AIBox: CTR Prediction Model Training on a Single Node

Weijie Zhao[1], Jingyuan Zhang[1], Deping Xie[2], Yulei Qian[2], Ronglai Jia[2], Ping Li[1]
[1]Cognitive Computing Lab, Baidu Research USA
[2]Baidu Search Ads (Phoenix Nest), Baidu Inc.
1195 Bordeaux Dr, Sunnyvale, CA 94089, USA
No.10 Xibeiwang East Road, Beijing, 10085, China
10900 NE 8th St, Bellevue, WA 98004, USA
{weijiezhao,zhangjingyuan03,xiedeping01,qianyulei,jiaronglai,liping11}@baidu.com

ABSTRACT

As one of the major search engines in the world, Baidu's Sponsored Search has long adopted the use of deep neural network (DNN) models for Ads click-through rate (CTR) predictions, as early as in 2013. The input futures used by Baidu's online advertising system (a.k.a. "Phoenix Nest") are extremely high-dimensional (e.g., hundreds or even thousands of billions of features) and also extremely sparse. The size of the CTR models used by Baidu's production system can well exceed 10TB. This imposes tremendous challenges for training, updating, and using such models in production.

For Baidu's Ads system, it is obviously important to keep the model training process highly efficient so that engineers (and researchers) are able to quickly refine and test their new models or new features. Moreover, as billions of user ads click history entries are arriving every day, the models have to be re-trained rapidly because CTR prediction is an extremely time-sensitive task. Baidu's current CTR models are trained on MPI (Message Passing Interface) clusters, which require high fault tolerance and synchronization that incur expensive communication and computation costs. And, of course, the maintenance costs for clusters are also substantial.

This paper presents **AIBox**, a centralized system to train CTR models with tens-of-terabytes-scale parameters by employing solid-state drives (SSDs) and GPUs. Due to the memory limitation on GPUs, we carefully partition the CTR model into two parts: one is suitable for CPUs and another for GPUs. We further introduce a bi-level cache management system over SSDs to store the 10TB parameters while providing low-latency accesses. Extensive experiments on production data reveal the effectiveness of the new system. AIBox has comparable training performance with a large MPI cluster, while requiring only a small fraction of the cost for the cluster.

CCS CONCEPTS

• **Information systems → Online advertising**; • **Hardware → Analysis and design of emerging devices and systems**.

KEYWORDS

Sponsored Search; GPU Computing; SSD Cache Management

CIKM '19, November 3–7, 2019, Beijing, China

ACM ISBN 978-1-4503-6976-3/19/11…$15.00
https://doi.org/10.1145/3357384.3358045

ACM Reference Format:
Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR Prediction Model Training on a Single Node. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM '19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357384.3358045

1 INTRODUCTION

As one of the leading search engines in the world, Baidu's Sponsored Search system (a.k.a. "Phoenix Nest") [20] has long adopted deep neural network (DNN) models for ads click-through rate (CTR) predictions, as early as in 2013. CTR prediction [19, 22] plays a key role in determining the best ad spaces allocation as it directly influences user experience and ads profitability. Usually CTR prediction takes multiple resources as input, e.g., query-ad relevance, ad features and user portraits. It then estimates the probability that a user clicks on a given ad. Recently, deep learning has achieved great success in computer vision [29] and natural language processing [5]. Inspired by this, learning methods are proposed for the CTR prediction task [13, 15, 20, 61, 65, 66]. Compared with commonly used logistic regression [40], deep learning models can substantially improve the accuracy at a significant increase of training cost.

In the current production system at Baidu Search Ads, The training process of our model is both resource-intensive and time-consuming. The models are trained with a parameter server [24, 32] in an MPI (Message Passing Interface) [52] cluster with hundreds of CPU-only nodes. The main model used in production is of size exceeding 10TB and is stored/managed on a special hardware. The parameter server solution suffers from node faults and network failures in the hundred-scale nodes environment. What's worse, synchronizations in parameter server block the training computations and incur large network communication overhead, while asynchronous training frameworks have model convergence problems caused by the stale models on each worker node [48, 62].

There are fascinating opportunities and challenges to improve the production system of sponsored search, in many different directions. One area of active research is to improve the quality of "recalls" (of ads) before calling the CTR model. For example, Baidu has recently shared such a technical paper [20] to the community, which was built on top of fast near neighbor search algorithms [55, 64] and maximum inner product search techniques [51, 56].

In this paper, we present Baidu's another concurrent major effort for improving the online advertising system, that is, moving the CTR model training from MPI clusters to GPUs. While the use of GPUs for machine learning and scientific computing has been a common practice, using GPUs for training commercial CTR models

still imposes many substantial challenges. The most notable challenge is that the training data size is of PB (PeteByte) scale and the trained model is of size exceeding 10TB. The number of training examples can be as many as hundreds of billions and the number of features can be thousands of billions (we typically use 2^{64} as a convenient surrogate for the size of feature space.). The data fed to the model are also extremely sparse, with merely several hundred non-zero entries for each feature vector.

As a commercial sponsored search system, any model compression techniques should not compromise the prediction performance (revenue). Indeed, even a tiny (e.g., 0.1%) decrease of prediction accuracy would result in an unacceptable loss of revenue. In fact, the entire system has been highly optimized with little redundancy (for example, the parameters have been carefully quantized into integers). It appears that there is very little room for improvement.

Popular model compression techniques such as downsampling [8, 30, 46] and hashing [12, 27, 33, 34, 57] turn out to be less effective for training commercial CTR models with training data which are extremely high-dimensional (e.g., hundreds of billions of features) and extremely sparse (e.g., with only several hundred of non-zero entries in each feature). Common arguments seen in research papers, such as "reducing the training cost by half with only a 0.3% loss of accuracy", no longer work in this industry. On the other hand, training DNN CTR models is a daily routine work at Baidu. Engineers and data scientists have to experiment with many different models/features/strategies/parameters and have to train and re-train CTR models very frequently. The cost of hardware (such as MPI clusters) and energy consumption can be highly prohibitive.

To tackle these challenges, we present AIBox, a novel centralized system to train this huge machine learning model on a single node efficiently. AIBox employs emerging hardware, SSDs (solid-state drives) and GPUs, to store the large number of parameters and to accelerate the heavy neural network training computations. As a centralized system, AIBox directly eliminates those drawbacks caused by network communications in distributed systems. In addition, the single node AIBox has orders of magnitudes fewer hardware failures comparing with thousands of nodes in a large computing cluster [49]. Furthermore, synchronization costs in a single node are significantly reduced because only in-memory locks and GPU on-chip communications are required. There is no data transferring through network compared with distributed environments.

Nonetheless, there are two major challenges in the design of AIBox. The first challenge is to store the 10TB-scale model parameters on a single node. The memory price hikes when the capacity reaches more than 1 TB. It is not scalable when the model becomes larger in the future and is not practical in mass production for real-world applications. Due to the high cost, we cannot store the entire 10TB parameters in the main memory. The emerging Non-Volatile Memory express (NVMe) SSDs on PCIe buses have more than 50X lower latency than hard drives [60]. We leverage SSDs as a secondary storage to save the parameters. However, SSDs have two drawbacks. First, in terms of latency, SSDs are still two orders of magnitude slower than the main memory, resulting in slow access and update of parameters in the training process. The other downside of SSDs is that storage cells in SSDs only last several thousand write cycles. Thus, we have to maintain an effective in-memory cache to hide SSD latencies and reduce disk writings to SSDs.

The second challenge is to employ multiple GPUs on one single node to accelerate the training computation. Recently, the single-precision performance of Nvidia Tesla V100 with 32 GB High-Bandwidth Memory (HBM) [28] achieves 15.7 TFLOPS that is 47X faster than a top-end server CPU Node (Intel Xeon series) on deep learning inference [38]. This provides unique opportunities to design a multi-GPU computing node that has comparable computation performance to a cluster. However, current off-the-shelf GPUs do not have TB-scale HBM. We cannot keep the entire CTR prediction neural network in the GPU HBMs. In this work, we propose a novel method (Section 2) to partition the neural network into two parts. The first part is memory-intensive and is trained on CPU. The other part of the network is computation-intensive while having a limited number of input features. We train it on GPU. Training data and model parameters are transferred between the main memory and the HBM of multi-GPUs. However, communications between the main memory and the GPU HBM are limited by the PCIe bus bandwidth. The high GPU numerical computation performance is blocked when the communication bandwidth becomes the bottleneck. The emerging NVLink [21] and NVSwitch [44] techniques enable direct GPU-to-GPU communication without involving PCIe buses. We employ NVLink and design an in-HBM parameter server to reduce GPU data transfers.

In summary, the main contributions of this work are:

- We introduce AIBox, a single node accelerated by SSDs and GPUs, to train CTR prediction model with 10TB parameters. The single node design paradigm eliminates expensive network communication and synchronization costs of distributed systems. As far as we know, AIBox is the first centralized system designed for real-world machine learning applications at this large scale.
- We show a novel method that partitions the large CTR prediction model into two parts. After the partition, we are able to keep the memory-intensive training part on CPU and utilize memory-limited GPUs for the computation-intensive part to accelerate the training.
- We propose Sparse Table to reduce SSD I/O latency by storing model parameters on SSDs and leveraging memory as a fast cache. Moreover, we implement a 3-stage pipeline that overlaps the execution of the network, Sparse Table and CPU-GPU learning stages.
- We conduct extensive experiments to evaluate the proposed system by comparing it against the distributed cluster solution with 75 nodes on real-world CTR prediction data consisting of 10 PB examples. It shows the effectiveness of AIBox—AIBox has comparable training performance with the cluster solution, while it only requires less than 1/10 of the cost that we pay for the cluster.

The rest of this paper is organized as follows. Section 2 presents our CTR prediction neural network model. In Section 3, we introduce the high-level design of AIBox. We investigate the challenges for the large model storage on Sparse Table and propose solutions that employ a bi-level cache management system over SSDs in Section 4. Experimental results are shown in Section 5. Section 6 discusses the related work, and Section 7 concludes the paper.

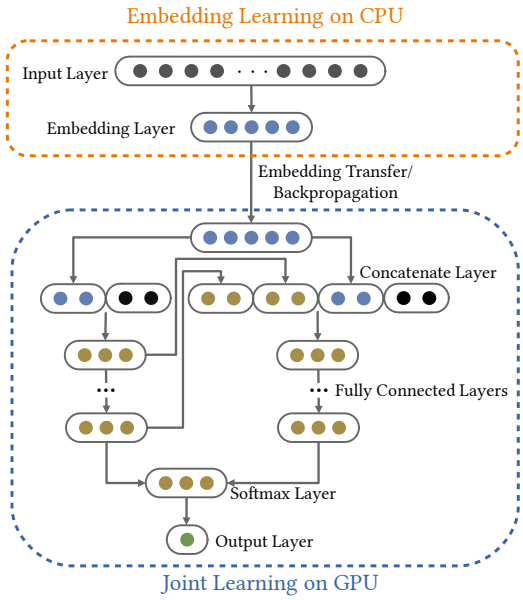


Figure 1: The overview of the designed CTR prediction neural network model. The nodes at the input layer of the embedding learning represent the high-dimensional sparse features. The nodes without incoming arrows at the concatenate layer of the joint learning are the dense personalized input features (in addition to the embeddings).

2 CTR PREDICTION NEURAL NETWORK

The industrial deep networks are designed and trained with massive-scale data examples to help predict the CTR of an advertisement accurately, quickly, and reliably. Features in Baidu's CTR prediction models are typically extremely sparse features (e.g., hundreds or even thousands of billions of features), with only a very small number (e.g., several hundreds) of non-zero values per vector. This huge DNN model has parameters of size exceeding 10TB, after storing only the nonzero parameters with careful quantization. Because of the limited HBM capacity of GPUs, it is clearly impractical to keep the 10TB parameters of the entire model in the HBM of GPUs.

In this paper, we present the two-module architecture for training huge DNN CTR models on CPUs+GPUs. The first module focuses on the embedding learning with high-dimensional & sparse features and the second module is for joint learning with dense features resulted from the first module. The embedding learning is processed on CPUs to help learn low dimensional dense embedding representations. Since the memory-intensive issue of the 10TB parameters makes it impossible to maintain the entire model in memory during training, we leverage SSDs to store model parameters. The parameters can be rapidly accessed from SSDs to CPUs. By transferring the learned embedding vectors from CPUs to GPUs, the computation-intensive joint learning module can make full use of the powerful GPUs for CTR prediction. In the joint learning module, several fully connected neural networks are modeled by taking the embeddings as inputs. The last layers of these neural networks are concatenated together for the final CTR prediction. Figure 1 shows the overview of the designed CTR neural network model. We introduce the details of the model in the following subsections.

For ease of discussion, in the rest of the paper, we will use 10^{12} (hundred billions) as the dimensionality of the extremely sparse features in CTR models. Readers should keep in mind that the actual number of features in production system could well exceed 10^{12}.

2.1 Embedding Learning on CPUs

The embedding learning module aims to map the high dimensional sparse vectors (e.g., 10^{12} dimensions) into low dimensional dense representations. As shown in Figure 1, the embedding learning module includes the input layer of high-dimensional sparse features and the output embedding layer. ReLU is used as the activation function. This module is mainly memory-intensive since the 10^{12} features result in 10TB-scale model parameters and it is infeasible to load all the parameters in the main memory. In order to learn embeddings, we store the 10TB parameters into SSDs. Due to the efficient access speed between SSDs and CPUs, we can easily load parameters from SSDs and learn embeddings on CPUs.

2.2 Joint Learning on GPUs

After we compute the embeddings on CPUs for high dimensional sparse features, we transfer the embeddings from CPUs to GPUs for the CTR prediction process. The input of the joint learning consists of the dense personalized features and the learned embeddings. The personalized features are usually drawn from a variety of sources, including the text of the ad creative, the user personalized behaviors, and various ad-related metadata. If we directly concatenate these features and feed them into neural networks, important information from the personalized features may not be fully explored, resulting in inaccurate CTR prediction results. Therefore, we design several deep neural networks and jointly learn meaningful representations for the final CTR prediction. As shown in Figure 1, the joint learning module contains several (two in the figure) deep neural networks. Each network concatenates the learned embeddings and one kind of personalized information together as the input layer. Then several fully connected layers are applied to help capture the interaction of features in an automatic manner. The last hidden layers of these networks are combined for the softmax layer and the output layer of the CTR prediction. We use the negative log-likelihood as the objective function:

$$L = -\frac{1}{|S|} \sum_{(\mathbf{x}, y) \in S} (y \log p(\mathbf{x}) + (1 - y) \log(1 - p(\mathbf{x}))) \qquad (1)$$

Here S is the training set and $|S|$ is the size of training examples. $\mathbf{x}$ represents the input features of our CTR prediction model and $y \in \{0, 1\}$ is the label indicating whether the ad is clicked or not by a user. $p(\mathbf{x})$ is the output after the softmax layer, denoting the predicted probability of the data example being clicked.

In order to effectively learn from the previous neural networks, the representations are extracted from the first and the last hidden layers and then concatenated with the input layer of the current neural network for joint learning. Specifically, the first hidden layer represents a low-level feature learning and extracts the most related information from the input layer. The last hidden layer shows a high-level feature learning and detects the most abstract but helpful

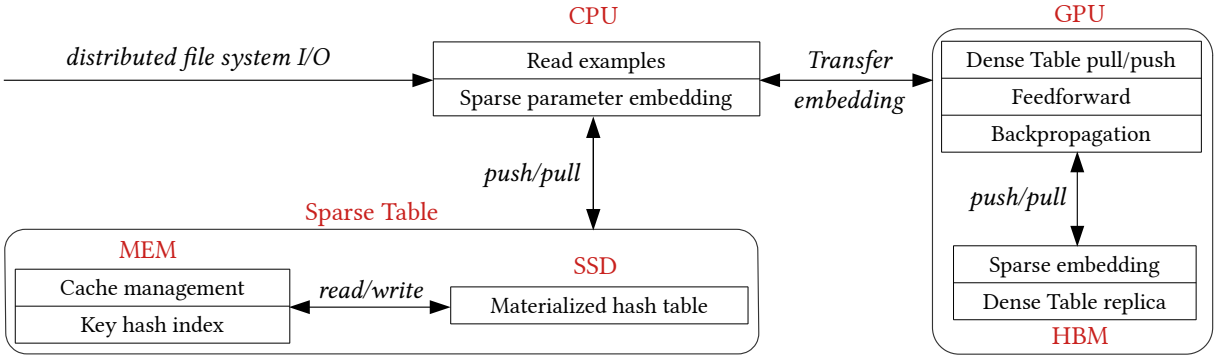


Figure 2: The architecture of AIBox.

information for the final CTR prediction. We incorporate the most meaningful low-level and the most powerful high-level information from previous networks for a more accurate CTR prediction result.

This module is mainly computation-intensive since there are multiple neural networks. We train this module on GPUs for speed-up. After one or several rounds of joint learning are done, backpropagation is used to first update the neural network parameters and transferred embeddings on GPUs. Then the updated embeddings are returned to update the model parameters on CPUs. In this way, the two modules work together for the CTR model training.

3 AIBOX SYSTEM OVERVIEW

In this section, we present the AIBox system overview and describe its main modules from a high-level view. Figure 2 depicts the architecture of AIBox. It contains three components, CPU module, Sparse Table module and GPU module.

CPU module for coordination and embedding learning: The CPU module coordinates the entire training workflow. First, it reads the training examples from a distributed file system (e.g., HDFS [7]) through high-speed network and constructs mini-batches for the following parallel processing. We call the mini-batches a pass. Then the CPU module computes the feature embedding of the mini-batches by interacting with the Sparse Table module to obtain the referenced parameters. After that, the embeddings are transferred to GPUs for the joint learning part of the neural network. Upon completion of the joint learning on GPUs, backpropagated gradients of the joint neural network inside the GPU module are returned to the CPU module. It updates parameters on the CPU side through the push operation of Sparse Table. In addition to the main workflow, the CPU module periodically saves the current training snapshots as checkpoints to the distributed file system for job failure recovery.

Sparse Table: The Sparse Table is a key-value storage system that stores values (e.g., model weights and click information) of 10^{12} discrete features on SSDs. These features consume more than 10TB space. A key hash index of the feature-to-file mapping locates in the memory for the key look-up. Although SSDs have much lower latency than hard drives, they still cannot catch-up with the nano-second level memory latency. To deal with this critical issue, we design an explicit in-memory cache strategy for SSD accesses to hide the latency. Meanwhile, the cache also acts as a buffer that avoids frequent writings to SSDs (Section 4).

GPU module for joint learning: The GPU module receives sparse embeddings transferred from the CPU module and stores them in the HBMs. Then the embeddings are fed to the dense joint learning network. The embedding transferring through PCI-E bus and the joint learning computation on GPU cores are overlapped. We create a separate CUDA stream [43] for data transferring and another CUDA stream for learning computations. GPU HBMs act as an on-chip parameter server. The parameters of the dense neural network are replicated across all GPUs and stored in the on-chip GPU HBM. For each pass of mini-batches, each GPU computes the new parameters according to its local copy. After that, all GPUs in the AIBox perform collective communications to synchronize parameters via NVLink high-speed interconnections.

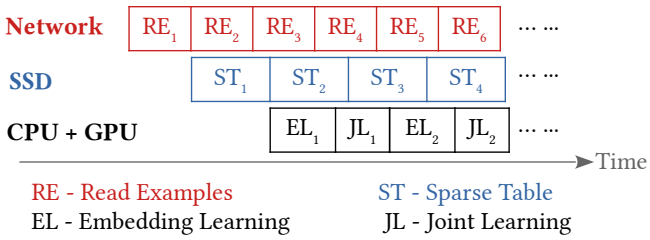


Figure 3: The 3-stage pipeline that overlaps executions on network, SSDs and neural network training (CPUs + GPUs).

3-stage pipeline: The training workflow mainly involves 3 time-consuming tasks: training example reading, sparse table operators and neural network training (embedding learning + joint learning). The 3 tasks correspond to independent hardware resources: network, SSDs and CPUs + GPUs, respectively. As shown in Figure 3, we build a 3-stage pipeline to hide the latency of those tasks and maintain a prefetch queue for each stage. Moreover, the prefetching also hides the communication time between stages. For each stage, we create a worker thread that takes jobs from the prefetch queue and feed the corresponding resource. Then the processed results are pushed into the prefetch queue of the next stage. The thread for each stage stalls when the prefetch queue of the next stage is full, i.e., the next stage has already gotten too many unprocessed jobs. The capacity of the prefetch queue is preset according to the execution time of each stage. For the RE stage in Figure 3, we do

not have any execution dependency. Therefore, we extract examples from the distributed file system through network until we have enough unprocessed examples in the queue for ST. The ST stage only depends on the RE—the dependency on EL and JL are eliminated by our cache management system. The ST stage loads referenced parameters in RE from SSDs. Since it is sufficiently large for our cache memory to cache the parameters referenced in the past several passes, we do not need to wait for the completion of EL and JL. We only have to ensure that the loaded parameters cannot be removed from the cache before they are consumed by the EL + JL stage of the corresponding pass. The EL computation depends on the ST stage because we can only start to compute the embedding after we load the parameters into the memory. And the JL training has to wait for the embeddings computed from the EL part. Due to the data dependency, JL and EL are in one pipeline. AIBox with multiple high-end GPUs has a comparable computation power as a distributed cluster with dozens of CPU-only nodes. We can achieve similar execution time for the training of the neural network model. Besides, RE is the simplest and fastest stage. Its latency is hidden in the pipeline. However, comparing with the distributed cluster solution having all the parameters in memory, the Sparse Table operations interacting with SSDs are purely overhead. We are required to optimize ST aggressively to ensure that ST is faster than EL + JL. In this case, the latency of ST is hidden in the pipeline so that we can have a comparable training speed. Therefore, we focus on the design details of the Sparse Table in the following paper.

4 SPARSE TABLE

The Sparse Table aims to store model parameters on SSDs efficiently. It leverages the memory as a fast cache for SSDs while reducing SSD I/O and providing low-latency SSD accesses. It consists of two major components, key hash index and bi-level cache management.

4.1 Key Hash Index

In order to access the parameter file on SSD by feature keys, we have to store 10^{12} key-to-file mappings for the 10^{12} parameters in the CTR prediction model. Storing each key-to-file mapping as a 64-bit value pair in the memory requires 1.6 TB = (8 Bytes key + 8 Bytes offset on SSD) $\times$ 10^{12}, which exceeds the 1 TB memory budget. We have to carefully design the key hash index and the file structures on SSDs to reduce the memory footprint.

We introduce a grouping function that maps a key to a group id such that each group contains m keys in expectation, i.e., $group(key) \rightarrow \{0, 1, \cdots, 10^{12}/m - 1\}$. Here 10^{12} keys are partitioned into $10^{12}/m$ groups. After grouping the keys, we are able to keep the group-to-file mappings in the memory as the memory consumption is only $1/m$ of the original key-to-file mapping. Since the keys are continuous from 1 to 10^{12}, the *group* function can be easily obtained by uniformly partitioning the key space, e.g., $group(key) \rightarrow key \bmod 10^{12}/m$. We set $m = \lfloor BLOCK/(8 + sizeof(value)) \rfloor$, where $BLOCK$ is the I/O unit of SSD and it is determined by the SSD block size (usually 4096), 8 represents the bytes a key occupies, and $sizeof(value)$ is the value (model parameters) size in bytes and it is around 50 bytes in our CTR prediction model. m should never be set to a value that is less than $\lfloor BLOCK/(8 + sizeof(value)) \rfloor$, since an SSD access has to fetch $BLOCK$ Bytes from the disk. It is suboptimal to have a too small m. On the other hand, the greater the m we choose, the smaller the key hash index memory footprint is. However, a large m leads to a large group as we have to fetch multiple pages from SSDs to get one group. Therefore, the m value we set is optimal when the group-to-file mapping occupies acceptable space in the memory. It is true when the block size is much larger than the size of the value.

As a trade-off of the memory footprint, the disadvantage of the grouping strategy is that values in the same group are fetched from the disk, even though they are not referenced in the current mini-batch—I/O bandwidth is wasted. One possible optimization is to group features with high co-occurrence together, e.g., pre-train a learned hash function [26] to maximize the feature co-occurrence. This belongs to another research area of vertical partitioning [42, 63], which is beyond the scope of this paper. Besides, this disadvantage is reduced by the cache management component where we skip the group reading from the disk for cached keys.

4.2 Bi-level Cache Management

The cache management design is guided by the following two challenges: access performance and lifespan of SSDs.

First, the memory access latency is in the order of nano-second while SSD takes microseconds to peek the data because SSD is about 1,000 times slower than memory. However, the parameters in CTR are so sparse and skewed that less than 1% of parameters are referenced in one pass of mini-batches. It provides us with an opportunity to build an in-memory cache system that keeps the frequently used "hot parameters" in the limited memory budget.

The second challenge is that the physical property of SSDs only allows thousands of writes to each storage cell. Parameters are updated in every iteration of the training. It would significantly shorten the SSD lifespan if the parameters are promptly updated. The cache management also acts as a parameter buffer. Buffered parameters are updated in memory without involving SSD I/O. They are materialized to SSDs lazily when the buffer reaches its capacity and the cache replacement policy swaps it out of the cache.

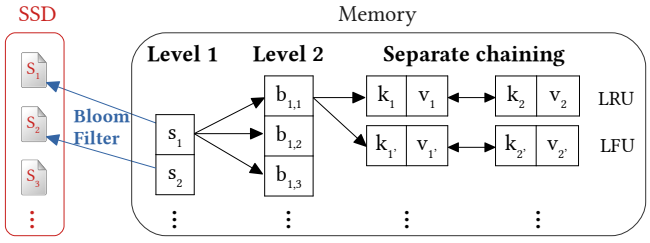


Figure 4: The architecture of the bi-level cache management.

We propose a bi-level cache management to tackle these challenges (Figure 4). Comparing with classic hash tables that handle conflicts by chaining with linked lists, our system has an additional hashing level before the chaining. Moreover, we introduce two separate linked lists for the chaining to optimize the cache probing performance. We also attach a Bloom filter for each SSD file to reduce unnecessary SSD readings. A worker thread runs in the background to maintain the cache policy, i.e., performing cache replacement and writing updated parameters back onto SSDs.

The first level, **Level 1** in Figure 4, hashes the key group id to a cache slot s_i—$s_i = hash_1(g\_id)$, where g_id is the group id computed in the key grouping ($g\_id = group(key)$). Each s_i has a pointer to a file in the SSD that stores the serialized model parameters of all processed keys with $hash_1(g\_id) = s_i$. Because of the extreme sparsity of our model, many parameters are not touched in the early stage of the training. It is inefficient to initialize all the parameters with the default value and save them onto SSDs. For the first time a key is referenced, we have to read a block of data from SSDs and it only returns us the default value. With 10^{12} parameters in our model, it may generate 10^{12} unnecessary SSD readings. Therefore, we perform a lazy parameter initialization, i.e., we do not materialize a parameter until it is referenced by a mini-batch. Besides, we attach a Bloom filter [9, 41] for each materialized file to filter out unnecessary disk readings. Bloom filter is a low-memory-footprint probabilistic data structure that is used to test whether an element exists in a set. It guarantees there are no false negatives as the tested element is definitely not a member of the set when the Bloom filter returns false. In addition, querying Bloom filter is fast and memory-bandwidth-efficient. Only a constant number of bits are accessed. The overhead introduced by the Bloom filter is ignorable comparing with the SSD reading cost. Since we skip probing the SSD file when the Bloom filter returns false, a Bloom filter presumably reduces the number of potential key misses in the SSD files.

The hash code in the second level (**Level 2** in Figure 4) is computed by $hash_2(g\_id, bucket)$, where $bucket$ is a tunable parameter that controls the number of bucket in each cache slot s_i. We introduce the second level hashing to shorten the chaining length of linked lists. Probing over linked lists requires iterating through the linked list until we find the probing key. It may take the length of the linked list steps in the worst case, while other hash table operations only take constant time. Therefore, iterating through the linked list is the most time-consuming step in hash table probing. The second level hashing separates a long chaining linked list of s_i into multiple buckets with short linked lists to reduce the inefficient iterating. The parameter $bucket$ trades off space for the efficient probing. The larger the $bucket$ is, the shorter the linked lists we have in expectation.

For all the key-value pairs hashed to the same bucket in the second level, we construct two separate linked lists, Least Recent Used [45] (LRU) list and Least Frequently Used [53] (LFU) list, to cache those pairs in the memory (**Separate chaining** in Figure 4). The LRU list stores parameters that are used in the current pass of mini-batches. It keeps the recently used key-value pair in the front of the linked list so that it reduces the iterating steps over the linked list to locate the data. On the other side, the LFU list acts as a buffer that contains the candidates for cache replacement. The most frequently used values locate at the head of the linked list so that they can be accessed efficiently. While the least frequently used values are flushed into SSDs and removed from the cache memory. A worker thread running in the background periodically moves the data not in the current pass of mini-batches from the LRU list to the LFU list. The chaining incurs many small linked list node allocations and deallocations. We maintain a memory pool that employs the Slab memory allocation mechanism [6] to manage the creations and deletions of nodes in linked lists.

4.3 Sparse Table Operators

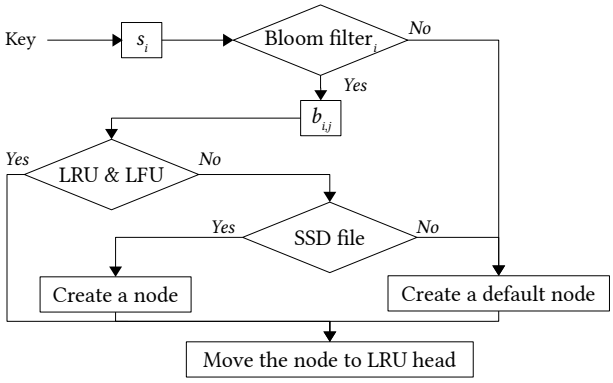


Figure 5: The workflow of the Sparse Table pull operator.

The pull operator of Sparse Table takes a collection of keys that are referenced in a current pass as input and loads the values of these keys into the memory. Figure 5 illustrates the workflow as follows. First, we locate cache slots s_i in the first level of the bi-level cache management by group ids. The group ids are computed from the key grouping hash function. Then, we query the Bloom filter of each located cache slot to test whether those referenced keys may exist (a key does not exist when it is referenced the first time). For each key that fails to pass the Bloom filter, we create a linked list node with default values in the memory and insert the key into the Bloom filter. For the keys that pass the Bloom filter (they may exist in the cache or SSD files), we locate the corresponding cache buckets $b_{i,j}$ on the second layer by their keys and group ids, and iterate through the two linked lists (LRU and LFU) to find the referenced keys. It is a cache-hit when the keys are found in the linked lists since we do not need to visit the SSDs. On the other hand, we have to read the files on SSDs for the keys that are not located in the linked lists. With a low probability that is theoretically guaranteed by the Bloom filter, the referenced keys are not found in the files. We create a default node and update the corresponding Bloom filter for each unfound key as what we do for the keys that fail to pass the Bloom filter. Finally, we move the located/created linked list nodes to the heads of the LRU list. After the pull operation, the parameters of all the referenced keys are stored in the memory cache and they are ready to use.

The push operator of Sparse Table updates model parameters by a given collection of keys. The update increments the referenced parameters by a combined value with the learning rate and the backpropagated gradients. Our workflow (Section 3) ensures that all parameters of the given keys have already been loaded into the memory by the pull operator with the same keys. Therefore, we can locate the values of those parameters through the bi-level cache management and increment them efficiently without involving data writing onto SSDs. The disk writing is deferred until the updated cache nodes are removed from the cache memory. When the cache memory budget is full, i.e., the length of an LFU list is larger than a pre-set threshold MAX_LFU, we remove the last $FLUSH\_STEP$ nodes from the LFU list, where $FLUSH\_STEP$ is a tunable constant. Since nodes are sorted by descending frequency in the LFU list,

the last *FLUSH_STEP* nodes have the least usage frequency. The removed parameters have to be flushed to SSDs because they are updated by the past push operations. We locate the files that contain the removed keys by the first level in the cache management component, read them from SSDs and merge them with the updated values into new files.

4.4 File Management

The push Sparse Table operator keeps creating small files on SSDs while native file systems on modern operating systems, e.g., ext4 [39] and XFS [54], are not designed for handling many small files. The large number of file metadata overwhelms those file systems.

We design a specific light-weight file management system for this task by taking advantage of the fact that small-file creations always come in batches. Having the similar design as key grouping, we group F small files into a large file to create fewer files. In addition, it fully utilizes the disk bandwidth when grouping a batch of independent small-file disk writing into a sequential writing. For each small file, its offset in the large file is saved. The combination of the offset and the name of the large file acts as the "file name" of the small file. This combination is maintained in the cache slot on the first layer of the cache management system. It is updated when we flush the updates onto SSDs. After we create more and more new files, some old files that are not referenced by any cache slots are out-dated. We maintain a reference counter for each large file in the memory. The counter decreases when a "small file" in it is updated and references to a new file. We delete the large file as soon as its counter reaches zero. Moreover, we monitor the disk usage periodically. We merge the files with low reference count and delete them when the disk usage is higher than the model size *MAX_REPLICATION* times. Here *MAX_REPLICATION* is computed by $\textit{SSD capacity} * (85\% + \textit{overprovisioning}) / \textit{model size}$, where *overprovisioning* is a reserved storage space for SSDs specified by manufacturers. In order to maximize SSD lifespans, SSD controllers employ wear leveling algorithms [11] that require free spaces to distribute disk writings. The performance of SSDs drops when there are less than 85% free space [2].

5 EXPERIMENTS

The objective of the experimental evaluation is to investigate the overall performance, as well as the impact of optimizations, of the proposed system. Specifically, we target at answering the following questions in the experiments:

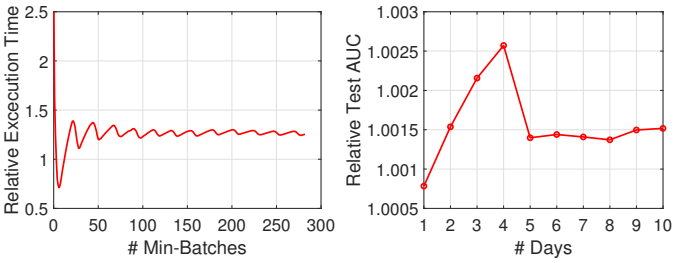
- How does AIBox perform when comparing with the MPI cluster solution?
- What latency is introduced by the Sparse Table operations?
- Does the 3-stage pipeline hide this latency?
- What is the effect of the proposed bi-level cache management on execution time?

We conduct experiments on an AIBox with 8 cutting-edge GPUs. It has server-grade CPUs, 1 TB of memory, and a RAID 0 with NVMe SSDs. For training the baseline models, the nodes in the MPI cluster are maintained in the same data center and are inter-connected through a high-speed Ethernet switch. CPUs in the MPI cluster have similar performance specifications as the CPUs of AIBox.

For the training data of the CTR model, we collect user click history ads logs from Baidu's search engine. The size of the training data is about 10 PB. The average number of non-zero features of each example is about 500. The trained CTR model is evaluated over the production environment of our search engine in an A/B testing manner.

5.1 Performance Evaluation

We evaluate the performance of the proposed AIBox on the training time and the prediction quality. We take an MPI cluster training framework on 75 computing nodes as the baseline. The baseline solution maintains a distributed parameter server that keeps the entire 10-TB model in the memory across all the nodes.



(a) (b)

Figure 6: Comparison of AIBox and a 75-node cluster, in terms of their relative performances, i.e., AIBox/MPI: (a) Training time, (b) CTR prediction AUC.

Figure 6a depicts the cumulative execution time of AIBox relative to the cluster baseline. The training involves about 300 passes of mini-batches. The execution of AIBox in the first pass is slow (267% of the cluster), because the parallelism of our 3-stage pipeline is not involved in the first pass—the execution is sequential. After 2 passes, the pipelines of all the 3 stages are filled. As a result, we can observe that the execution time is reduced significantly after the first several passes. The fluctuation of the relative cumulative time is caused by the different training batch sizes of AIBox and cluster solution. In our legacy cluster-based CTR prediction training framework, passes are obtained by slicing data on timestamps. Since there are more queries in working hours, the amount of search data is not uniformly distributed in one day and the number of examples in a pass varies a lot in the cluster solution. On the other hand, AIBox uniformly partitions the training data into passes of min-batch to make the processing pipeline-friendly. The relative cumulative time across passes is normalized based on the data of each pass. In the figure, we show the relative execution time of processing the same number of training examples. Comparing with 75 nodes, AIBox completes the training within a comparable time frame—25% more than the cluster solution. However, the hardware and maintenance cost of AIBox is less than 10% of the cluster cost.

For the prediction quality evaluation, we use the Area Under the Curve [25] (AUC) to measure the quality of the models. Figure 6b shows the AUC of AIBox relative to the cluster. The AUCs of both trained models are tested online for 10 days in the real sponsored search production environment. Since the CTR prediction accuracy is crucial to the revenue, we have to ensure the optimization on AIBox does not lose accuracy. We observe in Figure 6b that AIBox has

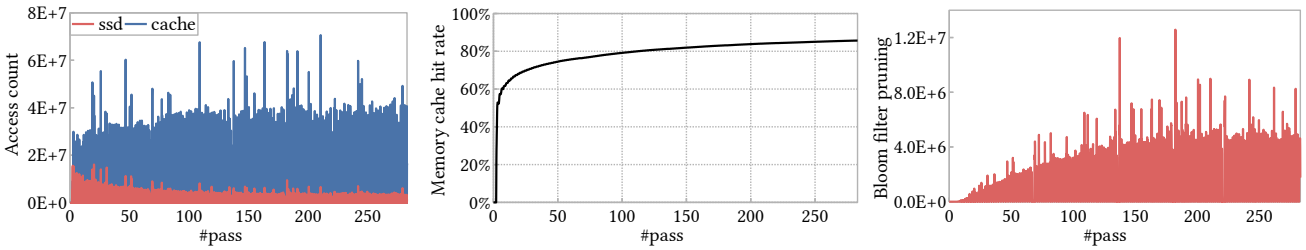


Figure 7: (a) Cache and SSD accesses, (b) Cache hit ratio, and (c) Bloom filter pruning times.

even slightly better AUC than the cluster. This could be just due to the inherent randomness in the experiments, but it might also true in general that AIBox may produce slightly more accurate models. A plausible reason is that the model training on a single node relies on fewer stale parameters than the case on the distributed parameter server. The single-node design of AIBox synchronizes the parameters more frequently by performing a collective communication across GPUs at the end of every pass.

5.2 Optimization Effects

In this subsection, we investigate the effects of the proposed optimizations: 3-stage pipeline and bi-level cache management.

Stage	Avg time	STD DEV
RE	65%	14%
ST	78%	15%
EL+JL	92%(66% + 26%)	11%(8% + 3%)

Table 1: Relative execution time of each stage in the pipeline. 100% is the average time of training a pass of mini-batch.

We first analyze the proposed pipeline. Table 1 presents the relative execution time of each stage in the pipeline. 100% corresponds to the AIBox's average execution time of training one pass of mini-batch. The stage EL + JL takes 92% time of a pass. It dominates the entire pipeline training time. The rest 8% time out of the pipeline is the memory communication and processing overhead. The latencies of RE (65%) and ST (78%) stages are hidden in the pipeline.

Then we evaluate the bi-level cache management. Figure 7a illustrates the number of access counts on cache and SSD, and the cache hit rate is shown in Figure 7b. SSD access count drops after several passes because the "hot parameters" are identified and stored in the LFU list in the cache. The cache hit rate goes to about 80% at the 100^{th} pass and then converges to about 85%. It confirms that the proposed bi-level cache management significantly reduces SSD I/Os.

We finally investigate the Bloom filter pruning. The Bloom filter in our cache system aims at reducing unnecessary SSD readings. Figure 7c demonstrates the Bloom filter pruning times in the cache management. We skip the SSD probing when the Bloom filter returns false. As illustrated in the figure, after 50 passes of mini-batches, the Bloom filter prunes out more than 1 million SSD readings per pass. The number of Bloom filter pruning remains around 4 million after 150 passes of mini-batches. The Bloom filter helps us eliminate millions of SSD readings per pass.

5.3 Discussion

Based on the results presented above, we can answer the questions driving the experimental evaluation. AIBox has comparable training time (125%) with the 75-node MPI cluster solution while only costs less than 10%. Furthermore, the AUC of the model trained by AIBox is even slightly better possibly because more parameter synchronizations are performed in the single-node architecture of AIBox than the distributed parameter server in the cluster solution. The neural network training of EL + JL dominates the execution time of the pipeline so that the latency of the Sparse Table operations is hidden in the 3-stage pipeline. Besides, the proposed bi-level cache management also avoids the bottleneck of the Sparse Table operations. By identifying the"hot parameters" and keeping them in the memory, the average cache hit rate is about 80% with our cache policy.

6 RELATED WORK

In this section, we discuss relevant work including CTR prediction models, in-memory cache management systems, and key-value stores on SSDs.

CTR prediction model: Inspired by the success of deep neural networks in computer vision [29] and natural language processing [5], many deep learning methods [13, 15, 35, 59, 61, 66] are proposed to deal with the large-scale sparse features for CTR. For example, Deep Crossing [50], Wide&Deep Learning [13], YouTube Recommendation CTR model [15] and Deep Interest Network (DIN) [66] are introduced as a class of deep neural networks with several hidden layers. These models first employ an embedding layer to learn low-dimensional dense representations from the sparse inputs and then impose specially designed fully connected layers to capture the latent interactions among features. Product-based Neural Network (PNN) [47] uses a product layer on top of the embedding layer to capture high-order feature interactions. DeepFM [23] and xDeepFM [35] considers factorization machines (FM) to model both low-order and high-order feature interactions for better CTR performance. Overall, all of these models combine an embedding layer and several fully connected layers to learn meaningful representations for CTR prediction. They greatly reduce the feature

engineering jobs and enhance the mode capability. In this paper, we follow this kind of model structure to design AIBox. Furthermore, in order to employ GPUs to accelerate the training, we partition our model into two parts, the memory-intensive embedding learning on CPU and the computation-intensive joint learning on GPU.

In-memory cache management: Many caching policies have been developed for storage systems, such as the LRU-K [45], DBMIN [14], LRFU [31], and Semantic Caching [16]. These algorithms evict cache according to a combined weight of recently used time-stamp and frequency. In the web context, there is extensive work developed for variable-size objects. Some of the most well-known algorithms include Lowest-Latency-First [58], LRU-Threshold [1], and Greedy-Dual-Size [10]. Different from our caching problem, the size of parameters in our CTR prediction model is fixed and there is a clear access pattern during the training, i.e., some parameters are frequently referenced. It is effective to keep those "hot parameters" in the cache by applying an LFU eviction policy. While our additional LRU linked list maintains the parameters referenced in the current pass to accelerate the hash table probing.

Key-value store for SSDs: There is a significant amount of work on key-value stores for SSD devices. The major designs [4, 36] follow the paradigm that maintains an in-memory hash table and constructs an append-only LSM-tree-like data structure on the SSD for updates. FlashStore [17] optimizes the hash function for the in-memory index to compact key memory footprints. SkimpyStash [18] moves the key-value pointers in the hash table onto the SSD. BufferHash [3] builds multiple hash tables with Bloom filters for hash table selection. WiscKey [37] separates keys and values to minimize read/write amplifications. Our Sparse Table design follows the mainstream paradigm, while it is specialized for our training problem. We do not need to confront the challenges to store general keys and values. One reason is that the keys we store are the index of parameters that distributes uniformly. It is unnecessary to employ any sophisticated hashing functions. In addition, the values have a known fixed length and the serialized bucket on SSD exactly fits in a physical SSD block. In this way, the I/O amplification is minimized. Instead of tackling general key-value store challenges, our Sparse Table maintains a bi-level hashing mechanism so that it is more cache-friendly.

7 CONCLUDING REMARKS

Since 2013, Baidu Search Ads (a.k.a. "Phoenix Nest") has been using deep learning CTR models as the standard production tool [20]. While in a way, building CTR models appear to be a matured technology, there are still many possible directions for improving industrial CTR models. One exciting direction is to integrate approximate near neighbor (ANN) search and maximum inner product search (MIPS) technique into the pipeline of sponsored ads production system [20]. This paper is about another fascinating direction: moving the CTR model training from CPUs to GPUs.

In this paper, we introduce AIBox that is capable of training a CTR prediction model with over 10TB parameters on a single node with SSDs and GPUs. We show a novel method that partitions the large CTR prediction model into two parts. With the neural network partitioning, we keep the memory-intensive training part on CPUs and utilize memory-limited GPUs for the computation-intensive part to accelerate the training. In addition, we propose Sparse Table to store model parameters on SSDs. In order to reduce SSD I/O latency, we leverage memory as a fast cache. Furthermore, a 3-stage pipeline that overlaps the execution of network, Sparse Table and the CPU-GPU learning is implemented to hide the latency of Sparse Table operations. We evaluate AIBox by comparing it against the standard MPI distributed cluster solution using production data. The results reveal the effectiveness of AIBox in that it achieves substantially better price-performance ratio than the cluster solution while preserving a comparable training speed.

ACKNOWLEDGEMENT

We thank the anonymous reviewers of KDD 2019 and CIKM 2019, for their helpful comments.

REFERENCES

[1] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A Fox. 1996. Caching Proxies: Limitations and Potentials. *World Wide Web Journal* 1, 1 (1996).
[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference (USENIX ATC)*, Vol. 57.
[3] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. 2010. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems.. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 10. 29–29.
[4] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SIGOPS)*. 1–14.
[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv preprint arXiv:1409.0473* (2014).
[6] Jeff Bonwick et al. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer*, Vol. 16.
[7] Dhruba Borthakur. 2007. The Hadoop Distributed File System: Architecture and Design. *Hadoop Project Website* 11, 2007 (2007), 21.
[8] Y-Lan Boureau, Jean Ponce, and Yann LeCun. 2010. A Theoretical Analysis of Feature Pooling in Visual Recognition. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*. 111–118.
[9] Andrei Z. Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1, 4, 485–509.
[10] Pei Cao and Sandy Irani. 1997. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, Vol. 12. 193–206.
[11] Li-Pin Chang. 2007. On Efficient Wear Leveling for Large-Scale Flash-Memory Storage Systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*. 1126–1130.
[12] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. 2015. Compressing Neural Networks with the Hashing Trick. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. 2285–2294.
[13] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems (RecSys)*. 7–10.
[14] Hong-Tai Chou and David J DeWitt. 1986. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica* 1, 1-4 (1986), 311–336.
[15] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for Youtube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys)*. 191–198.
[16] Shaul Dar, Michael J Franklin, Bjorn T Jonsson, Divesh Srivastava, Michael Tan, et al. 1996. Semantic Data Caching and Replacement. In *Proceedings of 22th International Conference on Very Large Data Bases (VLDB)*, Vol. 96. 330–341.
[17] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-Value Store. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1414–1425.
[18] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 25–36.
[19] Benjamin Edelman, Michael Ostrovsky, and Michael Schwarz. 2007. Internet Advertising and the Generalized Second-Price Auction: Selling Billions of Dollars

Worth of Keywords. *American Economic Review* 97, 1 (2007), 242–259.
[20] Miao Fan, Jiacheng Guo, Shuai Zhu, Shuo Miao, Mingming Sun, and Ping Li. 2019. MOBIUS: Towards the Next Generation of Query-Ad Matching in Baidu's Sponsored Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. 2509–2517.
[21] Denis Foley and John Danskin. 2017. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro* 37, 2 (2017), 7–17.
[22] Thore Graepel, Joaquin Quinonero Candela, Thomas Borchert, and Ralf Herbrich. 2010. Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*. 13–20.
[23] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: a Factorization-Machine based Neural Network for CTR Prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*. 1725–1731.
[24] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems (NIPS)*. 1223–1231.
[25] Jin Huang and Charles X Ling. 2005. Using AUC and Accuracy in Evaluating Learning Algorithms. *IEEE Transactions on Knowledge and Data Engineering* 17, 3 (2005), 299–310.
[26] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. [n. d.]. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Biennial Conference on Innovative Data Systems Research (CIDR)*.
[27] Qing-Yuan Jiang and Wu-Jun Li. 2017. Deep Cross-Modal Hashing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 3232–3240.
[28] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 5 (2011), 7–17.
[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*. 1097–1105.
[30] Jason Kuen, Xiangfei Kong, Zhe Lin, Gang Wang, Jianxiong Yin, Simon See, and Yap-Peng Tan. 2018. Stochastic Downsampling for Cost-Adjustable Inference and Improved Regularization in Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 7929–7938.
[31] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.* 12 (2001), 1352–1361.
[32] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vol. 14. 583–598.
[33] Ping Li, Art B Owen, and Cun-Hui Zhang. 2012. One Permutation Hashing. In *Advances in Neural Information Processing Systems (NIPS)*. 3122–3130.
[34] Ping Li, Anshumali Shrivastava, Joshua Moore, and Arnd Christian König. 2011. Hashing Algorithms for Large-Scale Learning. In *Advances in Neural Information Processing Systems (NIPS)*. 2672–2680.
[35] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. 1754–1763.
[36] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. 1–13.
[37] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 5.
[38] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. *arXiv preprint arXiv:1803.04014* (2018).
[39] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The New Ext4 Filesystem: Current Status and Future Plans. In *Proceedings of the Linux Symposium*, Vol. 2. 21–33.
[40] H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. 2013. Ad Click Prediction: a View from the Trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 1222–1230.
[41] Michael Mitzenmacher. 2002. Compressed Bloom Filters. *IEEE/ACM Transactions on Networking (TON)* 10, 5 (2002), 604–612.
[42] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. 1984. Vertical Partitioning Algorithms for Database Design. *ACM Transactions on Database Systems (TODS)* 9, 4 (1984), 680–710.
[43] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *ACM Queue* 6, 2, 40–53.
[44] NVIDIA. 2018. NVLink Fabric. https://www.nvidia.com/en-us/data-center/nvlink. (2018). Accessed: 2019-01-29.
[45] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. 22, 2 (1993), 297–306.
[46] Alexander D Poularikas. 1998. *Handbook of Formulas and Tables for Signal Processing*. CRC press.
[47] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. 2016. Product-based Neural Networks for User Response Prediction. In *Proceedings of the IEEE 16th International Conference on Data Mining (ICDM)*. 1149–1154.
[48] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems (NIPS)*. 693–701.
[49] Bianca Schroeder and Garth Gibson. 2010. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Transactions on Dependable and Secure Computing* 7, 4 (2010), 337–350.
[50] Ying Shan, T Ryan Hoens, Jian Jiao, Haijing Wang, Dong Yu, and JC Mao. 2016. Deep crossing: Web-scale Modeling without Manually Crafted Combinatorial Features. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 255–262.
[51] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for Sublinear Time Maximum Inner Product Search (MIPS). In *Advances in Neural Information Processing Systems (NIPS)*. 2321–2329.
[52] Marc Snir, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. 1998. *MPI–the Complete Reference: the MPI Core*. Vol. 1. MIT press.
[53] Leonid B Sokolinsky. 2004. LFU-K: An Effective Buffer Management Replacement Algorithm. In *International Conference on Database Systems for Advanced Applications (DASFAA)*. 670–681.
[54] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System. In *USENIX Annual Technical Conference (USENIX ATC)*, Vol. 15.
[55] Shulong Tan, Zhixin Zhou, Zhaozhuo Xu, and Ping Li. 2019. *Fast Item Ranking under Neural Network based Measures*. Technical Report. Baidu Research.
[56] Shulong Tan, Zhixin Zhou, Zhaozhuo Xu, and Ping Li. 2019. On Efficient Retrieval of Top Similarity Vectors. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
[57] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature Hashing for Large Scale Multitask Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*. 1113–1120.
[58] Roland P Wooster and Marc Abrams. 1997. Proxy Caching that Estimates Page Load Delays. *Computer Networks and ISDN Systems* 29, 8 (1997), 977–986.
[59] Jun Xiao, Hao Ye, Xiangnan He, Hanwang Zhang, Fei Wu, and Tat-Seng Chua. 2017. Attentional Factorization Machines: Learning the Weight of Feature Interactions via Attention Networks. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*. 3119–3125.
[60] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR)*. 6:1–6:11.
[61] Shuangfei Zhai, Keng-hao Chang, Ruofei Zhang, and Zhongfei Mark Zhang. 2016. Deepintent: Learning Attentions for Online Advertising with Recurrent Neural Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 1295–1304.
[62] Shanshan Zhang, Ce Zhang, Zhao You, Rong Zheng, and Bo Xu. 2013. Asynchronous Stochastic Gradient Descent for DNN Training. In *Proceedings of 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 6660–6663.
[63] Weijie Zhao, Yu Cheng, and Florin Rusu. 2015. Vertical Partitioning for Query Processing over Raw Data. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management (SSDBM)*. 15:1–15:12.
[64] Weijie Zhao, Shulong Tan, and Ping Li. 2019. *SONG: Approximate Nearest Neighbor Search on GPU*. Technical Report. Baidu Research.
[65] Lei Zheng, Vahid Noroozi, and Philip S Yu. 2017. Joint Deep Modeling of Users and Items Using Reviews for Recommendation. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM)*. 425–434.
[66] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-Through Rate Prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. 1059–1068.