as ColBERT for Question Answering [20], SLICE [14] (Bing Search), and DSSM [13] (Amazon Search).

Despite being one of the most venerable problems in IR, Approximate Near Neighbor (ANN) search continues to pose problems. In the past decade, learning-based solutions for ANN have shown significant promise in improving the entailing trade-offs between inference time and recall. This is primarily because ANN search and extreme classification (XML) share significant commonalities that we exploit in this work. In ANN search, the goal is to map a query q to a small set of k elements from a large inventory. It could be viewed as a classification problem with an extremely large set of possible labels. The only difference to extreme classification from this point of view is that, in the latter, the labels are given as part of the learning problem. In contrast, in ANN search, they are implicitly defined by the geometry of the space.

1.1 Prior Approaches

For several years, space partitioning methods like **Locality Sensitive Hashing (LSH)** [9] have been the backbone of ANN search. However, these methods are unmindful of the distribution of vectors, often leading to lop-sided partitions and long query times. The popular **K-Means** algorithm serves as the long-standing data-dependent clustering algorithm until the familiar issue of lop-sided partitions in conjunction with long indexing times made it infeasible to scale to millions of points. **Learning to Hash (LTH)** [22, 33] came to the rescue, wherein machine learning was first used to learn space partitions based on the data distribution. One of the notable works in this area of research is **Spectral Hashing** [34]. This algorithm learns to minimize the hamming distance between the hash codes of vectors that are close in metric space.

A highlighting feature of Spectral Hashing is that it avoids learning degenerate partitions by adding a combinatoric regularizer

to balance the load in each partition. However, optimizing such regularizers proves to be a challenge. Spectral Hashing remains a space partitioning with simple hyperplanes, which prohibits it from learning the complex manifolds that modern datasets exhibit.

In order to upgrade from pure space partitions to set partitions, where vectors in one part need not be geometrically proximal, we need to obtain a non-linear function f that maps the query q to a reasonable sized discrete data partition. Additionally, if these partitions (total B) are reasonably well balanced, f reduces the search space from L to L/B. It will ensure that we get a very relevant and sufficiently small partition, irrespective of any query distribution compared to index vectors' distribution.

Learning to Index: In recent years, due to the abundance of behavioral data like query to product purchases and query to ad clicks, several algorithms [25] have been proposed to learn the partitions directly from the associations between a query q and the items. This set of methods collectively opened up the space of **Learning to Index (LTI)** [6, 7, 21]. While LTI can learn complex indexing functions, it does not address the inherent load imbalance in the learned partition and creates a power-law distribution in partition/buckets' sizes. The fundamental cause of this imbalance is the power-law distribution of data. Post partitioning, the frequently queried (or similar) items tend to coalesce in large numbers into a few buckets, leaving the infrequent ones in the remaining buckets. This imbalance leads to higher inference times as we query heavy buckets more often than the lighter ones.

This warrants research in **load-balanced indexing** schemes. One of the early attempts in this regard is Balanced K-Means [23], which has a higher construction time than vanilla K-means, thus making it difficult to apply at scale. A recent noteworthy work blends pre-creation of an index and LTI is **NeuralLSH** [8]. NeuralLSH uses KaHIP [30], a balanced graph-partitioning algorithm to partition all the item vectors. It then maps a query to the relevant partition(s) *via* a neural network trained using soft labels (using the partition centers' distance with the query).

However, any balanced clustering approach to a power-law dataset faces challenges: 1) We are forced to split two relevant (close by) vectors into different clusters into dense regions. 2) In sparse regions, we are forced to accommodate two irrelevant (distanced) vectors in the same cluster. We call this the **'Curse of Clustering'**.

Over the past six years, small world graphs like **HNSW** [24] and approximate distance-based methods like Vector Quantization (VQ)[11] and Product Quantization (PQ) [16] have emerged as the most scalable solutions to industry-scale problems. Graph-based ANN methods like HNSW are efficient but have several limitations: 1) Due to the sequential nature of breadth-first-search traversal for indexing and querying, HNSW is not trivial to parallelize. 2) They are limited to standard distance metrics and do not generalize easily to any learning-based retrieval problem where the metric comes from the query-item relevance data. 3) They are designed for efficiency on CPUs but are less suitable for GPUs, distributed architectures, and secure computations (as quoted in [8]). 4) HNSW's humongous memory requirement is a well-known issue [10]. On the other hand, libraries that are fine-tuned on VQ and PQ (like the popular FAISS [18]) have much smaller indexes but are not very precise. Given these limitations, many commercial applications that deal with large-scale data are shifting again to learned data partitioning schemes (LTI) as they potentially form accurate and small indices.

The current semantic search pipeline still struggles with learning embeddings from query-item relevance. It is a pairwise training process [13] leading to a massive amount of training samples and extended training time. Additionally, negative sampling techniques have to be employed to prevent degenerate solutions, which only exacerbate the problem for large output spaces. Directly learning and indexing from query-item(label) relevance should be perfected as it can be faster. This problem is the same as XML. In this context, **Parabel** [29] is one of the primary algorithms that partitions the label space into roughly equal sets via a balanced 2-means label tree, where the label vectors are constructed using input instances. Subsequent improvements like eXtremeText [35] and Bonsai [19] relax the 2-way partitioning to higher orders of hierarchy. Another recent work, **SLICE** [14], builds an ANN graph on the label vectors obtained from a pre-trained network. It maps a query to the common embedding space during inference and performs a random walk on the ANN graph to obtain the relevant labels.

In the end, the existing approaches *decouple the partitioning step from the learning step*. Once a partition is created, it is fixed for the rest of the process while we map the query using either centroids, hashing, or a learned model. Like in NeuralLSH, the partitioning process could be an off-the-shelf algorithm (like KaHIP).

1.2 Our Contributions

In this paper, we make the following algorithmic and theoretical contributions with experimental corroboration.

- We propose a new learning-to-index algorithm - **BLISS** (**Ba**l**L**anced **I**ndex for **S**calable **S**earch). It takes the iterative training and re-partitioning approach, each time doing two things: 1) learning to map a query to a relevant bucket and 2) re-allocating points to buckets each time in a balanced way to achieve a superior index.
- We prove that BLISS achieves better recall than competing algorithms while maintaining load balance.
- We index 4 billion scale datasets from the BigANN benchmarks and a 100MM dataset and compare against the best baselines HNSW and FAISS.
- For a similar recall and a competitive query time, BLISS has a minimal index size (30x smaller than HNSW and 8x smaller than FAISS), making it feasible to work with low-spec machines.
- We propose a variant of our method called BLISS+, wherein we further bring down the in-memory index size to a few MBs (6000x smaller than HNSW and 1500x smaller than FAISS) with our data re-ordering approach.
- On two large Extreme Classification datasets, we outperform the best baselines Parabel and SLICE on the precision with $\approx 5\times$ faster inference.

BLISS is an illustration of the power of randomization. It breaks the barriers which are impossible with a deterministic assignment. It is possible to draw parallels between BLISS and clustering approaches like Balanced K-Means [23]. However, we break the aforementioned 'curse of clustering' by using three design choices - 1)

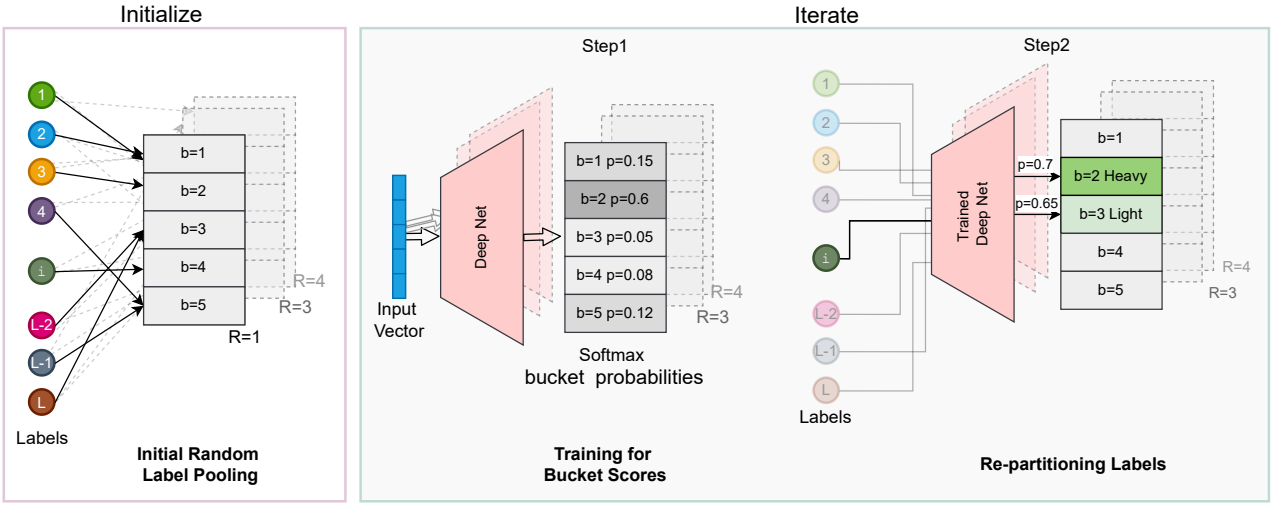

Figure 1: LEFT: Initialization step - the vectors are pooled randomly into B buckets using a 2-universal hash function. The above figure shows only five buckets (while we have a few thousand in practice). MIDDLE: Training - We train R fully-connected networks on N data points, where any bucket containing at-least one of the near neighbors is positive. RIGHT: After training for a few epochs, the vectors are re-assigned to the buckets. Each vector is fed into the R networks. We select the top-K buckets from the respective outputs and assign the vector to the least occupied bucket. K=2 in the figure yields 2^{nd} and 3^{rd} buckets as the top-K buckets. The light-green bucket is the lesser occupied one; hence, we assign the label to the 3^{rd} bucket). A larger K ensures perfect load balance, while it can also reduce the downstream precision and recall.

mapping a label to multiple clusters instead of a single assignment, 2) making a probabilistic interpretation of bucket assignments, 3) making multiple and independent index repetitions starting from random partitions. These choices allow us to have balanced partitions where the elements in each part bear a high degree of relevance (we achieve this by aggregating outcomes from multiple repetitions -Section 2.3).

2 BALANCED INDEX FOR SCALABLE SEARCH

BaLanced Index for Scalable Search (BLISS) begins with a random-pooling-based index initialization followed by an iterative process of alternating train and re-partition steps. We train R independent such indexes and use them for efficient item retrieval. Figures 1 and 2 illustrate our algorithm with a toy example of 5 buckets.

Notation: For a given dataset $\mathcal{D}$, we denote a query vector by x and the set of its near neighbors by $\bar{y}$. Let N be the total number of train vectors, d be the input vector dimension, and L be the total number of vectors/labels to index. R is the number of repetitions (independent indexes), and B is the number of partitions in each repetition. $f(.)$ is the learned deep-net model (we have R such models), and K is the load balance parameter.

2.1 Initialization

We initialize the partitions randomly. For this, we use R 2-universal hash functions $h_r : [L] \rightarrow [B]$, $r \in \{1, 2, .., R\}$. The hash function $h_r(.)$ uniformly maps the L vectors into B buckets. As the pooling is randomized, the buckets contain an equal number of labels in expectation. The first part of figure 1 shows the initialization.

Algorithm 1: BLISS Index Training

Input: data $(x_i, y_i) \in \mathbb{R}^{N \times d}$ and labels $l_i \in \{1..L\}$;
for $r = 1$ ***to*** R **do**
  Bucket$(l_i) = b$ #Initial random bucket assignment (Initialization);
  for $epoch = 1$ ***to*** T **do**
    Learn bucket scoring- $f_r(x_i) \in \mathbb{R}^B$ #Model training;
    for $l_i = 1$ ***to*** L **do**
      $\mathcal{B}_K$ = topK$(f_r(l_i))$ #Top K scoring buckets;
      b =argmin Load$(\mathcal{B}_K)$ #Least loaded bucket;
      Bucket$(l_i) = b$ #Re-assignment of the label;
    end
  end
end

2.2 Alternative Training and Re-partitioning:

Training: We train a fully connected neural network to learn the indexing function f_r of a given point x to B buckets where $B \ll L$. We have R independent partitions and thereby R independent neural networks $\{f_r | r \in [1, 2, .., R]\}$ (We use $R = 4$ in our experiments). We are effectively solving a classification problem using the cross-entropy loss: $\mathcal{L}(x, y, B) = -\sum_{b=1}^{B} y_b \log(p_b) + (1 - y_b) \log(1 - p_b)$, by providing B softmax scores (p_bs) against the respective ground-truths (y_bs). $y_b = 1$ if there is at-least one near neighbor present in the bucket b, else $y_b = 0$. The 2^{nd} part of figure 1 illustrates this step. In our experiments, for each dataset, we pre-generate the 100 exact near neighbors to a query point (using the corresponding distance metric) and treat them as labels.

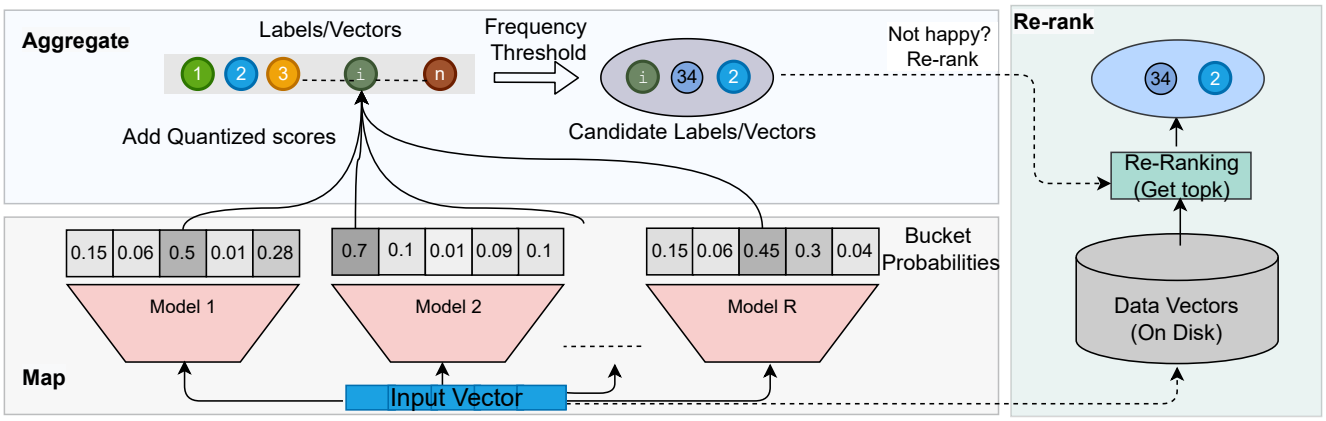

Figure 2: QUERY PROCESS: Here, the query vector is passed through R trained models, and each one gives a probability distribution over the corresponding buckets. We pick the top-m buckets with the highest score (space grey colored) and query an inverted index for candidates. The above figure shows $m = 1$ for illustration purposes (in our experiments, m tends to be around 10). We use a two-step ranking, first using a frequency-based filter (across the repetitions) and then computing the true distances on the retrieved candidate set.

Re-partitioning: This is the unique and vital step in BLISS as it creates a partition with more relevant labels pooled together in a bucket than the incumbent one. For every label $l \in \{1, 2..L\}$, the intent is to assign it to the bucket given by $argmax\ f(\bar{l})$, in each of the R repetitions. It is important to note that, for similar labels, the network will provide very similar $argmax$ values, leading to too many vectors being pooled into a few buckets.

Load Balancing: To overcome this imbalance, we select K top buckets for each label, choose the least occupied bucket among these, and assign the label to it. It ensures that the label fills the lighter bucket first to keep up with the load of the top buckets of similar labels. As we observe later in section 4, we will only need a small K (<10) to maintain a near-perfect load balance. For example, on GloVe100 dataset with around 1 million vectors, for $B = 4096$, we only need $K = 2$ buckets to achieve a load variance that is less than the LSH (Signed Random projection) bucket assignment.

We re-assign labels once every few training epochs (once every five epochs in our experiments). We alternate between the training and re-partitioning steps until the number of re-assignments converges to zero.

2.3 Inference/Query:

After training, we store the trained models and inverted indexes for all R repetitions. During the query process, a vector $q \in \mathcal{R}^d$ is passed through R trained nets independently in parallel, where each gives a B dimensional probability vector.

We select the top-m buckets (a hyper parameter- usually $5 - 20$ in our experiments) from each model, which gives a total of $m \times R$ buckets to probe. The target candidate set is a union of points/labels in these buckets. Additionally, we count each candidate's frequency of occurrence in the total $m \times R$ sets. A higher frequency of a candidate label signifies a higher relevance to the query point. Ultimately, we keep only the higher relevance candidates by filtering and rejecting the candidates below a certain frequency threshold from the pool. If the number of candidates exceeds the required, we re-rank them by true distance computation on the data residing on disk. We put them as a memmap array for quick fetching. Please refer to Figure 2 for an illustrative view of the query process.

Algorithm 2: BLISS Index Query

Input: Models $f_r(.)$, BLISS Index $\Pi_r, r \in \{1..R\}$;
Query point: $q \in Q$;
for $r = 1$ ***to*** R **do**
  $\mathcal{B}[1:m, r] = \text{topm}(f_r(q))$;
end
for $\mathbf{b} = \mathcal{B}[1, 1]$ ***to*** $\mathcal{B}[m, R]$ **do**
  $\phi = \phi \cup InvIndex(\mathbf{b})$;
end
$\phi = \text{ScoreThreshold}(\phi)$;
Candidate set = $\text{Reranking}(\phi)$;

Our procedure will ensure that, for every label vector, each of the R networks has a higher probability of selecting a relevant label than it can with learning on any predefined random partitioning (Section 3). Also, candidate set selection from R repetitions and frequency-based filtering exponentially decreases the variance of our true label estimates.

2.4 Tiny index (BLISS$^+$), using data reordering:

We observed that if we train a single model instead of R independent models, BLISS's index size can drastically reduce to a few MBs for a Billion scale dataset. To achieve this, we reorder the data vectors (residing on the disk and accessed through a *memmap*) according to the flattened lookup table (indices in buckets) for just one of the models. We can then discard the lookup table and only keep an array of bucket offsets in memory. It benefits us in two ways: 1) we do not store a billion int32 values for the index, and 2) it is faster to fetch vectors from the reordered data for re-ranking. We call this variant BLISS+ and empirically show the benefits in Table 2.

2.5 Modifying BLISS for Extreme Classification

For the case of extreme multi-label classification experiments, the labels do not have any pre-defined vector representations like the ANN datasets. Hence, we cannot plug in a label vector and get a probability distribution over the buckets. To address this missing link, we modify the probability distribution definition to the following.

$$P_l = \sum_{\forall i \ s.t \ l \in y_i}^{N} f(x_i), \quad f(.) \in R^B$$

Here, for every label l, we aggregate $f(x_i)$ for all x_is for which l is one of the true labels. It yields a B dimensional representative core for the label across all buckets. Furthermore, we re-assign it to the least occupied of the top-m buckets, as we do for ANN datasets.

3 ANALYSIS

In this section, we theoretically analyze BLISS from two main perspectives. First, we show that the predicted probability of buckets corresponding to the relevant labels increases after re-assigning the labels. Second, we show that re-assigning a label to the least occupied of the top-K buckets is the effective strategy to ensure load balance across the buckets.

As mentioned, we have L vectors being hashed to B buckets using a universal hash function. It randomly partitions the classes into B meta-classes. We estimate the top bucket probability $max\ Pr(b/x)$ for a input vector x, where $b \in \{1, 2..B\}$. Since each of the R repetitions is an instantiation of the same process, we only need an R-agnostic proof that re-assignment enhances the prediction probability of the most relevant bucket.

3.1 Increasing Relevant Labels' Probability through Re-assignment and Training

Theorem 1. *For a given dataset with $x \in \mathbb{R}^{N \times d}$, and its label l, the expected probability of the bucket containing l given the input x increases by a positive margin after re-partitioning, i.e.,*

$$\mathbb{E}\left(f^{'}[h^{'}(l)/x]\right) \geq \mathbb{E}\left(f[h(l)/x]\right)$$

where $f(.)$ is the scoring function given by the neural network and $h(.)$ is the incumbent hash function that maps labels to buckets. The increment in this probability results in an increment in the quality of the retrieved candidates during inference.

The main implication of this theorem is that the bucket containing the relevant label p_{l_1} gets a higher aggregated probability as it will have other true labels with higher probability. This increment will be higher in the beginning as the relevant labels get pooled together. Then the increment dies down to 0 as training progress, and the model converges to the ideal partition. Refer to the experimental analysis in the table 1. (Proof is in the Appendix.)

Please note that this increment in probability is manifested only after retraining on this new partitioning of the labels for a few epochs. After every re-partitioning step, BLISS naturally trains for a few epochs. The increased probability directly increases the recall during the evaluation. Given R repetitions, the estimated probability of l is given by $\hat{P}_l = \frac{1}{R}\sum_{r=1}^{R} P^r_{h(l)}$. With increasing R, the error in the correct label estimation also decreases exponentially.

Average predicted probability ($\times 10^{-5}$) of the true labels vs Epoch						
Epoch	5	10	15	20	25	30
No re-partitioning	5.95	6.54	5.89	4.47	3.44	3.58
BLISS	5.89	10.77	18.35	31.33	34.1	35.34

Table 1: The table shows the increase in the average predicted probability of the true labels' buckets with increasing epochs for two cases; one without re-partitioning (initialize the indexes randomly and fix them throughout the process) and the other with BLISS. We used the Amazon-670K dataset for this representative study[14].

3.2 Power of K choices

Theorem 2. *Consider the process in which, at each step, one of L labels is chosen independently and uniformly at random and inserted into B buckets. Each new label l inserted in the index can choose one among K possible destination buckets, which are the top-K buckets for this label l based on a similarity metric. We place l in the least occupied of these buckets.*

After sufficiently large number of insertions t, the most crowded bucket at that time contains fewer than $\frac{\log(\log(L)+f_1(K))}{\log(K)} + O(1) + f_2(K)$ labels with high probability, where f_1 and f_2 are monotonically decreasing functions of K.

Please refer to the appendix for the formal proof. It draws parallels from the famous power-of-2-choices framework [27].

When $K = B$, we choose the least occupied bin from all B buckets. Irrespective of any picking up strategy, the buckets will be the most balance here. With increasing $K > 1$, the loan balance increases; however, the chance of label reassignment to a high-affinity bin goes down, reducing the near-neighbor property of partitions. Higher K is suitable for load balancing, but it impacts final recall and precision. The value of K used in the experiment is empirically found optimal.

4 EXPERIMENTS

4.1 Near Neighbor Search

We segregate our ANN experiments into two sub-parts. The first part deals with two small million scale ANN datasets Glove100 and Sift128. This part is not intended to exemplify the scalability of BLISS but rather to have a comparison against NeuralLSH, which is in spirit the closest algorithm to BLISS. Also, the smaller datasets provide a fertile ground to perform several ablation studies on the choice of parameters like K and B.

In the second part, we show the massive scalability of BLISS by indexing 4 billion scale datasets and a 100 MM dataset. We compare against the two most prominent libraries, HNSW and FAISS, and show that at a similar recall and a competitive inference time, we require up to 30x smaller indexes in memory. Unfortunately, we could not scale NeuralLSH to the billion scale datasets as the underlying KaHIP partitioning is not conducive to the scale.

4.1.1 ***Evaluation on Small Scale datasets.*** In this section, we evaluate the quality of BLISS's learned data partitions by comparing the recall against candidate size (a smaller candidate size will lower the true distance computations). For this, we use two million-scale datasets from ANN benchmarks [1]- Glove100 [28] and Sift1M. Glove100 has a total of 1183514 points, each a 100 dimensional

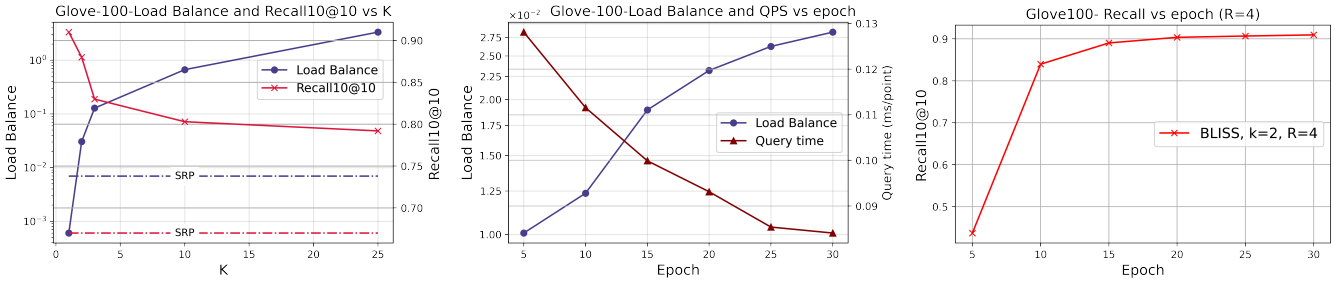

Figure 3: The left figure shows the Load Balance (1/Standard-Deviation) vs. K for Glove-100 with $B = 5000$. Larger K gets a better load balance while smaller K gets better precision and recall. The middle figure shows the Load Balance as training progresses for Glove-100. More training and re-partitioning get better load balance. Better load distribution leads to lower query time. The figure on the right shows the Epoch-wise Recall10@10 for Glove100 for a candidate size of 15K.

vector. Sift-1M has exactly 1 Million points, each a 128 dimensional vector. We train a fully connected network with one hidden layer of size 512 for both datasets. The input dimension is 100 and 128, respectively, for Glove and SIFT. The output layers for both datasets have the size $B = 4096$. We use $R = 4$ for both experiments. The number of top buckets (K) to choose while re-partitioning is set to 2. It is the smallest value with the slightest compromise in accuracy with a good load balance. The choice parameters for this experiment will be explained in the next section 4.1.2. We compare BLISS against

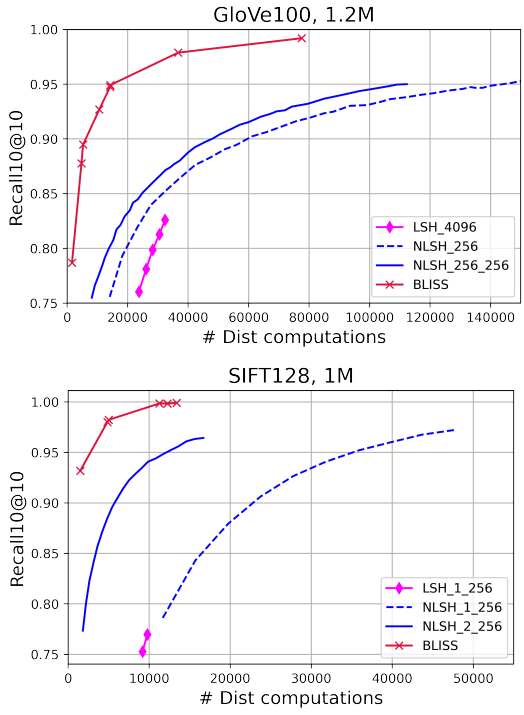

Figure 4: Comparison of BLISS with other partitioning methods on Glove100 and SIFT128 dataset: We compare Recall10@10 vs. the number of candidates (number of true distance computations). BLISS is noticeably better than the very recent learning-based index NeuralLSH.

NeuralLSH and Locality Sensitive Hashing, the former being a learned partition and the latter a non-learned one. We omitted all variants of K-Means as they were shown to be inferior in [8]. For NeuralLSH, a tag of _256 denotes 1 level with 256 buckets, while a tag of _256_256 denotes two levels, 256 buckets each (Figure 4). The number of parameters for the best version of NeuralLSH is $256 \times 128 \times (256 + 1) = 8.42$ million. In comparison, for the choice of our hyper-parameters, BLISS has a network size of $512 \times (4096 + 128) \times 4 = 8.65$ million. LSH uses the same number of buckets as BLISS does ($B = 4096$).

Metric: Our metric of interest is the recall of the top 10 neighbors for a particular candidate size. To be precise, both BLISS and the baselines only provide a set of candidate points within which we compute true distance computations to obtain the top 10 closest points. The intersection of this set with the true top-10 neighbors is Recall10@10.

Results : Figure 4 shows the comparison with NeuralLSH and other baselines. Here BLISS varies the number of top m buckets during retrieval. We notice that BLISS (red curve) comfortably surpasses the baselines for a given candidate size.

4.1.2 ***Ablation Study and choice of parameters.*** Figure 3(a) shows the Load Balance of the buckets after 20 epochs for Glove100 for various values of K. *Load Balance* is defined as the inverse of the standard deviation of bucket sizes. For each K, we start with a random partition of points (using a 2-universal hash function) and train for five epochs, after which we reassign the 1.2M vectors, as explained before. As K increases, each bucket tends to have nearly the same number of candidates. However, a larger K might compromise the relevance of buckets at query. Since the Load Balance at $K = 2$ is about five times more than Signed Random Projection (SRP, a popular LSH function), we chose $K = 2$ as an appropriate trade-off. It gives us the right mix of load balance, recall, and candidate size.

Figure 3(b) shows that the load gets more evenly distributed as we iteratively train and re-partition. The re-partitioning is done after every 5^{th} epoch. We also vindicate that an even distribution lead helps with the query time (as the number of candidates decreases. Figure 3 (c) shows the epoch-wise recall for Glove100 dataset. As we train more, we expect top buckets for a query to contain highly relevant vectors. The recall increases with epochs in the plot and converges around epoch 20. Hence, we can safely choose the 20 epochs trained model for our index.

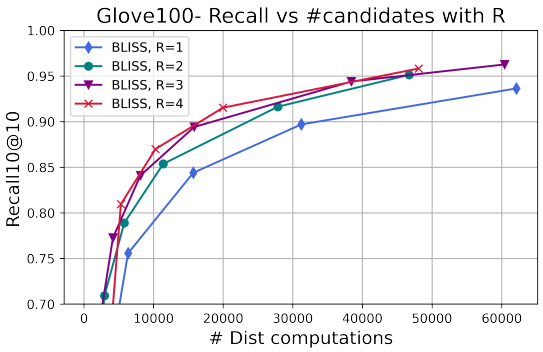

Figure 5: Recall10@10 for R varying from 1 to 4.

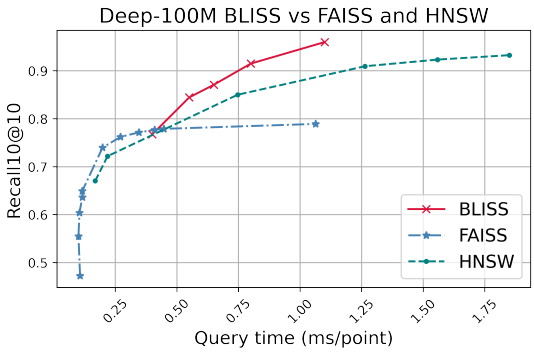

Figure 6: Recall10@10 vs. Query Time (with batch size 32). Comparison of BLISS with the baselines on CPU. On the 100M data, we see performance gains in the higher recall region. Additionally, we beat HNSW and FAISS on index size by a factor of 22× and around 2×, respectively.

Figure 5 shows the improvement in recall for different candidate sizes as we increase R from 1 to 4. For a given candidate size, we see diminishing gains in recall after $R = 4$.

4.1.3 ***Query time and Index size.*** Higher B leads to finer partitions, and hence it makes the candidate set smaller (of the order of $\frac{mN}{B}R\alpha$, where $\alpha < 1$ is compensating for common labels across R). This decreases the overhead for filtering the candidate set with frequency threshold and the eventual true distance computations. However, larger B makes the neural networks bigger and harder to train and infer. Including the top-m sorting, the time complexity of the forward pass on the neural network during inference is $(B+d)h_l + B\log m$, where h_l is the hidden layer size. The total cost exhibits a time complexity that looks like

$$Cost_{Inf} = O\left((B+d)h_l + B\log m + \frac{mN}{B}R\alpha\right) \quad (1)$$

This minimizes at $B = O(\sqrt{N})$, with respect to the number of labels to index. Going by this principle, we choose B to be the nearest power of 2 to $\sqrt{N}$ (to be system-friendly). This number turns out to be 4096 for the 1 million datasets, 16384 for the 100M dataset and 65536 for the 1 billion datasets.

The total size of index is $R(B+d)h_l + (N+B)R$ (model size + lookup size). With $B = O(\sqrt{N})$ and h_l being a constant factor of d, the index size grows as $O(N + d\sqrt{N} + d^2)$. For dimension (or effective dimension) $d << N$, we can safely say that the BLISS Index size $=O(N)$.

4.2 Large scale ANN Experiments

In this section, we stress test the scalability of BLISS by measuring the indexing and retrieval performance on Billion scale datasets [2] [17] [26] [37]. The details of the datasets are given in Table 3. As mentioned earlier, the prior baseline NeuralLSH's underlying KaHIP partitioning could not scale to the datasets of concern. Hence, we compare against the two most optimized libraries for industry scale ANN search, HNSW and, FAISS.

Like with the smaller datasets, we train $R = 4$ independent feed-forward networks with one hidden layer of size 512 for all datasets. The input dimensions are specific to each dataset. The output dimension is the number of buckets $B = 65536$ for the 1 billion datasets and is 16384 for the 100 million datasets, as justified in equation 1. We train for a total of 20 epochs and reassign the vectors to the buckets once every 5 epoch. We do not use the entire set of vectors to train. Instead, we take a 1% uniform sample of the data and obtain the ground-truth near-neighbors for this sample. For example, for a dataset with 1 billion vectors, we compute the pairwise ground truth distances for just the 10 million samples and index only them for training. Post-training, we index the entire set of 1 billion vectors in under 1 hr. The number of parameters for all 4 models in BLISS is approx. 134 million Additionally, BLISS stores 4 billion integers. This gives a cumulative index size of 15 GB.

For all the experiments (indexing and inference) on BLISS and baselines, we use a server equipped with AMD EPYC 7742 64-Core Processor and system memory of 1.5 TB. The RAM helped accommodate bulky HNSW indices; however, BLISS$^+$ and BLISS index takes just a few MBs to a few GBs, enough to run on a standard desktop.

For inference, we only use 32 threads for all experiments, and we do not use any GPUs. Also, we measure the throughput (queries-per-sec, QPS) and Recall10@10, keeping the number of queries in a single batch capped at 32. In a real scenario of a large-scale commercial search system, 32 is a safe batch size that can meet the stringent latency requirements.

Baselines: We use FAISS's IVFFLAT index type for a fair comparison against a partition-based approach. The number of clusters we use for FAISS is the same as B, i.e., 65536 for the billion scale datasets and 16384 for the 100 million dataset. For HNSW, we use the standard settings of degree= 8 and the ef-construction parameter= 100.

Results: Figures 6 compare BLISS against FAISS [18] and HNSW on Deep100M dataset. On this, we use 16384 buckets for BLISS and the same number of clusters for FAISS-IVF. In the high recall region, we see that BLISS gives better recall for the same inference speed. BLISS uses around 0.5 GB for the model and 1.6 GB for the indices in RAM compared to 44 GB for HNSW and 5.4 GB for FAISS.

During inference, the recall vs. time trade-off is governed by m (the number of top-scoring buckets we probe among the B). The red curve in figure 6 shows the recall vs query time with m ranging from 5 to 20. We pick only those candidate vectors from the top m buckets which appear in at least 2 of the 4 repetitions.

The table 2 shows the Query Per Second (QPS), Recall10@10, Index size, and the construction time for the four datasets with

	QPS				Recall10@10				Index size (construction time)			
Dataset	BLISS	BLISS$^+$	HNSW	FAISS	BLISS	BLISS$^+$	HNSW	FAISS	BLISS	BLISS$^+$	HNSW	FAISS
Deep1B	249	**400**	277	222	**0.9183**	0.81	0.8825	**0.919**	**15.5GB**	**137MB**	437GB	97GB
	384	**1724**	476	**769**	**0.8828**	0.6	0.846	0.7887	(1hr)	(1.2hrs)	(9hrs)	(>5days)
BIGANN	121	344	**909**	243	0.8443	0.658	0.8734	**0.8764**	**15.5GB**	**137MB**	557GB	127GB
	344	**909**	625	526	**0.792**	0.516	0.76	0.7495	(1hr)	(1.1hrs)	(10hrs)	(>5days)
Yandex TI	**110**	**384**	15	4	**0.568**	0.434	0.566	0.4919	**15.5GB**	**137MB**	826GB	194GB
	1631	**1470**	102	18	**0.4544**	**0.3053**	0.2629	0.272	(1hr)	(1.3hrs)	(16hrs)	(>5days)
MSSpaceV	220	270	322	**613**	0.8328	0.73	0.830	0.843	**15.5GB**	**137MB**	452GB	101GB
	416	**1333**	1075	1216	0.7879	0.6705	0.784	0.7852	(1hr)	(1.1hrs)	(9hrs)	(>5days)

Table 2: Query Per Sec, Recall10@10, Index size and construction time number on 4 Large scale data [2] [17] [26] [37] against the popular baselines FAISS[18] and HNSW [24]. FAISS took a long time for indexing, it couldn't finish the construction for Yandex and MSSpaceV in the given time constraint.

billion vectors. The two sets of BLISS Recall and QPS numbers represent top $m = 15$ and $m = 5$ for a frequency threshold of 2 out of R=4. FAISS and HNSW numbers are shown with different *nprobe* parameters.

BLISS performs quite well and beats HNSW and FAISS by a significant margin on in-memory index size while achieving very similar (and sometimes even better) performance on Recall10@10 and QPS (table 2). HNSW's current library has an index size requirement of around 2× the data, often becoming infeasible at a large scale.

Datasets	
Deep1B	1 Billion image data by Yandex[2], where the embeddings are generated from the last fully connected layer of a pre-trained GoogleLeNet [31], and compressed to 96 dimensions using PCA. Deep100M is its subset.
BIGANN	The embedding are SIFT descriptors of 128 dimensions extracted from 1B images [17] .
Yandex TI	Yandex's 1 Billion Text to image data[37]. It is best for cross-model retrieval tasks as it contains both text and image modalities. The Billion images' embeddings are produced by the Se-ResNext-101[12] model. The queries are generated from the text data by the DSSM[13] model. The embeddings share the representation space, although the distribution of queries and data is different.
MSSpaceV	1 Billion documents data, released by Microsoft Bing[26]. The embeddings are generated by Microsoft SpaceV Superior model[3].

Table 3: Details of the Billion scale datasets used for indexing.

The **BLISS$^+$** index further brings down the index size to a staggeringly low number. Given that the data vectors are reordered according to the BLISS partitions, the index boils down to an array of buckets' size offsets and the learned model. The data resides on the disk in a memmap format for all the re-ranking processes. Using just one repetition and ease of data vector reads, BLISS$^+$ has significant gains in QPS at some loss in the recall.

Additionally, we note that BLISS and BLISS$^+$ are very good with out-of-distribution queries in Yandex Text-to-Image data experiments. We have up to 10x better QPS than HNSW for the same recall. It proves that BLISS cal learns complex multi-modal distance metrics that HNSW or the likewise libraries do not. In this dataset, the input is a text embedding, and the indexed vectors are image embeddings.

4.3 Multi-label Classification

We use the dense versions of Wiki-500K and Amazon-670K [14] datasets available on the Extreme classification repository [4]. Both the datasets have 512 dimensional input vectors. Wiki-500K has 501070 classes with 1646302 train points and Amazon-670K has 670091 classes with 490449 train points.

We train a fully connected network with an input dimension of 512, a hidden layer of size 1024 nodes, and an output of $B = 20000$. The training was done for 20 epochs, and labels are re-assigned every five. The baselines of interest are Parabel [29], SLICE [14], AnnexML [32], Pfast XML [15] and SLEEC [5]. Here we used the best configuration of parameters used in the experiments of each paper. The evaluation metric we use is precision at 1,3,5 (P@1, P@3, P@5) and query time. It is defined as: Precision@$k = \frac{1}{k} \sum_{l \in rank_k(\hat{y})} y_l$.

As no label vectors are provided, we use the corresponding inputs for each label to get a proxy distribution over the buckets (as explained in section 2). Here we pay an additional re-partitioning cost of $O(L)$ once every five epochs. From the table 4, we observe that BLISS Index gives the best precision and runtime, beating all baselines for the Wiki-500K dataset. On the Amazon-670K, it is faster and more precise on the P@5 metric than the baselines in comparison.

5 CONCLUSION

We theoretically and experimentally corroborate that BLISS's iterative learning scheme with the power of K choices achieves data partitions that are better load balanced as well as conformal to the similarity metrics of any dataset. BLISS is scalable and system-friendly near neighbor retrieval algorithm that can learn and index with any similarity oracle, be it a distance metric (unsupervised setting) or given query-item relevance (supervised setting). Serving both large-scale ANN and XML problems, BLISS is a perfect tool for enterprise search. Additionally, BLISS can massively scale up the retrieval augmented language models, which are highly knowledge-intensive. We show the real-time performance on Billion scale datasets, with a minimal index size (few 100 MBs for BLISS$^+$) in memory.

6 ACKNOWLEDGEMENTS

This work was supported by: the National Science Foundation IIS-1652131, Office of Naval Research DURIP, Office of Naval Research BRC, and gift grants from Intel, Adobe, and Amazon research.

	Wiki-500K				Amazon-670K			
Method	P@1	P@3	P@5	QT	P@1	P@3	P@5	QT(ms)
BLISS (10 buckets)	**60.77**	**46.09**	**43.49**	**0.56**	35.56	32.68	**31.02**	**1.08**
BLISS (5 buckets)	**60.69**	**45.78**	**43.15**	**0.47**	35.13	32.20	**30.58**	**0.76**
SLICE	59.89	39.89	30.12	1.37	**37.77**	**33.76**	30.7	3.49
Parabel	59.34	39.05	29.35	2.94	33.93	30.38	27.49	2.85
AnnexML	56.81	36.78	27.45	-	26.36	22.94	20.59	-
Pfastre XML	55.00	36.14	27.38	6.36	28.51	26.06	24.17	19.36
SLEEC	30.86	20.77	15.23	-	18.77	16.5	14.97	-

Table 4: Precision @1, @3, @5 and Inference speeds- QT-Query Time (*ms/point*) for BLISS on Wiki-500K and Amazon-670K vs. popular Extreme Classification benchmarks.

REFERENCES

[1] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374. https://doi.org/10.1016/j.is.2019.02.006

[2] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[3] BigANN Benchmark. 2021. Billion-Scale Approximate Nearest Neighbor Search Challenge: NeurIPS'21 competition track. https://big-ann-benchmarks.com/index.html#call

[4] K. Bhatia, K. Dahiya, H. Jain, A. Mittal, Y. Prabhu, and M. Varma. 2016. The extreme classification repository: Multi-label datasets and code. http://manikvarma.org/downloads/XC/XMLRepository.html

[5] Kush Bhatia, Himanshu Jain, Purushottam Kar, Manik Varma, and Prateek Jain. 2015. Sparse local embeddings for extreme multi-label classification. In *Advances in neural information processing systems*. 730–738.

[6] Chih-Yi Chiu, Amorntip Prayoonwong, and Yin-Chih Liao. 2019. Learning to index for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 42, 8 (2019), 1942–1956.

[7] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries.. In *EDBT*. 407–410.

[8] Yihe Dong, Piotr Indyk, Ilya Razenshteyn, and Tal Wagner. 2019. Learning space partitions for nearest neighbor search. *arXiv preprint arXiv:1901.08544* (2019).

[9] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.

[10] Github. 2016. HNSW memory footprint 104. https://github.com/nmslib/nmslib/issues/104

[11] R.M. Gray and D.L. Neuhoff. 1998. Quantization. *IEEE Transactions on Information Theory* 44, 6 (1998), 2325–2383. https://doi.org/10.1109/18.720541

[12] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7132–7141.

[13] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning Deep Structured Semantic Models for Web Search Using Clickthrough Data *(CIKM '13)*. Association for Computing Machinery, New York, NY, USA, 2333–2338. https://doi.org/10.1145/2505515.2505665

[14] Himanshu Jain, Venkatesh Balasubramanian, Bhanu Chunduri, and Manik Varma. 2019. Slice: Scalable Linear Extreme Classifiers trained on 100 Million Labels for Related Searches. In *WSDM '19, February 11–15, 2019, Melbourne, VIC, Australia*. ACM. Best Paper Award at WSDM '19.

[15] Himanshu Jain, Yashoteja Prabhu, and Manik Varma. 2016. Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 935–944.

[16] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[17] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 861–864.

[18] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* (2019).

[19] Sujay Khandagale, Han Xiao, and Rohit Babbar. 2020. Bonsai: diverse and shallow trees for extreme multi-label classification. *Machine Learning* 109, 11 (2020), 2099–2119.

[20] Omar Khattab and Matei Zaharia. 2020. *ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/3397271.3401075

[21] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[22] Shaishav Kumar and Raghavendra Udupa. 2011. Learning hash functions for cross-view similarity search. In *Twenty-second international joint conference on artificial intelligence*.

[23] Mikko I Malinen and Pasi Fränti. 2014. Balanced k-means for clustering. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Springer, 32–41.

[24] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[25] Tharun Kumar Reddy Medini, Qixuan Huang, Yiqiu Wang, Vijai Mohan, and Anshumali Shrivastava. 2019. Extreme Classification in Log Memory using Count-Min Sketch: A Case Study of Amazon Search with 50M Products. In *Advances in Neural Information Processing Systems 32*. 13265–13275.

[26] Microsoft. 2021. SpaceV1B. https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B

[27] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[28] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[29] Yashoteja Prabhu, Anil Kag, Shrutendra Harsola, Rahul Agrawal, and Manik Varma. 2018. Parabel: Partitioned label trees for extreme classification with application to dynamic search advertising. In *Proceedings of the 2018 World Wide Web Conference*. 993–1002.

[30] Peter Sanders and Christian Schulz. 2013. Think locally, act globally: Highly balanced graph partitioning. In *International Symposium on Experimental Algorithms*. Springer, 164–175.

[31] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper With Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[32] Yukihiro Tagami. 2017. Annexml: Approximate nearest neighbor search for extreme multi-label classification. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 455–464.

[33] Jingdong Wang, Ting Zhang, Nicu Sebe, Heng Tao Shen, et al. 2017. A survey on learning to hash. *IEEE transactions on pattern analysis and machine intelligence* 40, 4 (2017), 769–790.

[34] Yair Weiss, Antonio Torralba, and Rob Fergus. 2008. Spectral hashing. *Advances in neural information processing systems* 21 (2008).

[35] Marek Wydmuch, Kalina Jasinska, Mikhail Kuznetsov, Róbert Busa-Fekete, and Krzysztof Dembczynski. 2018. A no-regret generalization of hierarchical softmax to extreme multi-label classification. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc., 6355–6366. https://proceedings.neurips.cc/paper/2018/file/8b8388180314a337c9aa3c5aa8e2f37a-Paper.pdf

[36] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World wide web*. 401–410.

[37] Yandex. 2021. Text-to-Image-1B. https://research.yandex.com/datasets/bigannsB

7 APPENDIX

7.1 Theorem 1: Increasing Relevant Labels' Probability through Re-assignment and Training

Proof: Let x be an input vector whose label set is $\bar{y}$ and p_l denote the probability of l being a true label to x. Let the current partitioning be given by a mapping $h(l)$, where $h(l) \in \{1, 2, .., B\}$. Also, assume $l_1, l_2 \in \bar{y}$ and $l_3 \notin \bar{y}$. Given x, the probability assigned to a bucket that contains l_1, i.e., $h(l_1)$ is given by the summation of probability of label l_1 and probability of other labels in the same bucket, i.e.,

$$P_{x,h(l_1)} = p_{l_1} + \sum_{k \neq l_1} \mathbf{1}_{h(k)=h(l_1)} p_k$$

where $\mathbf{1}$ is the indicator function.

Now, let us reassign the labels as mentioned in section 2.2. Let the new partition be given by $h^{'}(.)$. If $h(l_1) \neq h(l_2)$ and $h(l_1) = h(l_3)$, we want the re-partitioning to reverse this adversarial scenario, i.e., we expect that $h^{'}(l_1) = h^{'}(l_2)$ *and* $h^{'}(l_1) \neq h^{'}(l_3)$. Let Z represent the event of l_3 being removed from l_1's bucket and l_2 being added to it.

$$P^{'}_{x,h^{'}(l_1)} = p_{l_1} + \sum_{k \neq l_1} \mathbf{1}_{h^{'}(k)=h^{'}(l_1)} p_k + \mathbf{1}_Z (p_{l_2} - p_{l_3})$$

In expectation, we get -

$$\mathbb{E}\left(P_{x,h^{'}(l_1)}\right) = \mathbb{E}\left(P_{x,h(l_1)}\right) + \mathbb{E}(Z)(p_{l_2} - p_{l_3})$$

By design, l_1 and l_2 are the highest scoring labels given x. Hence, $p_{l_1} = p_{l_2} > p_{l_3}$. Also $\mathbb{E}(Z) \geq 0$ as Z is a probabilistic event. Hence,

$$\mathbb{E}\left(P_{x,h^{'}(l_1)}\right) = \mathbb{E}\left(P_{x,h(l_1)}\right) + \delta$$

where $\delta \geq 0$.

The bucket containing the relevant label p_{l_1} gets higher aggregated affinity as it will have other true labels with higher probability. The term δ represents the gain in probability by swapping a negative label with the true label in this bucket. The δ dies down to 0 as training progress, and the model converges to the ideal partition.

7.2 Theorem 2: Power of K choices

Proof: Let the number of buckets with load $\geq i$ after L insertions (i.e., the end of the re-partitioning) be $\leq \beta_i$, i.e., $\#buckets_{\geq i}(L) \leq \beta_i$. Our goal is to find an upper-bound $\#buckets_{\geq i+1}(L)$ to find the maximum load of all buckets.

In the event of a collision, consider that each label stacks up the on the existing labels like a tower. The height of a label in that case is the number of labels below it. Let $\#labels_{\geq i}(t)$ represent the total number of labels that have height $\geq i$ after total t insertions. Note that $\#labels_{\geq i}(t)$ is always higher than the $\#buckets_{\geq i}(t)$, as each bin with $\geq i$ has atleast one label with height $\geq i$.

For a new label to land at height $\geq i+1$, all K buckets (that we pick) should have load of atleast i. With the assumption made in [27], where the K buckets are chosen randomly, the probability of choosing K buckets that have height $\geq i$ is at most $p_i = \left(\frac{\beta_i}{B}\right)^K$.

Since the number of buckets with $\frac{2L}{B}$ insertions can atmost be $\frac{B}{2}$ (as total insertions are fixed to L), we have $\beta_{\frac{2L}{B}} \leq \frac{B}{2}$. However, we select the top K buckets based on the maximum affinity scores for a given query vector. In this case, for any sufficiently large $K \geq K_0$, the probability p_i is atmost $p_i \leq \left(\frac{\beta_i}{B}\right)^K + \delta$, where δ is monotonically decreasing function of K, $\delta = f_{\downarrow}(K)$. Please note that the δ used here is just an abuse of notation and has no relation whatsoever with the δ in Theorem 1.

The t^{th} label with height $\geq i+1$ has probability atmost p_i. Number of labels that have height $\geq i+1$ is atmost Lp_i. For a fixed BLISS index parameters $L > B$. We can safely assume that $L = \frac{B}{c}$, where $c < 1$.

$$\beta_{i+1} = Lp_i = \frac{B}{c}\left(\left(\frac{\beta_i}{B}\right)^K + \delta\right)$$

Just like the case of random K selection, we can set $\beta_{\frac{2}{c}} \leq B/2 + \delta$. We now find an expression for $\beta_{\frac{2}{c}+1}$ using induction $\beta_{\frac{2}{c}+1} = \frac{B}{c}\left(\frac{1}{2^K} + \delta_1\right)$ and $\beta_{\frac{2}{c}+2} = \frac{B}{c}\left(\frac{1}{2^{K^2}c^K} + \delta_2\right)$. Here each δ_i is a positive real number, monotonically decreasing in K. The $\beta_{\frac{2}{c}+i}$ is given by

$$\beta_{\frac{2}{c}+i} = \frac{B}{c}\left(\frac{1}{2^{K^i}c^{K^{i-1}}} + \delta_i\right)$$

The probability of a bin with number of balls less than i^* is more than $(B-1)/B$ when $\beta_{i^*} > 1$. That happens when

$$\frac{B}{c}\left(\frac{1}{2^{K^i}c^{K^{i-1}}} + \delta_i\right) > 1$$

for $i^* = 2/c + i$, we have $i < \log_K\left(\log\left(\frac{L}{1-L\delta_i}\right)\right) - \log_K\left(1 + \frac{1}{K}\log c\right)$. This can be further simplified to-

$$i^* < \frac{\log\left(\log L + f_1(K)\right)}{\log K} + f_2(K) + 2L/B$$

Where $f_1(K) = -\log(1 - L\delta_i)$ and $f_2(K) = -\log_K(1 + \frac{1}{K}\log c)$. δ_i and hence $f_1(K)$ monotonically decreases with K. Additionally, using the fact that the derivative of $\frac{\log(1+\frac{\log c}{x})}{\log x}$ is positive for $x > 1$, we can conclude that $f_2(K)$ is also a decreasing function of K. Refer figure 7 below for L = 1M, B=4096 and $L\delta_i < 1$.

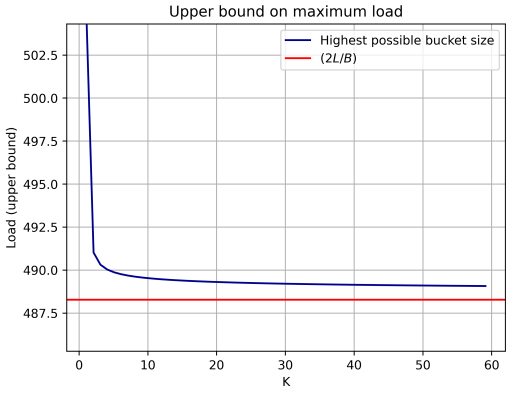

Figure 7: Upper bound on maximum load vs K

Hence, we can say that, with probability more than $1 - (1/B)$ maximum load is upper bounded by

$$\log_K\left(\log\left(\frac{L}{1-L\delta_i}\right)\right) - \log_K\left(1 + \frac{1}{K}\log\frac{B}{L}\right) + 2\frac{L}{B}$$