BANDANA: USING NON-VOLATILE MEMORY FOR STORING DEEP LEARNING MODELS

ABSTRACT

Typical large-scale recommender systems use deep learning models that are stored on a large amount of DRAM. These models often rely on embeddings, which consume most of the required memory. We present Bandana, a storage system that reduces the DRAM footprint of embeddings, by using Non-volatile Memory (NVM) as the primary storage medium, with a small amount of DRAM as cache. The main challenge in storing embeddings on NVM is its limited read bandwidth compared to DRAM. Bandana uses two primary techniques to address this limitation: first, it stores embedding vectors that are likely to be read together in the same physical location, using hypergraph partitioning, and second, it decides the number of embedding vectors to cache in DRAM by simulating dozens of small caches. These techniques allow Bandana to increase the effective read bandwidth of NVM by 2-3$\times$ and thereby significantly reduce the total cost of ownership.

1 INTRODUCTION

An increasing number of web-scale applications are relying on deep learning models, including online search (Clark, 2015), online ads (Zhou et al., 2018) and content recommendation systems (Covington et al., 2016). Typically, the precision of deep learning algorithms increases as a function of the model size and the number of features. Therefore, application providers are dedicating ever more compute and storage resources to training, storing and accessing deep learning models.

For example, at Acme[1], thousands of servers are dedicated to storing deep learning models to recommend relevant posts or content to users. The deep learning features are often represented by vectors called embeddings, which encode the meaning of each feature, such that similar vectors are closer in the embedding Euclidean space. To compute the most relevant posts to serve each user, Acme uses two types of embeddings: user and post embeddings. Post embeddings represent the features of the post themselves (e.g., the main words), while the user embeddings represent features unique to each user, which represent their topics of interest and past activity. At Acme, both types of embeddings are fully stored in DRAM, in order to enable real-time access for computing the most relevant content for each user.

However, DRAM is a relatively expensive storage medium, and in fact has gotten even more expensive recently, due to shortages in global supply (Wu, 2018; Bary, 2018). In this work, our goal is to minimize the amount of DRAM used to store embeddings, and therefore the total cost of ownership (TCO). Since post embeddings need to go through more ranking and processing, they have a much longer pipeline than user embeddings. Therefore, user embeddings can be read early in the process from a slower but cheaper storage medium than DRAM.

[1]Large web company, name anonymized for double-blind submission.

Non-volatile Memory (NVM), also termed Non-volatile Main Memory or persistent memory, offers an attractive storage medium for user embeddings, since it costs about an order of magnitude less per bit than DRAM, and its latency and throughput can satisfy the requirements of computing user embeddings. However, even though NVM provides sufficient latency to meet the system's requirements, its bandwidth is significantly lower than DRAM. Exarcerbating the problem, NVM devices offer maximum bandwidth only if the size of reads is 4 KB or more, while user embedding vectors are only 64-128 B. Therefore, naïvely substituting DRAM for NVM results in underutilized bandwidth, and causes both its latency to increase and the application's throughput to drop significantly.

We present Bandana, an NVM-based storage system for embeddings of recommender systems, which optimizes the read bandwidth of NVM for accessing deep learning embeddings. Bandana uses two primary mechanisms to optimize bandwidth: storing embedding vectors that can be prefetched together to DRAM, and deciding which vectors to cache in DRAM to maximize NVM bandwidth.

Prefetching embedding vectors. Our NVM device benchmarks show that to optimize bandwidth, NVM needs to be read at the granularity of a 4 KB block or more. Therefore, Bandana stores multiple embedding vectors that are likely to be accessed together in the same 4 KB NVM block, and when one of the objects needs to be read, it has the option of prefetching the entire block to DRAM. We evaluate two techniques for partitioning the vectors into blocks: Social Hash Partitioner (SHP) (Kabiljo et al., 2017), a supervised hypergraph partitioning algorithm that maximizes the number of vectors in each block that were accessed in the same query, and K-means (Lloyd, 1982; Arthur & Vassilvitskii, 2007), an unsupervised algorithm that is run recursively in two stages. We found that SHP doubled the effective bandwidth compared to K-means for some workloads.

Caching vectors in DRAM. Even after storing related vectors together, some blocks contain vectors that will not be read and can be discarded. Therefore, Bandana decides which vectors to keep in DRAM, by using a Least Recently Used (LRU) queue, and only inserting objects from prefetched blocks to the queue that have been accessed t times in the past. We find that the performance varies widely across different embedding tables based on the value of t and the cache size. Therefore, inspired by recent research in key-value caches (Waldspurger et al., 2017), Bandana runs dozens of "miniature caches" that simulate the hit rate curve of different values of t for each embedding table, with a very low overhead. Based on the results of the simulations, Bandana picks the optimal threshold for each embedding table.

We demonstrate that Bandana significantly improved the effective bandwidth of NVM, enabling it to be used as a primary storage medium for embeddings. To summarize our contributions:

1. To our knowledge, Bandana is the first system that leverages NVM to store deep learning models. It is also one of the first published systems to use NVM in large-scale production workloads.
2. Using the past access patterns of embedding vectors, Bandana applies hypergraph partitioning to determine which vectors are likely to be accessed together in the future.
3. Applies recent techniques from key-value caching to run lightweight simulations of dozens of miniature caches to determine how aggressively to cache prefetched vectors for different cache sizes.

2 BACKGROUND

In this section we provide background on two topics: deep learning embedding vectors and how they are used at Acme for recommending posts, as well as NVM.

2.1 Embedding Vectors

The goal of Acme's post recommendation system is to recommend relevant content to users. A straightforward way to train a ranking system for this purpose, would be to encode the post and user features and use them for predicting the

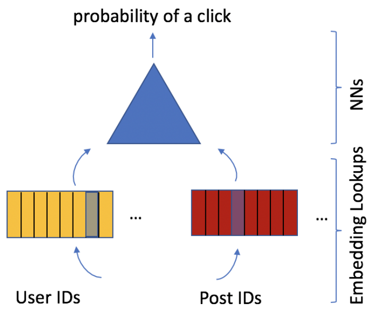


Figure 1. A deep learning recommendation model.

likelyhood of a click. For example, we can represent users based on the pages they liked. In this case, each page would correspond to a unique ID, and IDs would be mapped to an element index in a binary vector. For instance, if a user liked pages with the IDs 2 and 4, out of a total number of 5 pages, the user feature vector would be (0,0,1,0,1).

On the other hand, we could represent posts based on the words used to describe them. In this scenario, each word would correspond to a unique ID, and once again IDs would be mapped to an element index in a binary vector. For example, if the post's text is "red car" out of word dictionary "bicycle, mototrcycle, car, blue, red, green", the post feature vector would be (0, 0, 1, 0, 1, 0).

Since the total number of words in different languages is on the order of millions, while the number of pages is on the order of billions, such binary vectors would be very large and sparse (i.e., will contain mostly zeros). Moreover, such a naïve representation would not apply any contextual information of one page to another. In other words, very similar pages would still be encoded as separate IDs.

However, instead of representing each word or page by a single binary digit, if we represent them by a short vector, we could represent the similarity between words or pages. The mapping of items into a vector space is called an embedding. Embeddings are learned and created in such a way that sparse IDs with similar meaning (in terms of semantics, engagement, etc.) will also be located closer in terms of distance. The distance is often measured in Euclidian space. Recommending posts is a specific use case of recommender systems (Covington et al., 2016; Cheng et al., 2016; Wang et al., 2017), in which the goal is to recommend the most relevant content based on past user behavior.

At Acme, embedding vectors are stored in dedicated tables, where each column represents an embedding vector, and its column ID corresponds to its ID. Acme maintains two types of embeddings: user and post embeddings. Each user embedding table can typically represents some kind of user behavior, such as clicks, likes, and page views, and each embedding vector is a specific action taken by the user. The post tables can represent the actual content of

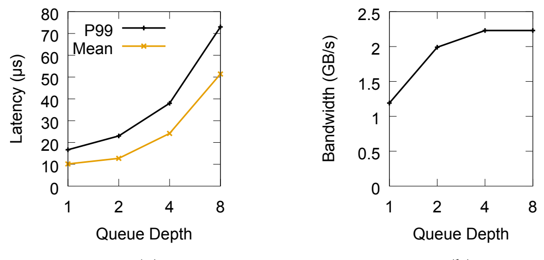


Figure 2. The latency and bandwidth for a 4KB random-read workload with variable queue depths.

the post, such as the specific phrases appearing in the post. The recommendation model receives input IDs, extracts the corresponding embeddings, and processes them with deep neural networks (NNs), as shown on Figure 1.

The typical vector dimension in our models is between 32 to 64, where each element occupies 1-4 bytes. Embedding tables can contain tens of millions of embedding vectors, requiring on the order of GBs per table. Due to their latency requirements, these tables are usually stored in DRAM.

The embedding vectors are computed during training, where for each data sample (e.g. user and post pair) only the columns accessed by the corresponding IDs are modified. Therefore, as the training proceeds through the dataset most (if not all) columns are updated, many of which are updated multiple times. The embeddings are then used without any adjustments or modifications during inference. The vectors may be retrained every few hours.

2.2 NVM

Non-volatile Memory (NVM), is a new memory technology that provides much lower latency and higher throughput than flash, but lower performance than DRAM, for a much lower cost than DRAM. NVM can be used in two form factors: the DIMM form factor, which is byte-addressable, or alternatively as a block device. The DIMM form factor is currently not supported by Intel processors (Mellor, 2018), and is much more expensive than using NVM as a block device. Therefore, for our use case, we focus on the block device form factor.

To understand how to use NVM, we explored its performance characteristics. For this purpose, we ran a widely used I/O workload generator, Fio 2.19 (fio), on an NVM device. We ran the Fio workloads with 4 concurrent jobs using the Libaio I/O engine with different queue depths. The queue depth represents the number of outstanding I/O requests to the device, which is a proxy for how many threads we run in parallel. We measured the latency and bandwidth of an NVM device with a capacity of 375 GB.

Figure 2 presents the average latency, P99 latency, and bandwidth for a read-only workload with random accesses of 4 KB. The results show that there is a trade-off between latency and bandwidth: a higher queue depth provides higher read bandwidth, at the expense of higher latency. Note that even at a high queue depth, NVM's read bandwidth (2.3 GB/s) is still $> 30\times$ lower than DRAM (e.g., 75 GB/s).

Note that unlike DRAM, NVM's endurance deteriorates as a function of the number of writes over its lifetime. Typical NVM devices can be re-written 30 times a day, or they will start exhibiting errors. Fortunately, the rate of updating the vectors at Acme is often between 10-20 times a day, which is below the rate that would affect the endurance of the NVM devices.

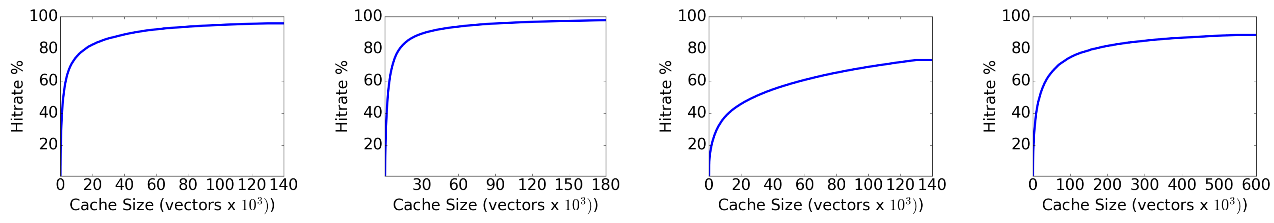
3 Workload Characterization

This section presents a workload characterization of the user embeddings at Acme. We analyze a production workload containing 1 billion embedding vector lookups, representing traffic of over one hour for a single model. Currently, the number of models per server and the size of each model are bounded by the DRAM capacity of the server.

Each user embedding table typically represents a different class of user behavior. For example, a table might represent pages liked by the user, where each embedding vector represents a page. Hence, a request in Bandana usually incorporates multiple tables and contains multiple vector lookups inside each table. Because different posts are ranked for a single user, the post embeddings are read much more frequently, and post lookups constitute about 95% of the total embedding reads. On the other hand, user embeddings contain more features, and consume about 75% of the total DRAM capacity.

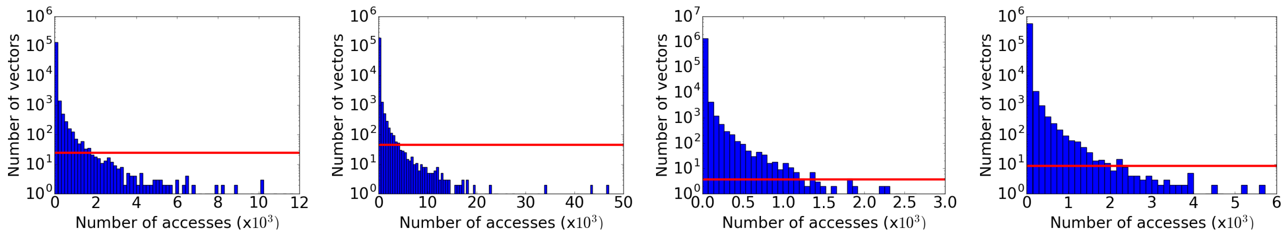
In the model we analyze, embedding vector are 128 bytes containing 64 elements of type `fp16`. Table 1 describes the characteristics of some representative user embedding tables in the model. Each embedding table is comprised of 10-20 million vectors (between 1.2 GB to 2.4 GB). The average number of vectors included in a single request varies across the tables, with 17.68 vector lookups (on average) in embedding table 8, and up to 92.8 vector lookups (on average) in embedding table 2. The table also presents the vector lookup distribution across the user embedding tables. The largest part of vector lookups is consumed by embedding table 2, which serves 25% of the user embedding lookups. Compulsary misses describe how many of these lookups were unique (i.e., how many lookups correspond to vectors that were not read before in the trace). The lower the percentage of compulsory misses, the more likely the table can be effectively cached.

To gain more insight on the reuse of the user embedding vectors, we calculate the stack distances (Mattson et al., 1970)



(a) Table 1 (b) Table 2 (c) Table 6 (d) Table 7

Figure 3. Hit rate curves of the user embedding tables with the top number of lookups.



(a) Table 1 (b) Table 2 (c) Table 6 (d) Table 7

Figure 4. Access histograms of the user embedding tables with the top number of lookups. Red lines represent the average number of accesses per vector.

Table 1. Characterization of the user embedding tables.

TABLE	VECTORS	AVG REQUEST SIZE	% OF TOTAL LOOKUPS	COMPULSARY MISSES
1	10M	34.83	9.44%	4.16%
2	10M	92.75	25.14%	2.19%
3	20M	26.67	7.23%	24.29%
4	20M	25.14	6.82%	19.46%
5	10M	30.22	8.19%	22.68%
6	10M	53.50	14.5%	26.94%
7	10M	54.35	14.73%	11.36%
8	20M	17.68	4.79%	60.83%

of each embedding table. To compute them, we assume each table is cached in an infinite LRU queue, where the stack distance of a vector is its rank in an LRU queue at the time it is requested, counted from the top of the eviction queue. For example, if the requested vector is at the second object in the eviction queue when it is requested, its stack distance is equal to 2. This allows us to compute the hit rate curve as a function of the memory allocated to embedding table. Figure 3 depicts the hit rate curves of the four embedding tables with the top number of lookups, in a trace of one billion requests. Figure 4 shows the access histogram of these tables, where each bar depicts how many vectors (X axis) were read a certain number of times (Y axis). The histograms show that there is a very high variance in the access patterns of the tables. For example, table 2 contains vectors that are read 100,000s of times, while for table 7 there are no vectors that are read more than 1,000 times.

4 DESIGN

This section presents the design choices and trade-offs when designing an NVM-based storage system for embedding tables.

4.1 Baseline

A simple approach for using NVM to store recommender system embedding tables is to cache a single vector that is read by the application in DRAM, and evict a single old vector at a time. We refer to this policy throughout the paper as the *baseline policy*.

Figure 5 presents the latency as a function of the throughput of the NVM device for the baseline policy, as well as for a synthetic workload that issues random 4 KB reads from the NVM. The reason that latency under the baseline policy as a function of throughput is much higher than that of a device where we issue 4 KB reads, is due to the fact that NVM reads in the block device form factor are in the granularity of 4 KB blocks, while the size of an embedding vector is only 128 B. Therefore, the baseline policy is not utilizing more than 96% of the read bandwidth of the NVM device. We use the term *effective bandwidth* to denote the percentage of NVM read bandwidth that is read by the application. In the case of the baseline policy, the effective bandwidth is only 4% of the total bandwidth of the NVM, and the rest is discarded. Therefore, under a high load, when the effective bandwidth is so low, the latency of NVM spikes (and throughput drops).

Instead of reading a single vector to DRAM when it is

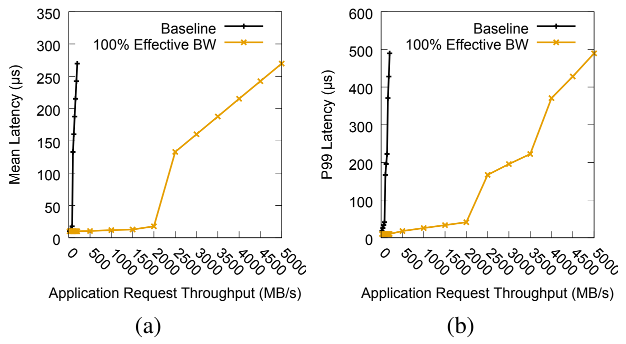


Figure 5. Mean and P99 latencies as a function of the throughput of a 375 GB NVM device. The baseline policy represents the scenario where the application issues a 128 B NVM read for each embedding vector. The 100% effective bandwidth line represents the performance of the NVM device with random 4 KB reads.

accessed, an alternative approach would be to read all 32 vectors stored in its physical 4 KB block. However, when we use a limited cache size, the policy of caching all 32 vectors that belong to a block, performs even worse than fetching one vector at a time, and reduces the effective bandwidth by more than 90% compared to the baseline policy. This is due to the fact that the vectors stored in the same physical block as the requested vector are not read before they are evicted, since they have no relationship with the vector that had just been read. Therefore, fetching them together with the requested vector offers no benefit (in §4.3 we further analyze the performance of caching the prefetched vectors without ordering them).

In summary, the limited effective bandwidth is the main bottleneck to adopting NVM as an alternative for DRAM for storing embeddings. Therefore, the main objective of Bandana is to maximize the *effective bandwidth increase* over the baseline policy.

4.2 Storing Related Vectors Together

If vectors that are accessed at short intervals from each other are also stored physically in the same blocks, Bandana would be able to read fewer NVM blocks while fetching the same amount of data. We explore two directions for physically placing the embedding vectors.

Semantic partitioning assumes that two vectors that are close in the Euclidian space may also share a semantic similarity, and thus should be stored together. We use an unsupervised K-means clustering algorithm to evaluate this direction. *Supervised partitioning* uses the history of past accesses to decide which vectors should be stored together.

In order to evaluate the benefits of physically storing related embedding vectors compared to the baseline policy, we start by experimenting with an unlimited cache (i.e. a DRAM cache with no evictions), in which all blocks that are read

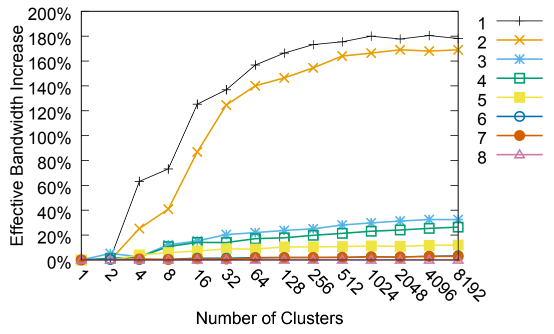


Figure 6. Effective bandwidth increase when ordering embedding vectors according to their K-means clusters. The lines represent different embedding tables.

are cached. We calculate the effective bandwidth increase by comparing the number of block reads from the NVM using a production workload of 1 billion requests.

4.2.1 *Semantic Partitioning with K-Means*

Our first hypothesis for deciding where to place vectors is that vectors that are close in the Euclidian space are also accessed at close temporal intervals. The intuition is that if vectors are close to each other, they might represent similar types of content (e.g., similar pages), and would be accessed at similar times.

We can formalize the problem by expressing the embedding table E as a $m \times n$ matrix

$$E = [\mathbf{v}_1, ..., \mathbf{v}_n] \qquad (1)$$

Suppose that n embedding vectors are semantically meaningful, in other words, two vectors $\mathbf{v}_i$ and $\mathbf{v}_j$ that are close to each other in Euclidian distance $||\mathbf{v}_i - \mathbf{v}_j||_2$ sense are also more likely to be accessed at close temporal intervals. Then, we would like to find a column reordering $\mathbf{p}$, such that

$$\min_p \sum_{i=0}^{n} ||\mathbf{v}_{p(i)} - \mathbf{v}_{p(i+1)}||_2 \qquad (2)$$

We can approximate the solution to this problem by using K-means to cluster the vectors based on Euclidian distance, and sort them so that vectors in the same cluster are ordered next to each other in memory.

Figure 6 shows the effective bandwidth increase for different number of clusters. The results show that for certain tables (e.g., tables 1 and 2), the effective bandwidth is increased significantly, while for others it is not. Note that for example, table 8, which does not experience a large effective bandwidth increase, suffers from a high compulsory miss rate (see Table 1). Ideally, we would use K-means with a large number of clusters. Figure 7(a) shows that the runtime of K-means increases exponentially as a function of

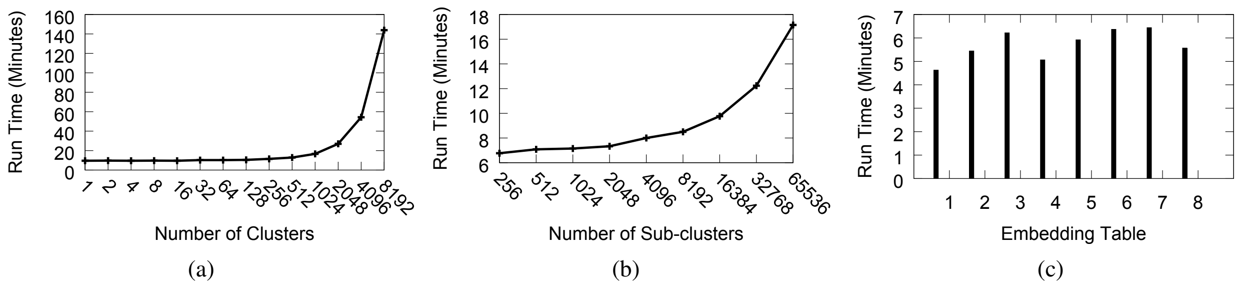


Figure 7. (a) The runtime of the K-means algorithm on embedding table 4, using the Faiss library with 20 iterations and 24 threads. (b) The runtime of two-stage K-means algorithm on embedding table 4, using the Faiss library with 20 iterations and 24 threads. (c) The runtime of the SHP algorithm with 16 iterations and 24 threads per embedding table.

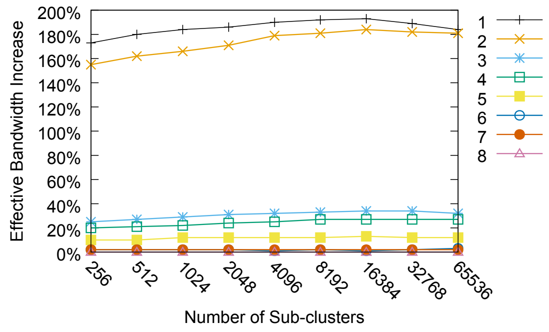


Figure 8. Effective bandwidth increase when ordering the embedding vectors using recursive K-means. The lines represent different embedding tables.

the number of clusters. Therefore, K-means does not scale to a large number of clusters (e.g., 625,000).

In order to reduce the runtime, we also experiment running an appromixation of K-means by running the algorithm recursively. We first run K-means to cluster the embeddings into 256 clusters, then recursively run it again on each of the clusters, creating "sub-clusters". Figure 8 depicts the effective bandwidth increase for different number of sub-clusters, and Figure 7(b) shows the corresponding runtime as measured on table 4. The results show that using recursive K-means does not reduce the effective bandwidth, and there is no benefit increasing the number of clusters beyond 8,192.

4.2.2 *Supervised Partitioning with SHP*

In general, a small Euclidean distance of vectors in embedding tables does not always guarantee they are going to be accessed together in time. Another problem of relying on the Euclidean distance, especially in the context of Acme, is that if vectors are frequently updated due to re-training (e.g., every hour), their Euclidean distance can change, and therefore K-Means would need to be re-run at every update. Therefore, we also experiment with an approach that does not rely on Euclidean distances, but rather on the past access patterns of each vector. This approach would not require re-computing the classifier each time the vectors get re-trained, since the identity of the vectors remains the same, even if their values get updated.

To do so, we are inspired by techniques introduced in the context of partitioning hypergraphs for optimizing data placement for databases (Devine et al., 2006; Kabiljo et al., 2017). Suppose that we have a representative sequence of accesses to the embedding table, which are represented as sparse IDs. Let these accesses be organized into lookup queries Q_j, each corresponding to a particular user. Then, we would like to find a column reordering **p** such that columns accessed together by the same user are stored in the same block.

We can find a solution to this problem by mapping it to a hypergraph. Let $H = (\mathcal{D}, \mathcal{E})$ be a hypergraph with a set of vertices $\mathcal{D}$ corresponding to sequence of accesses and a set of hyperedges $\mathcal{E}$ corresponding to lookup queries $\mathcal{Q}_j$. Also, let p be a partitiong of the vertices $\mathcal{D} = \bigcup \mathcal{D}_i$ into disjoint blocks $\mathcal{D}_i$ for $i = 1, ..., k$. Then, notice that the spatial locality of accesses can be expressed as minimizing the *average fanout*:

$$\min_p \frac{1}{n} \sum_{j=1}^{n} \left(\sum_{i=1}^{k} \mathtt{intersect}(Q_j, \mathcal{D}_i) \right) \quad (3)$$

where *fanout* in parenthesis is the number of blocks that need to be read to satisfy the query, with

$$\mathtt{intersect}(Q_j, \mathcal{D}_i) = \begin{cases} 1 & \mathtt{if}\ Q_j \bigcap \mathcal{D}_i \neq \emptyset \\ 0 & \mathtt{otherwise} \end{cases} \quad (4)$$

Notice that the average fanout measures the number of blocks accessed by each query, rather than the general proximity of accesses. Therefore, we temporally approximate vectors that are accessed in the *same query*. We can start with two blocks and apply the algorithm recursively on them (Kabiljo et al., 2017).

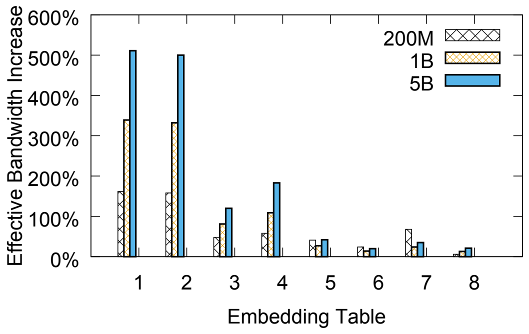


Figure 9. Effective bandwidth increase when ordering the embedding vectors using SHP with an unlimited DRAM cache, as a function of the number of requests used to train SHP.

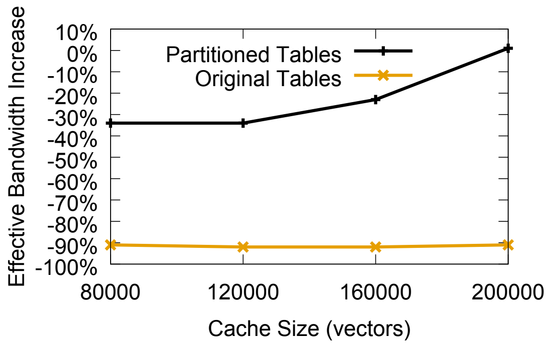


Figure 10. Effective bandwidth increase when ordering the embedding vectors using SHP with a limited DRAM cache, with a policy of treating prefetched vectors the same as vectors that are read by the application. The figure also depicts the effective bandwidth of the unsorted original tables.

We configure the SHP algorithm blocks to contain 32 embeddings. We first run the algorithm on a set of up to 5 billion requests, then measure the bandwidth reduction on a separate trace of 1 billion requests.

Figure 9 depicts the effective bandwidth increase per table, when running the SHP algorithm with different number of requests. Overall, SHP exeeds the bandwidth savings achieved with K-means for all tables. Utilizing larger datasets for running the algorithm improves its accuracy and the corresponding effective bandwidth (we did not see a significant bandwidth improvement for using datasets larger than 5 billion requests). Figure 7(c) shows the SHP runtime per table, running with 16 iterations and 24 threads.

4.3 Caching the Embedding Tables

So far, we assumed that Bandana uses an infinite cache. However, as we noted above, the amount of DRAM we can dedicate to each table is limited. Therefore, Bandana also needs to implement an eviction policy to decide which vectors to cache in DRAM. We experiment with using an eviction policy of Least Recently Used (LRU).

The first question when deciding which vectors to cache in DRAM, is how to treat the vectors that were pre-fetched when Bandana reads the whole block. Figure 10 depicts the effective bandwidth increase when all of the vectors in a block are cached and treated the same as the actual requested vector for the original tables and for the partitioned tables. As the figure shows, simply allocating all 32 vectors in a block to the cache will trigger 32 evictions of potentially more useful vectors that are ranked higher in the eviction queue, reducing the cache hit rate and reducing the effective bandwidth significantly.

4.3.1 Caching the Prefetched Vectors

Even though our ordering algorithms improve the spatial locality of blocks, some prefetched vectors are not accessed at all. This led us to experiment with inserting them in different positions in the eviction queue. Inserting prefetched vectors at a lower position in the queue prevents them from triggering the eviction of hot vectors, but also may shorten their lifetime in the cache and make them less likely to get accessed before they are evicted, thus decreasing the effective bandwidth. Overall, we noticed that while improving the hit rate and bandwidth (compared with inserting prefetches at the top of the queue), this method did not significantly affect the performance for lower cache sizes, and still provided low (and sometimes negative) bandwidth benefits. The main reason for this is that all prefetched vectors are still allocated to the cache, without filtering the less useful vectors. Figure 11(a) presents the bandwidth reduction in embedding table 2 when inserting prefetched vectors to different positions in the queue, over a baseline with no prefetches. The X axis represents the prefetch insertion position relative to the top of the eviction queue (e.g. 0.5 and 0 mean the middle and top of the queue, respectively).

Instead of allocating prefetched vectors to a lower point in the queue, we can use an admission control algorithm that decides whether prefetched vectors enter the queue at all. As a first step, we use a separate LRU list, which we term a *shadow cache*, which stores only the index of the vectors, without storing their content. We allocate only vectors that were explicitly read, thus simulating another cache (that has no prefetched vectors) without actually caching the value of the embedding vectors. When a block is read from the NVM, its vectors are prefetched only if they already appear in the shadow cache (note that the vector read by the application is always cached). Figure 11(b) demonstrates the effective bandwidth increase as a function of the size of the shadow cache with table 2. The shadow cache size is calculated using a multiplier over the real cache size. For example, a mutiplier of 1.5 for a cache of 80,000 vectors means that the shadow cache size is 120,000 vectors.

As shown in Figure 11(b), the shadow cache produces a very small effective bandwidth increase when used as an admission policy. The existence of a vector in the shadow cache does not correlate with its usefulness as a prefetched vector. We try to combine both methods, by using the shadow cache to decide where to allocate the prefetched vector in queue. If the prefetched vector hits in the simulated cache, it is allocated to the top of the queue. Otherwise, it is allocated to the separate insertion position. Figure 11(c) shows the effective bandwidth increase in embedding table 2 when using this method, over a baseline with no prefetches.

4.3.2 *Dealing with Rarely Accessed Vectors*

We try to improve bandwidth savings further by leveraging the insight that during SHP run, some vectors are rarely accessed, as demonstrated in Figure 4. SHP has very limited information about such vectors and on how to sort them. However, since SHP performs balanced partitioning, all the vectors are paritioned to equally-sized blocks, and blocks may contain vectors that were rarely (or never) accessed during the SHP run. In such cases SHP will simply assign them to arbitrary locations in blocks that have free space.

Driven by this insight, Bandana collects statistics on the number of times each vector was accessed during the SHP run (i.e, how many queries contained each vector). When reading a 4 KB block from NVM, vectors will be prefetched only if they were accessed $> t$ times during the SHP run. Figure 12 presents the effective bandwidth increase with different threshold values t for table 2, compared to a baseline of no prefeches. This policy significantly improves effective bandwidth. The number of vector accesses during an SHP run correlates with their utility as prefetched vectors, since SHP has more confidence in assigning a useful location for vectors that appeared in many queries. For smaller cache sizes, the price of cache evictions is higher, hence Bandana should utilize higher thresholds to filter out the more speculative prefetches. For higher cache sizes, Bandana should use lower thresholds to more aggressively prefetch.

4.3.3 *Configuring the Cache Parameters with Simulations*

As Figure 12 shows, the optimal threshold varies across different cache sizes. Picking an a-priori one-size-fits-all threshold for all the tables would lead to a low effective bandwidth. Ideally, Bandana should automatically pick the right threshold for each table and cache size and automatically tune it for each table and cache size.

To do so, we borrow an idea used from key-value caches, called "miniature caches" (Waldspurger et al., 2017). The idea behind miniature caches is to simulate the hit rate curve of multiple different cache configuration, or in our case, simulate the cache with different thresholds for prefetched vectors, and pick the one that provides the highest hit rate for a given cache size. The main problem with this approach is how to simulate multiple caches in real-time without incurring a high performance and memory overhead.

Miniature caches uses the insight that hit rate curves can be estimated efficiently without having to use the entire access workload, but rather by randomly sampling requests from the workload and computing the hit rate curve for the sampled requests. For example, if the total cache is of size S, and we sample the request stream at a rate of $\frac{1}{N}$, the miniature cache only needs to track $\frac{S}{N}$ vectors. In addition, the miniature cache does not have to store the value of the objects, only their IDs.

In our case, we find that in order to accurately simulate a cache, we can down-sample its requests by a factor of 1000. Table 2 compares the ideal threshold when running embedding table 2 with different cache sizes, to the thresholds chosen by the miniature cache simulations with different sampling rates. The results show that there is not a big difference in the effective bandwidth between the ideal thresholds and the ones chosen by the simulations. It also shows that for larger caches, Bandana can use a more relaxed threshold, while for smaller caches with less DRAM, Bandana benefits from using a more aggressive admission control policy for prefetched vectors.

In fact, the hit rate curves produced by the miniature caches not only provide the ideal threshold for prefetched vectors, but also allow the datacenter operator to optimize the amount of DRAM across the different tables to maximize performance. There are various techniques for maximizing total hit rate across multiple hit rate curves, including when the curves are convex (Cidon et al., 2015; 2017), and even when they are not convex (Beckmann & Sanchez, 2015; Cidon et al., 2016; Waldspurger et al., 2017). In our case, we found that the hit rate curves of all the tables are convex and do not change substantially across runs. Therefore, we ran Bandana on a trace with 5 billion requests and statically assigned the amount of DRAM to assign to each table with the goal of optimizing the total hit rate (Cidon et al., 2015).

5 END-TO-END EVALUATION

In this section we analyze the end-to-end effective bandwidth increase of Bandana under different scenarios: (1) as a function of cache size, (2) as a function of the simulated cache size, (3) as a function of SHP's training data size, and (4) as a function of the embedding vector size. In all the experiments in this section, unless otherwise specified, we ran Bandana using SHP trained on 5 billion requests, with a total cache of 4 million vectors, simulated caches with a size of 0.1% of the total cache, and a vector size of 128 B.

Figure 13 compares the total effective bandwidth increase across 8 different tables as a function of cache size. The

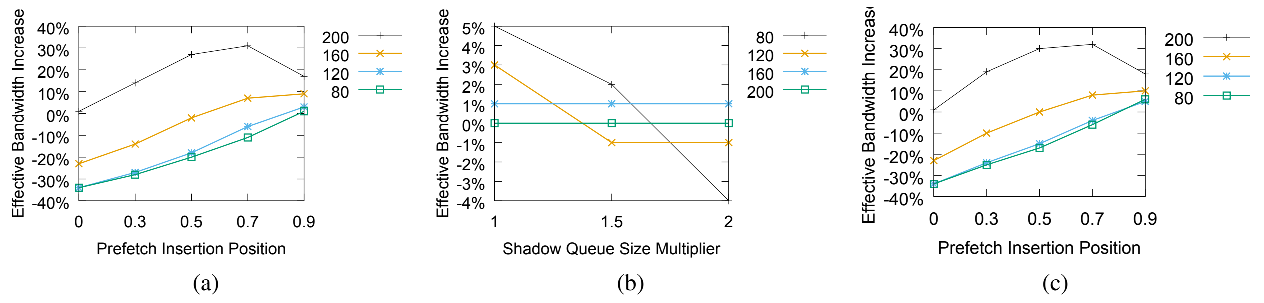


Figure 11. (a) Effective bandwidth increase when inserting prefetches to a different position in the queue, compared with no prefetching. (b) Effective bandwidth increase when filtering prefetches based on the shadow queue, compared with no prefetching. (c) Effective bandwidth increase when combining both methods, compared with no prefetching. The lines in all figures represent different cache sizes (vectors$\times 10^3$).

Table 2. Measuring the effectiveness of using miniature caches with different sampling rates with embedding table 2. On the left, the results show the ideal admission control threshold for the full cache, for different cache sizes. The results to the right show the chosen threshold when running miniature caches, using different sampling ratios. Even at 0.1% sampling, miniature caches provides a relatively similar bandwidth gain compared to the ideal threshold.

SIZE	FULL CACHE		10% SAMPLING		1% SAMPLING		0.1% SAMPLING	
	THRESHOLD	BW GAIN	THRESHOLD	BW GAIN	THRESHOLD	BW GAIN	THRESHOLD	BW GAIN
80,000	20	27.6%	20	27.6%	15	21.4%	15	21.4%
120,000	10	43.0%	15	36.3%	10	43.0%	15	36.3%
160,000	5	80.3%	5	80.3%	5	80.3%	10	61.0%
200,000	5	129.9%	5	129.9%	5	129.9%	5	129.9%

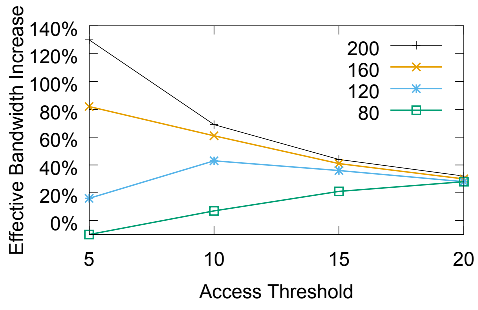


Figure 12. Effective bandwidth increase when filtering prefetched vectors based on the number of accesses during SHP run. The lines represent different cache sizes (vectors$\times 10^3$).

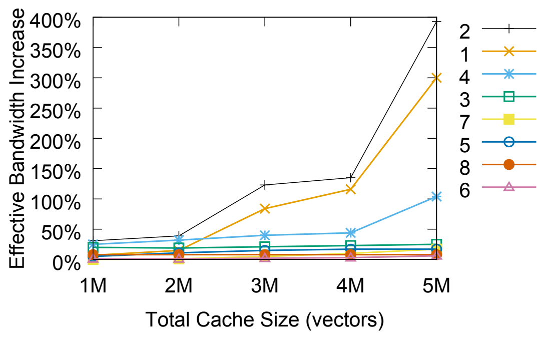


Figure 13. Effective bandwidth increase as a function of total cache size. The lines represent different embedding tables.

graph shows that for certain tables, the effective bandwidth significantly increases as a function of the cache size, up to almost 5× for table 2. For some tables however, the effective bandwidth remains relatively stable and low. The reason for this is that the access patterns of some tables are simply more random, and harder to effectively partition and cache. At the extreme, an application that accesses embedding vectors completely uniformly would not see any effective bandwidth increase.

We also analyze the impact of the size of the miniature caches in Figure 14. The figure shows that the effective bandwidth is almost the same with an oracle policy that selects the ideal prefetched vector threshold, compared to a miniature cache simulation that is scaled down to a thousandth of the size of the cache.

Figure 15 varies the number of training samples. It shows that as we increase the training time, the effective bandwidth increases. This is due to the fact that SHP's effectiveness in placing related vectors physically together improves as a function of the amount of training data.

While our model currently uses a vector size of 128 bytes, we also compare the effective bandwidth increase for different vector sizes in figure 16, using a total cache of 4 million vectors (i.e. the cache size changes proportionally with the vector size). When vector sizes are smaller, each NVM block accomodates more vectors, enabling Bandana

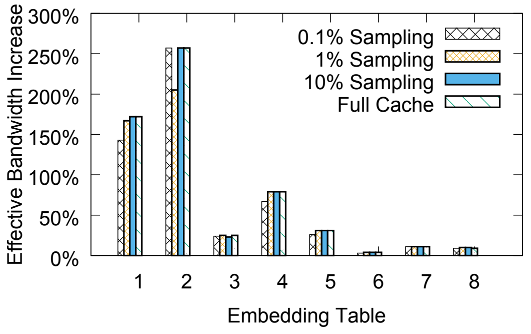


Figure 14. Effective bandwidth increase as a function of the sampling rate of the miniature caches, trained on 5 billion requests. The full cache policy represents an oracle policy that selects the ideal threshold for each table.

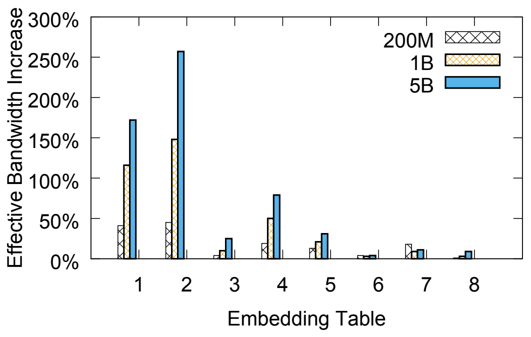


Figure 15. Effective bandwidth increase as a function of the number of requests used to train SHP, evaluated against 1 billion requests.

to achieve higher effective bandwidth increase.

6 Related Work

Bandana uses techniques inspired by prior research in using NVM as a substitute for DRAM, partitioning and caching.

NVM has been proposed as a low cost substitute for DRAM in other contexts, including databases and file systems. MyNVM (Eisenman et al., 2018) is a SQL database based on MyRocks, which uses block-level NVM as a second level cache for flash, and a lower cost substitute for DRAM. Similar to Bandana, one of the main challenges MyNVM deals with is NVM's limited bandwidth compared to DRAM. However, unlike Bandana, MyNVM stores objects in relatively large files (e.g., 4-6 KB). The novel challenge addressed by Bandana is how to physically place and cache embeddings vectors, which are much smaller than NVM blocks.

Other databases simulate NVM in its byte-addressable form, such as: CDDS (Venkataraman et al., 2011), Echo (Bailey et al., 2013), FPTree (Oukid et al., 2016), and HiKV (Xia et al., 2017). There has also been several prior projects in using NVM in its byte-addressable form for file systems, including: NOVA-Fortis (Xu & Swanson, 2016),

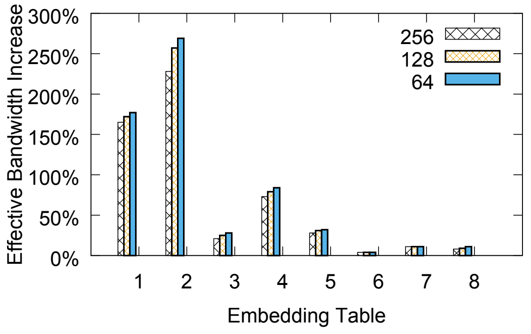


Figure 16. Effective bandwidth increase as a function of the embedding vector size (bytes).

LAWN (Wang & Chattopadhyay, 2018), and ByVFS (Wang et al., 2018). All of these systems use simulations to estimate how byte-addressable NVM will perform. Unfortunately since byte-addressable NVM is not commercially available, its real performance characteristics are unknown.

Bandana's mechanism for ordering vectors in physical blocks uses SHP, a hypergraph partitioning algorithm originally proposed for database query optimization (Shalita et al., 2016; Kabiljo et al., 2017). The reason we chose SHP is due to its scalability and ease of implementation. Another hypergraph partitioning algorithm used for database query partitioning is SWORD (Kumar et al., 2014). Zoltan (Devine et al., 2006) and Parkway (Trifunović & Knottenbelt, 2008) are other distributed hypergraph partitioning algorithms. However, both algorithms do not scale well for partitioning large workloads in our application (for more details see SHP (Kabiljo et al., 2017)).

Bandana uses micro-simulations to test different cache admission thresholds. A similar approach was used in recent work (Waldspurger et al., 2017) to approximate miss-rate curves of different caching algorithms, and select the optimal in real-time. Talus (Beckmann & Sanchez, 2015) and Cliffhanger (Cidon et al., 2016) demonstrate how miss-rate curves can be estimated by simulating a small cache. Other recent low cost miss-rate curve approximation techniques include Counter Stacks (Wires et al., 2014), SHARDS (Waldspurger et al., 2015), and AET (Hu et al., 2016).

7 Conclusions

Bandana is a novel NVM-based storage system for storing deep learning models. It provides a lower cost alternative to existing fully DRAM-based storage. Bandana reorders embedding vectors and stores related ones physically together for efficient prefetching, and dynamically adjusts its caching policy by simulating miniature caches for each embedding table. Similar techniques employed by Bandana can be extended for using NVM to store other types of datasets, which require granular access to data.

References

Flexible I/O tester. https://github.com/axboe/fio.

Arthur, D. and Vassilvitskii, S. K-means++: The advantages of careful seeding. *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1027–1035, 2007.

Bailey, K. A., Hornyack, P., Ceze, L., Gribble, S. D., and Levy, H. M. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pp. 4:1–4:8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2462-5. doi: 10.1145/2527792.2527799. URL http://doi.acm.org/10.1145/2527792.2527799.

Bary, E. DRAM supply/demand tightness should continue in second half of 2018, says Baird, 2018. https://www.marketwatch.com/story/dram-supplydemand-tightness-should-continue-in-second-half-of-2018-says-baird-2018-06-15.

Beckmann, N. and Sanchez, D. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 64–75, Feb 2015. doi: 10.1109/HPCA.2015.7056022.

Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X., and Shah, H. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, pp. 7–10, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4795-2. doi: 10.1145/2988450.2988454. URL http://doi.acm.org/10.1145/2988450.2988454.

Cidon, A., Eisenman, A., Alizadeh, M., and Katti, S. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. URL https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/cidon.

Cidon, A., Eisenman, A., Alizadeh, M., and Katti, S. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 379–392, Santa Clara, CA, March 2016. ISBN 978-1-931971-29-4. URL https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/cidon.

Cidon, A., Rushton, D., Rumble, S. M., and Stutsman, R. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 321–334, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/cidon.

Clark, J. Google turning its lucrative web search over to AI machines, 2015. https://www.bloomberg.com/news/articles/2015-10-26/google-turning-its-lucrative-web-search-over-to-ai-machines.

Covington, P., Adams, J., and Sargin, E. Deep neural networks for YouTube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA, 2016.

Devine, K. D., Boman, E. G., Heaphy, R. T., Bisseling, R. H., and Catalyurek, U. V. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pp. 124–124, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0054-6. URL http://dl.acm.org/citation.cfm?id=1898953.1899056.

Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K., Petersen, C., Cidon, A., and Katti, S. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pp. 42:1–42:13, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5584-1. doi: 10.1145/3190508.3190524. URL http://doi.acm.org/10.1145/3190508.3190524.

Hawkins, A. J. Inside Waymo's strategy to grow the best brains for self-driving cars, 2018. https://www.theverge.com/2018/5/9/17307156/google-waymo-driverless-cars-deep-learning-neural-net-interview.

Hu, X., Wang, X., Zhou, L., Luo, Y., Ding, C., and Wang, Z. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pp. 351–364, Denver, CO, 2016. USENIX Association. ISBN 978-1-931971-30-0. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/hu.

Kabiljo, I., Karrer, B., Pundir, M., Pupyrev, S., and Shalita, A. Social hash partitioner: A scalable distributed hypergraph partitioner. *Proc. VLDB Endow.*, 10(11):1418–1429, August 2017. ISSN 2150-8097.

doi: 10.14778/3137628.3137650. URL https://doi.org/10.14778/3137628.3137650.

Kumar, K. A., Quamar, A., Deshpande, A., and Khuller, S. SWORD: Workload-aware data placement and replica selection for cloud data management systems. *The VLDB Journal*, 23(6):845–870, December 2014. ISSN 1066-8888. doi: 10.1007/s00778-014-0362-1. URL http://dx.doi.org/10.1007/s00778-014-0362-1.

Lloyd, S. P. Least square quantization in pcm. *IEEE Trans. Information Theory*, 28:129–137, 1982.

Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

Mellor, C. Intel hands first Optane DIMM to Google, where it'll collect dust until a supporting CPU arrives, 2018. https://www.theregister.co.uk/2018/08/10/optane_dimm_ceremonially_ships_but_lacks_any_xeon_support/.

Oukid, I., Lasperas, J., Nica, A., Willhalm, T., and Lehner, W. Fptree: A hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pp. 371–386, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2915251. URL http://doi.acm.org/10.1145/2882903.2915251.

Shalita, A., Karrer, B., Kabiljo, I., Sharma, A., Presta, A., Adcock, A., Kllapi, H., and Stumm, M. Social hash: An assignment framework for optimizing distributed systems operations on social networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 455–468, Santa Clara, CA, 2016. USENIX Association. URL https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/shalita.

Trifunović, A. and Knottenbelt, W. J. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel Distrib. Comput.*, 68(5):563–581, May 2008. ISSN 0743-7315. doi: 10.1016/j.jpdc.2007.11.002. URL http://dx.doi.org/10.1016/j.jpdc.2007.11.002.

Venkataraman, S., Tolia, N., Ranganathan, P., and Campbell, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pp. 5–5, Berkeley, CA, USA, 2011. USENIX Association. ISBN 978-1-931971-82-9. URL http://dl.acm.org/citation.cfm?id=1960475.1960480.

Waldspurger, C., Saemundsson, T., Ahmad, I., and Park, N. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 487–498, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger.

Waldspurger, C. A., Park, N., Garthwaite, A., and Ahmad, I. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pp. 95–110, Santa Clara, CA, 2015. USENIX Association. ISBN 978-1-931971-201. URL https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger.

Wang, C. and Chattopadhyay, S. LAWN: Boosting the performance of NVMM file system through reducing write amplification. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pp. 6:1–6:6, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5700-5. doi: 10.1145/3195970.3196066. URL http://doi.acm.org/10.1145/3195970.3196066.

Wang, R., Fu, B., Fu, G., and Wang, M. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*, ADKDD'17, pp. 12:1–12:7, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5194-2. doi: 10.1145/3124749.3124754. URL http://doi.acm.org/10.1145/3124749.3124754.

Wang, Y., Jiang, D., and Xiong, J. Caching or not: Rethinking virtual file system for non-volatile main memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, 2018. USENIX Association. URL https://www.usenix.org/conference/hotstorage18/presentation/wang.

Wires, J., Ingram, S., Drudi, Z., Harvey, N. J. A., and Warfield, A. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 335–349, Broomfield, CO, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wires.

Wu, A. DRAM supply to remain tight with its annual bit growth for 2018 forecast at just 19.6%, according to Trendforce, 2018. https://www.dramexchange.com.

Xia, F., Jiang, D., Xiong, J., and Sun, N. HiKV: A hybrid index key-value store for DRAM-NVM memory systems.

In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 349–362, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia.

Xu, J. and Swanson, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 323–338, Santa Clara, CA, 2016. USENIX Association. ISBN 978-1-931971-28-7. URL https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

Zhou, G., Zhu, X., Song, C., Fan, Y., Zhu, H., Ma, X., Yan, Y., Jin, J., Li, H., and Gai, K. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pp. 1059–1068, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5552-0. doi: 10.1145/3219819.3219823.