Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache

Xingda Wei, Rong Chen, and Haibo Chen, *Shanghai Jiao Tong University*

https://www.usenix.org/conference/osdi20/presentation/wei

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache

Xingda Wei, Rong Chen, Haibo Chen
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

RDMA (Remote Direct Memory Access) has gained considerable interests in network-attached in-memory key-value stores. However, traversing the remote tree-based index in ordered stores with RDMA becomes a critical obstacle, causing an order-of-magnitude slowdown and limited scalability due to multiple roundtrips. Using index cache with conventional wisdom—caching partial data and traversing them locally—usually leads to limited effect because of unavoidable capacity misses, massive random accesses, and costly cache invalidations.

We argue that the machine learning (ML) model is a perfect cache structure for the tree-based index, termed *learned cache*. Based on it, we design and implement XStore, an RDMA-based ordered key-value store with a new hybrid architecture that retains a tree-based index at the server to perform dynamic workloads (e.g., inserts) and leverages a learned cache at the client to perform static workloads (e.g., gets and scans). The key idea is to decouple ML model retraining from index updating by maintaining a layer of indirection from logical to actual positions of key-value pairs. It allows a stale learned cache to continue predicting a correct position for a lookup key. XStore ensures correctness using a validation mechanism with a fallback path and further uses speculative execution to minimize the cost of cache misses. Evaluations with YCSB benchmarks and production workloads show that a single XStore server can achieve over 80 million read-only requests per second. This number outperforms state-of-the-art RDMA-based ordered key-value stores (namely, DrTM-Tree, Cell, and eRPC+Masstree) by up to 5.9× (from 3.7×). For workloads with inserts, XStore still provides up to 3.5× (from 2.7×) throughput speedup, achieving 53M reqs/s. The learned cache can also reduces client-side memory usage and further provides an efficient memory-performance tradeoff, e.g., saving 99% memory at the cost of 20% peak throughput.

1 Introduction

Network-attached in-memory key-value stores have become the foundation of many datacenter applications, including databases [47, 55], distributed file systems [7], web services [4, 37], and serverless computing [23, 42, 28], to name a few. With the prevalence of affordable high-performance networks in modern datacenters [46, 17, 20], such as InfiniBand, RoCE, or OmniPath, CPU quickly becomes the performance bottleneck and limits the scalability with the increase of clients [31]. RDMA (Remote Direct Memory Access) has recently generated considerable interests in optimizing network-attached in-memory key-value stores (aka RDMA-based KVs) in both academia [34, 25, 52] and industry [16, 55, 31], as it enables direct access to the memory of remote machines with low latency and CPU/kernel bypassing. However, leveraging RDMA to ordered key-value stores encounters a significant obstacle—traversing tree-based index with one-sided RDMA primitives is costly and complex (e.g., 11× slowdown in Fig. 2c). This is because it usually requires multiple network round trips (e.g., $O(\log N)$) and rapidly saturates bandwidth.

Many recent academic and industrial efforts [57, 17, 35] therefore proposed *index caching* to reduce RDMA operations. Yet, the conventional wisdom on implementing cache—replicating partial data and accessing them locally—does not work well with the tree-based index, and the drawbacks are amplified by maintaining the *tree-based* cache with RDMA primitives. First, the tree-based index can be large, so that the cache would suffer from unavoidable capacity misses. Second, the cache would aggravate random memory accesses and further increase the end-to-end latency. Third, updating the tree-based index may recursively invalidate the cache and cause false invalidation due to path sharing.

Inspired by recent research [29]—using machine learning (ML) models as an alternative index structure, we propose to leverage ML models as the (client-side) RDMA-based cache for the (server-side) tree-based index, termed *learned cache*. Specifically, the client uses learned cache to predict a small range of positions for a lookup key and then fetches them using one RDMA READ. After that, the client uses a local search (e.g., scanning) to find the actual position and fetches the value using another RDMA READ. Although using ML models as the index seems efficient (a few floating/int operations) and cheap (a small memory footprint) for static workloads (e.g., gets), it is also notoriously slow (frequently retraining ML models) and costly (keeping data in order) for dynamic workloads (e.g., inserts).

To address the above challenges, we propose a *hybrid* architecture that retains a tree-based index at the server to perform dynamic workloads (e.g., inserts) and leverages a learned cache at the client to perform static workloads (e.g., gets and scans). The hybrid architecture not only provides separate and appropriate execution paths for both workloads, but also simplifies the mechanism to guarantee the correctness of concurrent local and remote operations.

Based on this architecture, we further introduce a layer of indirection (i.e., a translation table) between the ML model and the tree-based index, which maps the logical position to the actual position of key-value pairs in the leaf-node granularity. The translation table decouples model retraining from index updating (e.g., node splits) and allows a *stale* learned cache (a combination of ML model and translation table) to continue predicting a *correct* position for a lookup key, as long as it is not overlapped with a leaf node split. It implies that the tree-based index can be concurrently updated in-place. Meanwhile, the ML model associated with its translation table can be retrained in the background and independently pulled by the clients on demand.

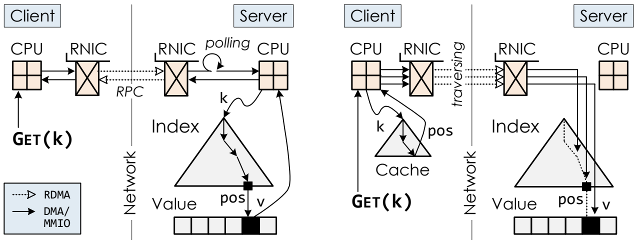
We have implemented XStore by extending a concurrent B+tree [50] with a well-tuned RDMA framework [51]. We evaluate XStore using the YCSB benchmarks [13] with two synthetic and one real-world [2] datasets, as well as two production workloads from Nutanix [30]. Our experimental results show that a single XStore server can achieve over 80 million read-only requests per second. This number outperforms state-of-the-art RDMA-based ordered key-value stores (i.e., DrTM-Tree [11], Cell [35], and eRPC+Masstree [24]) by up to 5.9× (from 3.7×). For workloads with inserts, XStore still provides up to 3.5× (from 2.7×) throughput speedup, achieving 53M reqs/s. The learned cache also reduces client-side memory usage significantly and further provides an efficient memory-performance tradeoff. For example, it can save 99% memory at the cost of 20% peak throughput, compared to caching the whole index.

In summary, this paper makes four contributions:

- The idea of *learned cache* that leverages machine learning (ML) models as index cache for RDMA-based, tree-backed KV stores;
- A hybrid architecture that combines (client-side) learned cache and (server-side) tree-based index to embrace static and dynamic workloads;
- A layer of indirection (translation table) that decouples ML model retraining from index updating and allows a *stale* learned cache to predict a *correct* position;
- A prototype implementation and an evaluation that demonstrates the advantage and efficacy of XStore.

2 RDMA-based Key-Value Store

In this paper, we focus on in-memory key-value (KV) stores that adopt the client-server model (network-attached) [34, 32, 25, 8] and range index structures (tree-backed) [33, 35, 57]. The server hosts both key-value pairs and indexes in main memory and handles requests from multiple clients concurrently. The client interacts with the server through a library that provides basic key-value interfaces, including Get(K), Update(K,V), Scan(K,N)[1], Insert(K,V), and Delete(K), as well as more complex operations built atop them.



(a) Server-centric RKV

(b) Client-direct RKV

Fig. 1. The architecture of RDMA-based key-value stores: (a) server-centric RKV and (b) client-direct RKV.

RDMA (Remote Direct Memory Access) is an emerging feature—appearing in affordable high-performance networks (e.g., InfiniBand, RoCE, or OmniPath)—that enables direct access to the memory of remote machines with low latency and CPU/kernel bypassing. It has generated considerable interest in deploying the network in modern datacenters [17, 46, 20] and optimizing key-value stores (aka RDMA-based KVs) [34, 25, 16, 9, 8]. However, few prior systems consider *ordered* key-value stores that rely on tree-based indexes to handle range queries (i.e., Scan(K,N)).

Server-centric design (S-RKV) [52, 26, 24]. An obvious design is to take a traditional KV store and reimplement the communication layer (e.g., RPC) using RDMA primitives. As shown in Fig. 1a, the clients ship their requests to the server via RDMA network using one round trip for each; the server traverses the tree-based index and performs the request locally. The *server-centric* design allows access to the server-side store with only two RDMA operations (one for sending and one for receiving), no matter how complex the index structures are, thereby avoiding multiple round trips and message size amplification [26]. However, this design exploits only high performance (low latency and high bandwidth) but not CPU efficiency (remote CPU bypassing) of RDMA network at the server, which limits the scalability of these KV stores with the increase of clients.

Client-direct design (C-RKV) [35, 17, 57]. The adoption of RDMA makes it practical to allow clients to access data hosted on the server directly, thereby permitting an alternative (*client-direct*) design that relaxes the burden on server CPUs. To simplify the mechanism for consistency, this design is restricted to read-only requests (i.e., Get and Scan) in most systems [34, 16, 35]. This common choice is also motivated by the read-dominated nature of most applications [6]. As shown in Fig. 1b, the clients use one-sided RDMA operations to traverse the tree-based index and fetch the value directly for read-only requests; the server still needs to perform the rest of requests (i.e., Update, Insert, and Delete) locally. The *client-direct* design can shift the CPU load on the server to the clients, which would alleviate the bottleneck (from CPU to network), especially on high-bandwidth networks (e.g., 100Gbps). However, it may consume extra network round trips for traversing the tree-based index due to the lack of richness of RDMA primitives, causing an order-

[1] Scan(k,n) provides a form of range query that retrieves first (up to) n key-value pairs, where their keys are larger than or equal to k.

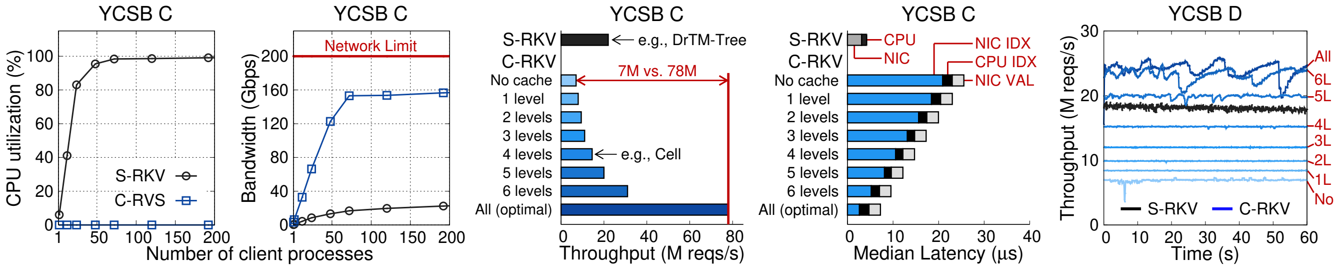


Fig. 2. A comparison of server-side (a) CPU and (b) network bandwidth utilization, (c) peak throughput, (d) end-to-end median latency at low load, and (e) throughput timeline for state-of-the-art server-centric (**S-RKV**) and client-direct (**C-RKV**) RDMA-based KV stores. **Workload**: YCSB C (100% read) and YCSB D (95% read and 5% insert), using 100M keys with a uniform distribution. **Testbed**: The server has two 12-core CPUs and two 100Gbps RNICs.

of-magnitude slowdown (e.g., 11× slowdown in Fig. 2c). For example, recent work [35, 57] uses RDMA READs to traverse remote B+tree index and invariably incurs multiple network round trips (O(logN) [3]).

Recently, *index caching* has been proposed to reduce network round trips for index traversal by RDMA-based systems [52, 48, 17, 35, 38], namely, the client caches the server-side index locally. It aims at reducing RDMA READs for fetching the position of the value (aka *lookup*), instead of caching the value directly.[2] Thus, an *optimal* result with index caching only needs two RDMA operations per request (one for lookup and one for read).

3 Analysis of RDMA-based Ordered KVs

CPU is the primary scalability bottleneck in the server-centric design. Fig. 2 compares hardware resource utilization between S-RKV and C-RKV with the increase of clients. For S-RKV, the server rapidly saturates all CPUs (24 cores) but just consumes 11% of network bandwidth. It implies that CPU first becomes the performance bottleneck and limits the scalability with the increase of clients, especially when deploying fast networks. This also runs counter to the recent trend of building servers in modern datacenters with CPU-bypassing networks [17, 46, 20]. As shown in Fig. 2c, S-RKV reaches the peak throughput of around 24M reqs/s. Traversing tree-based index occupies most of CPU time, as it involves massive random memory accesses. On our testbed, we measured that one CPU core can perform 43 million 64-byte random reads per second at full speed. Thus, each core can only process up to 1.8M reqs/s for traversing a (8-level) B+tree with 100M keys, even putting other CPU and network costs aside.

Costly RDMA-based traversal is the key obstacle in the client-direct design. C-RKV allows the client to traverse the server-side index directly by using one-sided RDMA READs, which can thoroughly bypass server CPUs (see Fig. 2a). However, RDMA-based index traversal usually requires multiple network round trips (e.g., O(logN) for tree-based index) and saturates the network bandwidth quickly. As shown in Fig. 2c, RDMA-based traversal limits the peak throughput of C-RKV to 7 million requests per second, even much lower than that of S-RKV. Using index caching at the clients can reduce RDMA operations by traversing index nodes locally. On our testbed, the throughput of C-RKV with index caching, similar to state-of-the-art design (Cell [35]), peaks at 14.5M reqs/s, as each request takes 4 RDMA READs (down from 8) for traversal.

Tree is not a proper structure for RDMA-based index cache. To our knowledge, existing RDMA-based index caches use *homogeneous structures* to store *partial* index nodes, similar to the conventional design. For example, each client replicates tree nodes and traverses them locally before accessing the tree-based index hosted on the server [17, 35, 57].

First, the tree-based index can be large [56, 18, 26], and the traversal demands multiple random accesses from the root to the leaf node. Thus, each client can only cache nodes near the root (e.g., top four levels [35]) to minimize thrashing and maximize hits [35, 17]. Yet, the index cache still suffers from *unavoidable capacity misses* (bottom node levels). In Fig. 2c, for a read-only workload, the effect of RDMA-based caching for tree-based index is dominated by inner node levels cached. The *optimal* throughput (a *whole-index* cache) reaches 78M reqs/s using one RDMA READ for each traversal (fetch the position of value), 3.3× better than S-RKV.

Second, traversing tree-based index is a memory-intensive but low-compute operation. The *homogeneous* index cache can just alter the type of memory accesses (i.e., remote and local), instead of reducing the number of memory accesses (O(logN)). Hence, despite the index cache, traversing tree-based index would still incur *massive random accesses* and suffer from CPU cache misses, TLB misses, and RNIC's page translation cache misses. As shown in Fig. 2d, even caching the whole index, the end-to-end latency of C-RKV is still 80% higher than S-RKV, and the CPU cost on index cache (CPU_IDX) occupies close to 30%.

Third, updates to the tree-based index (i.e., inserts and deletes) might propagate the changes from the leaf level to the root node, so that the index updates would probably invalidate the cache *recursively* [57] and cause *false* invalidations

[2] Considering RDMA performance degradation with increasing payload size [25], the client will only cache internal nodes [35, 38] and not directly fetch a batch of keys and (inline) values to avoid bandwidth amplification [35, 3].

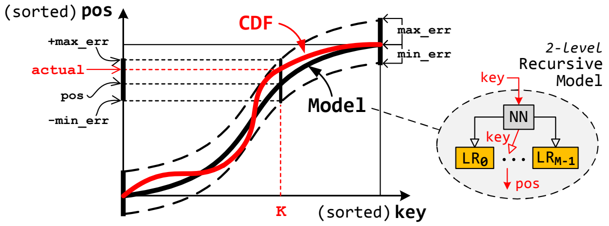


Fig. 3. An example of using ML models to predict the position within a sorted array for a given key.

(path sharing). It would result in frequent cache misses and RDMA READs to retrieve updated index nodes. Worse yet, the more tree nodes cached, the more performance degrades. Further, preserving traversal consistency for dynamic workloads demands sophisticated detection schemes (e.g., fence keys [19, 41]) and incurs additional overhead. In Fig. 2e, the *optimal* throughput significantly drops to 25M reqs/s with severe performance fluctuations, just because of 5% inserts.

4 Approach and Overview

Opportunity: ML Models. Our work is motivated by an attractive observation from the learned index [29]—a range index (e.g., B+tree) that finds the position of a given key inside a sorted array approximates the cumulative distribution function (CDF) of the keys in the index. As shown in Fig. 3, suppose the values have been sorted according to the lookup keys, the CDF (the red curve) is a mapping from the (sorted) keys to the (sorted) positions of their values, namely CDF(K) returns the *actual* position of the value corresponding to K. Prior work [29] proposes to approximate the shape of a CDF using machine learning (ML) models, like neural nets (NN) and linear regression (LR), since they are able to learn a wide variety of distributions. As an alternative range index, the ML model is trained with every key to record the worst over- and under-prediction of a position (i.e., min- and max-error). In Fig. 3, given a lookup key (K), the model (the black curve) can predict a position (pos) with a min- and max-error (min_err and max_err), and a local search (e.g., scanning) around the prediction is used to get the *actual* position. To further reduce the prediction error, a hierarchy of simple models (e.g., recursive-model index [29]) is used to partition the key space, where the model at level L picks the model at level L+1 based on the key.

Our approach: Learned Cache. The key idea behind XSTORE is to leverage machine learning (ML) models as (client-side) RDMA-based cache for the (server-side) tree-based index, termed "***learned cache***". The unique features of machine learning models can fundamentally overcome the drawbacks in the conventional wisdom for RDMA-based index caching (see §3). First, instead of using a *homogeneous* structure to cache a *partial* index, the ML model can cache the *whole* index at the cost of *accuracy*. Therefore, using the learned cache can completely avoid capacity misses, and each lookup only needs one RDMA READ. Further, the ML model is also famously memory-efficient (e.g., two parameters per LR model). Thus, the learned cache can match the *optimal* throughput of conventional design (a whole-index cache) but with practical memory consumption.

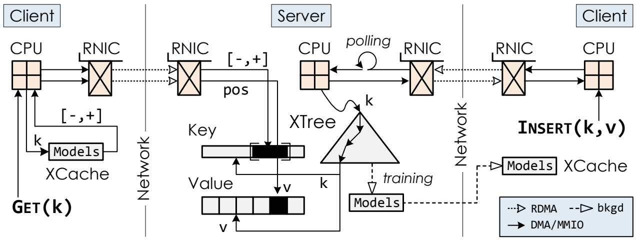


Fig. 4. The hybrid architecture behind XSTORE: client-direct operations (left) and server-centric operations (right).

Second, instead of finding the *actual* position by traversing a tree-based index with $O(\log N)$ random memory accesses, the ML model can approximately predict a *range* of positions for a lookup key by performing a single multiplication and addition (e.g., linear regression). It implies that the learned cache might also reduce the end-to-end latency, even compared to a whole-index cache, due to fewer CPU cache and TLB misses at the clients.

Finally, instead of *fine-grained* and *recursive* invalidation in the tree-based cache for accurate predictions, the ML model can reduce and delay cache invalidations since it only needs to provide approximate predictions. Updates to the index might only decrease the accuracy of the (partial) ML model. Thus, the learned cache can significantly save invalidation cost in terms of network round trips and bandwidth usage, especially compared to a whole-index cache.

Challenge: Dynamic Workloads. Dynamic workloads (e.g., inserts and deletes) would violate an (unrealistic) assumption of ML-based approach that all key-value pairs are stored in sorted order by key [29]. However, retraining ML models and keeping data in order are slow and costly, which is hard to match the high performance of in-memory key-value stores (tens of millions of requests per second). An intuitive solution is to maintain a delta index (e.g., B+tree) for (in-place or buffer-based) inserts and then periodically compact it with the learned index (data merging and model retraining) [44, 18]. Unfortunately, it cannot work well with RDMA-based index caching. First, additional RDMA-based lookups on the delta index would incur more network round trips and severely increase the latency. Second, it is also hard to cache a fast-changing (tree-based) delta index at the clients. Finally, the data and model compaction definitely interrupts (RDMA-based) remote accesses and completely invalidates the learned cache. Hence, *how to make learned cache keep pace with dynamic workloads at low cost* becomes a key challenge.

Overview of XStore. XSTORE is an in-memory ordered key-value store using a client-server model, where the server and the clients are connected with a high-speed, low-latency network with RDMA.[3] Using ML models as the index (aka *learned index*) is famously efficient and cheap for static workloads (e.g., gets and scans), while it is notoriously slow and costly for dynamic workloads (e.g., inserts and deletes). It is because the inserts would amplify the prediction error and incur model retraining frequently. Prior work [29, 44, 15] relies more on the profit from efficiently handling static workloads to amortize the negative influence on dynamic workloads. We argue that the *learned cache* opens the opportunity to solve this dilemma. Unlike prior work [29, 44, 15, 14], which replaces or augments the tree-based index with the learned index, we propose a *hybrid* architecture that *retains the tree-based index at the server to handle dynamic workloads and uses the learned cache at the clients to handle static workloads*.

The architecture of XSTORE is shown in Fig. 4. The server hosts a B+tree index (XTREE) in the main memory and stores key-value pairs at the leaf level physically, like the common practice. Each client interacts with the server through a library, which hosts a local learned cache (XCACHE). XSTORE uses the client-direct design for read-only requests (i.e., GET(K) and SCAN(K,N)) and the server-centric design for the rest (i.e., UPDATE(K,V), INSERT(K,V), and DELETE(K)). For client-direct operations, like GET(K) in Fig. 4, the client first predicts a range of positions for the key K using XCACHE and then fetches them using one RDMA READ. Finally, the client uses a local search to find the actual position and fetches the value using another RDMA READ. For server-centric operations, like INSERT(K,V) in Fig. 4, the client uses RPC over RDMA to ship the request to the server. The server searches the lookup key K by traversing the B+tree index first and then inserts the new KV pair (K,V). XSTORE will partially retrain ML models for updated tree nodes in the background, and each client will individually fetch the models for XCACHE on demand.

5 Design and Implementation

5.1 Data Structures

XTree. At the server, XSTORE retains a B+tree index (XTREE) and stores key-value pairs at the leaf level physically, like the common practice, as illustrated in the left part of Fig. 5. XTREE follows the basic design of a concurrent B+tree [33, 50], except that the leaf node (LN) adopts the structure optimized for remote reads. The leaf node consists of a 24-bit incarnation (INCA), an 8-bit counter (CNT), a 32-bit right-link pointer to next sibling (NXT), keys with N slots ($K_0 .. K_{N-1}$) and values with N slots ($V_0 .. V_{N-1}$).

Every leaf node is allocated from an RDMA-registered memory region using a slab allocator and can store at most N key-value pairs in sorted order. For brevity, we assume fixed-length key-value pairs here.[4] To save the size of RDMA READ for lookup, XSTORE stores keys and values separately but continuously. It can avoid storing the address of the value. The client can fetch N keys from the leaf node and calculate the (remote) address of expected value locally (a fixed offset from its key). Moreover, XSTORE uses incarnation checks [16, 52] to guarantee the consistency of remote accesses. The incarnation in the leaf node is initially zero and is monotonically increased when the leaf node is reused (e.g., split or free). The number of slots (N) can be tuned for RDMA performance (e.g., 16).

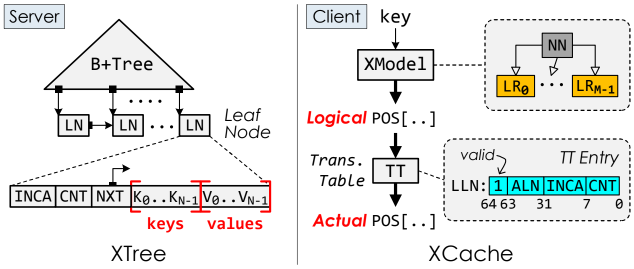


Fig. 5. The main structures in XSTORE: XTREE and XCACHE.

XCache. Each client hosts a local learned cache (XCACHE), which consists of a 2-level recursive ML model (XMODEL) and a translation table (TT). As illustrated in the right part of Fig. 5, given a lookup key, XMODEL is used to predict a range of positions (POS[..]) within a sorted array (logically stitching together all leaf nodes of XTREE). Currently, XMODEL uses a linear multi-variate regression model at level 0 (top-model) and simple linear regression models at level 1 (sub-model), a common setup recommended in prior work [15, 29, 44].

The ML model demands the positions (virtual address) of leaf nodes are always sorted by the keys. It is almost impossible for dynamic workloads, since the insertion of key-value pairs may insert a new node at the leaf level and break the sorted order of leaf nodes. The server maintains an additional translation table (TT) for leaf nodes, from logical to actual positions, and each client caches a part of the table on demand. The entry of TT is located by the logical leaf-node number (LLN) and consists of a valid bit, a 31-bit actual leaf-node number (ALN), a 24-bit expected incarnation (INCA), and an 8-bit counter, as shown in Fig. 5. The client can calculate the (host) virtual address of the target leaf node using ALN and the base address of an RDMA-registered memory region. Further, the match of incarnation between TT's entry and target leaf node guarantees that the leaf node has not been reused.

Training models and TT. The server (re-)trains a 2-level ML model (XMODEL) with a translation table (TT) over XTREE's leaf nodes in the background, and each client (re-)fills the learned cache (XCACHE) on demand. Fig. 6 shows

[3] The client may not be the end user but the computation node or the front-end of RDMA-based datacenter applications [34, 35, 16, 17, 25, 55, 57].

[4] Similar to prior RDMA-enabled KVS [16, 52, 35], XSTORE currently allows fixed-length key and fixed/variable-length value. For variable-length value, the leaf node should store a 64-bit fat pointer [16, 53] (the size and the position of value) instead of the value. We discuss how to support variable-length key in §6 and leave it to future work.

```
▸ M: Max. number of sub-models
▸ N: Max. number of keys in each leaf node
XModel {
  Model     top                  ▸ LR: k → [0,1)
  Model[M] subs   ▸ LR: k → [0,pos) w/ min/max_err
}
TRAIN_XMODEL(xmodel)
▸ train top-model
1   cdf = []                         ▸ training set
2   pos = 0
3   foreach k in xtree             ▸ in sorted order
4   | cdf.add(k, pos++)
5   xmodel.top = new LR trained on cdf
▸ assign keys to sub-models
6   kset = [][]       ▸ key set for each sub-model
7   foreach k in xtree
8   | mid = xmodel.top.predict(k) × M
9   | kset[mid].add(k)
▸ train sub-models
10  for i in [0:M)
11  | TRAIN_SUBMODEL(xmodel.subs[i]
    |                MIN(kset[i]), MAX(kset[i]))
TRAIN_SUBMODEL(model, min, max)
12  cdf = []                         ▸ training set
13  LLN = 0                ▸ logic leaf-node number
14  start = xtree.find_lnode(min)
15  end = xtree.find_lnode(max)
16  for lnode in [start:end]
17  | pos = LLN × N
18  | foreach k in lnode.keys   ▸ key-sorted order
19  | | cdf.add(k, pos++)
20  | model.tt[LLN++] = {1, ALN(lnode),
    |                     lnode.inca, lnode.cnt}
21  model = new LR trained on cdf
22  model.calc_err(cdf)    ▸ calculate min/max_err
```

Fig. 6. Pseudo-code of training XMODEL and TT over XTREE.

the pseudo-code of training a complete XMODEL and TT. Starting from a sorted array of keys with logical positions (line 4), we first train the top model. Based on the prediction of the top model, we then evenly partition keys into M sub-models (line 9). Finally, we train each sub-model on a sorted array of its keys with a private logical position at a leaf node granularity (line 12-21) and calculate min- and max-error for every sub-model (line 22). Note that the keys in the leaf node across sub-models will be trained by both of sub-models. Moreover, each sub-model has independent logical positions and an own translation table, making it easy to retrain a sub-model individually when necessary.

In practice, training XMODEL is fast and low-cost, since (1) all of the models in XMODEL are simple linear/multivariate regression models, can be efficiently trained; (2) XMODEL can be partially retrained at a sub-model granularity; and (3) the top model can be trained over a sampled data. As an example, for 100M keys, XMODEL with 500K sub-models takes about 4 seconds to train the top-model and 8 microseconds for each sub-model using a single thread. Further, the client can fill a 500K sub-models XCACHE from scratch in less than one second.

```
LOOKUP(key, &addr)
1   mid = xmodel.top.predict(key) x M
2   model = xmodel.subs[mid]
3   pos = model.predict(key)          ▸ prediction
4   start = (pos - model.min_err)/N    ▸ lnode ID
5   end = (pos + model.max_err)/N      ▸ lnode ID
6   rdma_doorbell = []
7   for n in [start:end]          ▸ from LLN to ALN
8   | entry = model.tt[n]              ▸ TT entry
9   | if entry.valid == 0 then
10  | | return invalid                 ▸ fallback
11  | ra = RA(entry.ALN)          ▸ remote address
12  | rdma_doorbell.add(ra)
  ▸ one RDMA to read disjoint memory regions
13  lnodes = RDMA_READ(rdma_doorbell)
14  for n in [start:end]
15  | lnode = lnodes[n-start]
16  | entry = model.tt[n]
17  | if entry.inca != lnode.inca then
18  | | entry.valid = 0            ▸ invalidation
19  | | return invalid                 ▸ fallback
20  | for i in [0:lnode.cnt)       ▸ local search
21  | | if key == lnode.keys[i] then
22  | | | addr = calc remote addr of ith value
23  | | | return found
24  return not_found              ▸ non-existent key
```

Fig. 7. Pseudo-code of LOOKUP operation based on XCACHE.

A memory-performance trade-off. The ML model is famously memory-efficient. In XMODEL, the basic sub-models are 14B large and consist of two 32-bit floating-point model parameters [5], two 8-bit min- and max-error, and a 32-bit TT size. Thus, XMODEL with 500K sub-models only needs less than 6.7MB. In contrast, TT might dominate the memory usage of XCACHE. For 100M keys, suppose each leaf node has 16 slots (N) and is half-full, TT requires nearly 100MB (15% of the tree-based index). In practice, each client could cache sub-models and TT entries on demand, and even just cache XMODEL to save 99% memory at the cost of 20% performance (using one RDMA READ to fetch a few TT entries).

5.2 Client-direct Operations

In the left part of Fig. 4, XSTORE uses the client-direct design for read requests, namely GET(K) and SCAN(K,N).

5.2.1 GET

Given a key, the client uses XCACHE to lookup the remote position of value using one RDMA READ commonly, replacing RDMA-based traversal in a tree-based index. As shown in Fig. 7, the client first uses XMODEL to predict leaf nodes that cover the lookup key (from `start` to `end`) and then calculates the actual (remote) address of these leaf nodes with TT (line 11). The client can use one RDMA READ with doorbell batching to fetch disjoint memory regions if necessary (line 13).[6] Note that the unit of remote

[5] LR may use more floating-points for prediction.

[6] One RDMA READ can only read a continuous memory region. Yet, we can use an RDMA-aware optimization called doorbell batching [27] to read

read is a leaf node (N keys with a 64-bit header); it is the most likely to read just one leaf node due to the low prediction error of XMODEL. Next, the client uses a local search (e.g., scanning) to find the key from leaf nodes retrieved (line 20-23) and calculates the remote address of the value if it is found (line 22). Finally, the client uses another RDMA READ to fetch the value. Note that any invalid TT entry (line 9 and 17) would result in a fallback path, which ships the GET operation to the server and fetches updated models and TT entries using a single request (i.e., server-centric design).

5.2.2 SCAN

SCAN(K,N) implements a form of range query that returns first (up to) N key-value pairs (in order by key), starting with the next key at or after K. The client first uses the lookup operation with K to determine the remote address of the first key-value pair (larger than or equal to K) and then predicts the leaf nodes that contain the next N key-value pairs, with the help of TT. The translation table provides the number of key-value pairs (CNT) and the actual remote address (ALN) of adjacent leaf nodes (LLN) in sorted order by key. Thus, the client can use one RDMA READ with doorbell batching to fetch these leaf nodes, including keys and values. In general, XSTORE only requires two RDMA READs for each range query. In the rare case, the unexpected result, such as an invalid leaf node (incarnation mismatch) due to dynamic workloads, would cause a fallback path, similar to GET. Note that the range query in XSTORE is also not atomic with respect to updates and inserts as usual [33, 35]; it could be implemented by applications (e.g., transaction [17, 38]).

5.2.3 Non-existent Keys

Intuitively, the ML model guarantees to find all keys have been trained since it stores the worst over- and under-prediction for a CDF (i.e., min- and max-error). However, for non-existent keys, the model should be *monotonic* to guarantee the correct upper and lower bound of a prediction [21, 54], so that a local search could make sure the lookup key does not exist (see line 24 in Fig. 7). Hence, XMODEL adopts monotonic models (e.g., linear regression). As shown in Fig. 8, for a non-existent key (KEY=6), the sub-model LR0 can provide a proper prediction (LR0(6)=[3,4]) that covers the non-existent key (KEYS={5,7}).

However, a hierarchy of models might leave a gap of non-existent keys between neighboring models. Consequently, it still might provide a wrong prediction for these non-existent keys, even if every model is monotonic. For example, the top model selects LR0 for KEY=10 (non-existent), and then LR0 will return a wrong prediction (LR0(10)=[6,7]) that cannot determine whether the key does not exist or the model is out of date from the results (KEYS={17,18}). Worse yet, the non-existent key is common in the range query (e.g., SCAN(K,N)), which demands to retrieve first (up to) N keys larger than or equal to K. As illustrated in Fig. 8, the lookup (LR0(10)) for a range query SCAN(10,3) will miss a key (KEY=13), so the result (KEYS={17,18,20}) is also wrong.

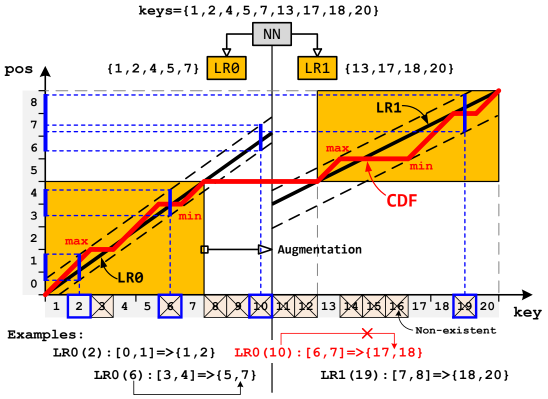


Fig. 8. An example of the prediction for non-existent keys.

Data augmentation. To remedy this, we augment the training set of sub-models to cover the gap of non-existent keys between neighboring models. However, data augmentation would increase the prediction error. We thus carefully add a boundary key to both sub-models, which can fill the gaps with minimal overlap between models. For example, in Fig. 8, we add a non-existent key in the gap (KEY=10) with the position of a previous KEY=4 into both sub-models (LR0 and LR1). After that, the lookup of non-existent keys would always return a correct prediction. Further, since the keys in the leaf node across sub-models have been trained by both, there is no need for data augmentation in most cases.

5.3 Server-centric Operations

As shown in the right part of Fig. 4, clients communicate with the server to perform UPDATE(K,V), INSERT(K,V), and DELETE(K) operations; the server updates XTREE concurrently and retrains XMODEL in the background.

Correctness. The correctness condition in XSTORE follows *no lost keys* [33]: the reader must return a correct value for a given key, regardless of concurrent writers. More specifically, when a reader and a writer run concurrently, the reader can return either the old or the new value, while both of them should be atomic.

Concurrency. The hybrid architecture behind XSTORE not only provides separate and appropriate execution paths for static and dynamic workloads (see Fig. 4), but also simplifies the mechanism to guarantee the correctness of concurrent operations. It is critical to the performance of RDMA-based systems due to the lack of richness of RDMA primitives [51]. In Fig. 9a, by using the learned cache (XCACHE), XSTORE restricts (client-direct) remote accesses to the leaf nodes (the dotted red arrow). Thus, we can avoid using sophisticated mechanisms to retrofit a concurrent tree-based index [35].

XTREE reuses an HTM-based concurrent B+tree [50][7] to support concurrent index updates (e.g., node splits) and

multiple disjoint memory regions in one network roundtrip.

[7] The implementation is based on Intel's restricted transactional memory (RTM) that is available as a mature feature in Intel's CPUs (e.g., Skylake).

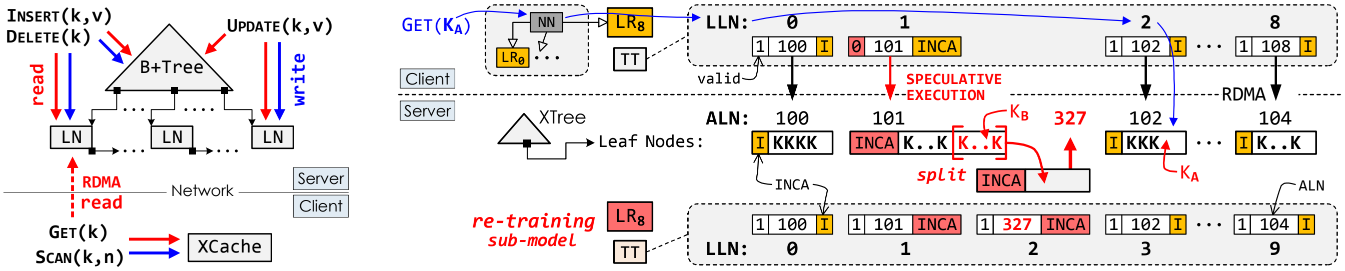


Fig. 9. (a) The access types of different operations for the main components in XStore. Red and blue arrows denote read and write accesses. (b) An example of model retraining for LR_8 due to a split of LN_{101}. The leaf node is named by its actual leaf-node number (ALN).

lookups on internal nodes, without the concern of RDMA-based remote accesses. For leaf nodes, XStore follows the technique proposed in DrTM+R [11]. Each tree operation at the server is enclosed within an HTM region, that provides strong atomicity in a single machine [5]. In addition, the strong consistency feature of RDMA (where an RDMA operation will abort an HTM transaction that accesses the same memory location [52]) further extends the atomicity when encountering remote accesses. Moreover, as the RDMA operation is only cache-coherent within a cache line, XStore adopts versioning [16] for consistent remote reads across multiple cache lines. For the data stored in the leaf node across multiple cache lines, a 16-bit version number is stored both in the header of data and at the start of each cache line. The remote reader matches these versions to detect inconsistent read and must retry if the versions differ. Note that XStore hides these versions to applications by automatically converting the data on reads and writes. Finally, the key is also stored in the header of its value, which guarantees consistent remote reads to the key and the value separately.

5.3.1 Update

For Update(K,V), the server first traverses XTree to the leaf node and updates the value with V if the key (K) exists. Note that the update to the value will not change the index, so that it will also not influence the learned cache and belongs to static workloads.

Optimization: position hint. Although Update(K,V) is a server-side operation, it can still benefit from the learned cache, especially when the server CPU becomes a bottleneck. The client could use XCache to predict a position (the remote address of leaf nodes) for the key (see line 1-12 in Fig. 7) and then ship the update request together with the position hint to the server. The server first checks the leaf nodes (by matching incarnation) according to the hint and updates the value if successful. It might skip index traversal and relax the burden on server CPUs. The optimization would increase the performance of update-heavy workloads, like YCSB A (50% update and 50% read).

5.3.2 Insert and Delete

Insert(K,V) and Delete(K) are shipped to the server and performed on XTree, as is usual on B+tree. The *in-place* inserts and deletes require moving many key-value pairs within a leaf node to preserve the order of keys. Thus, XTree chooses not to keep key-value pairs sorted within a leaf node, which can avoid moving key-value pairs and reduces working set in the HTM region. Note that the lookup based on the learned cache will not be affected since it fetches all keys (N) of a leaf node. For Delete(K), we always overwrite the key and value slot for K with the last key-value pair in the leaf node and update the counter (CNT). Further, the empty leaf node will not be reclaimed to avoid thrashing and model retraining. For Insert(K,V), we directly append K and V to the key and value slots in the leaf node if K does not exist (see K_A in Fig. 9b). Inserting a key-value pair into a full leaf node will result in a node *split* (see K_B in Fig. 9b). A new leaf node is allocated, and all key-value pairs (plus the new one) are evenly assigned to two leaf nodes in sorted order by key. The original leaf node should increment its incarnation, which makes the clients realize the split. The rest of the split process will execute on the tree index as well as usual.

Retraining and invalidation. The insert of a new leaf node (aka a split) will break the sorted (logical) order of leaf nodes and cause model retraining. An interesting observation behind our solution is that TT decouples model retraining from index updating and allows a *stale* combination of XModel and TT to provide a *correct* prediction for the lookup key as long as it is not overlapped with a split. This is because any insert will not cause data movement across leaf nodes, except the split node. For example, LR_8 initially maps K_A to logical node number LN_2, which stores the leaf's physical address 102. After leaf node LN_1 splits due to inserts (a new leaf node with physical address 327), the latest logical node number for K_A is LN_3 after retraining. Yet, the stale TT still maps K_A to physical address 102, the correct position of K_A. Thus, the client can still use a combination of stale models and TTs to find the keys as long as they are not overlapped with split leaf nodes.

Based on this, after a split, the server will *individually* retrain the sub-model and its translation table in the background (see Train_SubModel in Fig. 6) and perform all kinds of operations as usual based on XTree. Meanwhile, the clients can still directly perform read-only operations based on XCache. The incorrect prediction can be detected by incarnation mismatch between the leaf node and cached TT entry (line 17 in Fig. 7) and results in a fallback, which ships the operation to the server. The client will update XCACHE with a retrained model and its translation table fetched by the fallback. Noted that concurrent splits will not affect model retraining in progress and just make it stale. The new incarnation of the split leaf node ensures the client with this new (stale) model to realize the change of concurrent splits. Each split will issue a retraining task. The training thread currently does not merge or optimize the pending tasks to the same sub-model since it happens very rarely.

Optimization: speculative execution. A split of leaf node just moves the second half of key-value pairs (sorted by key) to its new sibling leaf node. Therefore, the prediction to the split node must still be mapped to this node or its new sibling, like LN_{101} and LN_{327} in Fig. 9b. Based on this observation, *speculative execution* is enabled to handle the lookup operation on a *stale* TT entry (i.e., incarnation check is failed). The client will still find the lookup key in the keys fetched from the split leaf node. If not found, the client will use its right-link pointer to fetch (the second half) keys from its sibling (one more RDMA READ). It means there is roughly half of the chance to avoid incurring a performance penalty. Currently, we only consider one sibling before using a fallback since a cascading split happens rarely. This optimization is important for insert-dominate workloads (e.g., YCSB D) since insert operations and retraining tasks might keep server CPUs busy; the fallbacks will also take server CPU time.

Model expansion. The growing size of key-value pairs in the ML model will likely increase the prediction error, resulting in performance degradation. Prior work [44] uses a sophisticated model split to adapt its learned structure for dynamic workloads, which demands physical data moving and atomic top-model replacement. In response to this problem, XSTORE supports *model expansion* that increases the number of sub-models in XMODEL at once (e.g., doubling) when necessary (e.g., exceeding a threshold of min- and max-error). The model expansion requires a complete training (see Fig. 6) on XTREE to build a new version of XMODEL and TT. Note that model expansion will not affect any requests performed by both the server and the client for several reasons. First, training models will not change or move data. Second, the top model can be trained over incomplete data. Third, the conflicting sub-model retraining could be made up later. Finally, the client can use the originally learned cache during model expansion. Moreover, after deleting a large number of key-value pairs, XSTORE can also resize XMODEL to shrink the number of sub-models using a similar process.

5.4 Durability

XSTORE should log writes (updates, inserts, and deletes) to log files stored in reliable storage for persistence and failure recovery (e.g., server's local disk). As RDMA-based remote accesses are restricted to reads (lookups, gets, and scans), they will not involve in logging and recovery. In addition, XMODEL and TT are tightly associated with XTREE (e.g., virtual address). Thus, they should be rebuilt after recovery.

To ensure correct recovery from a machine failure, XSTORE can reuse the existing durability mechanism in the concurrent tree-based index extended by XTREE, like version numbers [50, 33]. Each worker thread at the server appends the log (key, value, and version) to its in-memory log buffer. A corresponding logging thread, sharing the same core with the worker thread, writes out the log buffer to its log file in the background. The logger batches the log entries to avoid the storage backend becoming the bottleneck. During recovery, XSTORE scans log files to sort logs of the same key by its version number and applies the latest log of keys in parallel. Finally, XSTORE rebuilds XMODEL and TT by training over recovered XTREE.

5.5 Scaling out XSTORE

XSTORE follows a coarse-grained scheme [57], the dominant solution, to distribute an ordered key-value store span multiple servers (scale-out). XSTORE first assigns key-value pairs to the servers based on a range-based partitioning function for the keys. Then each server constructs XTREE individually for its assigned key-value pairs and further trains a corresponding XMODEL and TT. Note that the boundary keys should be added to the training set to cover the gap of non-existent keys between neighboring servers.

The client maintains a separate learned cache for each server and uses the same partitioning function to decide which server should perform a given request. Based on it, the client can perform requests as mentioned in §5.2 and §5.3, with one exception—SCAN(K,N) reads a range of key-value pairs span multiple servers. After the lookup of K on a specified server, the client might find that the expected number (N) exceeds the remaining key-value pairs in this server. Starting from the first logical leaf node on the next server, the client can predict the leaf nodes that contain the rest of key-value pairs. Finally, the client uses one RDMA READ for each server involved to fetch these leaf nodes.

6 Discussion

Support variable-length keys. XSTORE currently supports fixed-length key and variable/fixed-length value. To support variable-length key, XSTORE should store a fat pointer in the leaf node of XTREE (instead of the actual key), which encodes the size and position of the key. This scheme can traverse variable-length key locally by CPUs (i.e., server-centric design), while it would be hard to do it efficiently by using one-sided RDMA READs (i.e., client-direct design). XSTORE has to retrieve the actual keys using an additional RDMA READ for each (Line 21 in Fig. 7). Therefore, XSTORE further stores a fixed hash code of the key within the fat pointer. Consequently, the client could directly compare the hash codes instead of keys, after fetching the leaf node for a given key. Note that the actual (variable-length) key should

Table 1: YCSB workload description. **R**, **U**, **I**, **M**, and **S** denote read, update, insert, read-modify-update, and scan, respectively. Scan accesses N values, where N is uniformly distributed in [1,100].

YCSB	A	B	C	D	E	F
Type	R : U	R : U	R	R : I	S : I	R : M
Ratio (%)	50 : 50	90 : 10	100	95 : 5	95 : 5	50 : 50

be checked to avoid a hash collision. For example, the client can fetch the value associated with the key. We plan to extend XStore to support variable-length keys in future work.

Data distribution. XStore assumes machine learning (ML) models can effectively learn various data distributions (e.g., log-normal [29, 44, 15]). Based on it, we believe there is a trade-off among the memory consumption of XCache, the retraining costs of XModel, and the performance of XStore. When using simple models (e.g., linear regression) for fast model retraining, XStore has to use many models to achieve high accuracy for irregular data distributions. For such a scenario, clients can only cache partial sub-models due to the increased model memory consumptions. On the other hand, XStore could use complex models (e.g., neural network (NN)) to achieve high accuracy with few models. Yet, NN is slow on model retraining and may impact the performance under dynamic workloads (e.g., inserts), since the client may fall back more often due to stale XCache.

7 Evaluation

7.1 Experimental Setup

Testbed. Without explicit mention, we use one server machine and (up to) 15 client machines. Each machine has two 12-core Intel Xeon CPUs, 128GB of RAM, and two ConnectX-4 100Gbps IB RNICs. Each RNIC is used by threads on the same socket and connected to a Mellanox 100Gbps IB Switch. The server registers the memory with huge pages to reduce RNIC's page translation cache misses [16].

Workloads. We use YCSB [13] and two production workloads from Nutanix [30]. We mainly focus on YCSB as it contains various types of workloads [12]: update heavy (A), read mostly (B), read only (C), read latest (D), short ranges (E), and read-modify-write (F). Table 1 shows a summary of YCSB workloads (A-F). Since small requests dominate in real-life workloads [4], we evaluate KV stores with 100 million KV pairs initially (a 7-level tree-based index and a leaf level), where 8-byte key and 8-byte value are used, similar to prior work [33, 35, 24, 44]. Both Uniform and Zipfian key distributions are evaluated for all YCSB workloads. Note that YCSB D only has Uniform and Latest key distributions; the client is likely to query its recently inserted keys in Latest distribution. In addition, each client generates their insert key uniformly and randomly in YCSB D and E. The two production workloads both have a profile of 57:41:2 write:read:scan ratio, while the access patterns of them are relatively uniform (Prod1) and skewed (Prod2), respectively. Both of them have 500 million KV pairs with 8-byte key and 64-byte value. Finally, besides the default data distribution of the above workloads, we also use two synthetic and one real-life datasets (see Table 2) to study the behavior of learned cache in depth.

Table 2: Data distribution description for evaluating datasets.

Name	Description	Workloads
L	Linear	YCSB[13], Nutanix[30]
NL	Noised linear	YCSB[13]
OSM	Longitude location	Open Street Map[2]

Comparing targets. We compare XStore to three state-of-the-art RDMA-based ordered KV stores: DrTM-Tree [11] and eRPC+Masstree [24] (*server-centric* design), as well as Cell [35] (*client-direct* design). eRPC+Masstree (EMT) adopts eRPC [24] (RDMA-based RPC library) to extend Masstree [33] (in-memory ordered KV store). We implement DrTM-Tree and Cell in the same framework to provide an apple-to-apple comparison with two typical designs, but also because DrTM-Tree uses similar B+tree [50] and RDMA library [51] with XStore, and Cell is not open-source.[8] We further consider RDMA-Memcached v0.9.6 [22] (RMC) in our experiments, which is an RDMA version of memcached [1], a widely used network-attached KV in industry.

All systems fully utilize all of the 24 CPU cores (with hyperthreading disabled) and two RNICs. As EMT and RMC cannot use multiple NICs simultaneously, we deploy two instances at the server on different sockets, and each instance uses the RNIC attached to that socket. This actually makes them faster during experiments since it avoids cross-socket synchronizations. XStore uses (up to) two auxiliary threads to train ML models in the background for dynamic workloads. XTree is configured with a fanout of 16. XModel uses 500K sub-models for static workloads and 2M models for dynamic workloads to avoid model expansion during evaluation (because XStore can insert more than 150M KV pairs in 60s). In addition, logging is disabled in all systems, and the server hosts all data in main memory.

7.2 YCSB Performance

Fig. 10 compares the peak throughput of various RDMA-based key-value stores for YCSB with Uniform and Zipfian distributions, where all systems are saturated by up to 15 client machines. Note that RMC performs poorly in all experiments as it is bottlenecked by CPU synchronizations [43, 31]. Due to space limitations, we skip detailed discussion of experimental results on it.

Read-only workload (YCSB C). For Uniform distribution, XStore can achieve 82 million requests per second, even a little higher than the *optimal* throughput (a whole-index

[8] For DrTM-Tree, our experimental results were confirmed by the authors. For Cell, we follow the same caching strategy—the client caches nodes at least four levels above the leaf node at the clients with LRU policy to minimize churn and maximize hits. Based on a comparison against published numbers, we believe that the large performance difference between XStore and other systems (e.g., 27M reqs/s from our implementation vs. 0.95M reqs/s from Cell [35] for YCSB A with Zipfian distribution) offsets performance variations due to system and implementation details.

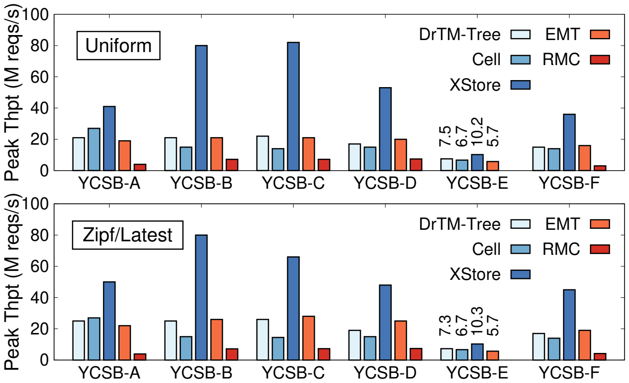


Fig. 10. Comparison of throughput on various RDMA-based KVs using YCSB. Note that RMC does not support range queries.

cache), since it only uses one RDMA READ to fetch one leaf node per lookup; the payload is 16B smaller by avoiding a sophisticated mechanism for consistency (i.e., min-max fence keys [35]). The prediction error of XCACHE is just 0.74. This number outperforms EMT, DrTM-Tree, and Cell by 3.9×, 3.7×, and 5.9×, respectively. Both DrTM-Tree and EMT are bottlenecked by server CPUs, while Cell is bottlenecked by RDMA amplifications; it still needs four RDMA READs to traverse tree nodes even index caching is enabled.

For Zipfian distribution, XSTORE can still outperform EMT, DrTM-Tree, and Cell by 2.4×, 2.5×, and 4.6×, respectively. The systems with server-centric design perform better due to better CPU cache locality. However, the peak throughput of XSTORE drops by 18% since RDMA has relatively poor performance when massive clients read a small range of memory simultaneously. We suspect that our current RNIC (ConnectX-4) checks conflicts between one-sided RDMA operations based on request's address [27], so that these operations may compete for NIC's internal processing resources, even if there is no conflict.

Static read-write workloads (YCSB A, B, and F). For update-heavy workloads (YCSB A), XSTORE is still bottlenecked by server CPUs for handling updates. However, compared to server-centric KVs (e.g., DrTM-Tree and EMT), the clients in XSTORE can directly perform read requests with the help of learned cache, which completely bypasses server CPUs. Therefore, XSTORE can still provide up to 2.2× and 2.3× (from 1.5× and 2.0×) throughput improvements for Uniform and Zipfian distributions, respectively, compared to other KVs. For read-mostly workloads (YCSB B), the speedup of throughput in XSTORE further reaches up to 5.3× (from 3.1×). There are two reasons: (1) the read requests are less skewed interleaved with (10%) updates, compared to read-only workloads (YCSB C); (2) the server of XSTORE has not been saturated (less than 40% of CPU utilizations); thus it is still sufficient to perform updates, compared to update-heavy workloads (YCSB A). The performance of XSTORE on YCSB F is somewhere in between since it has about 75% reads.

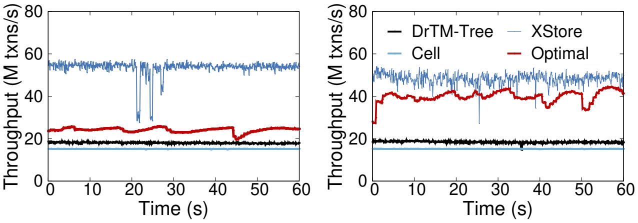


Fig. 11. The performance timeline of YCSB D with (a) Uniform and (b) Latest workloads.

Dynamic workloads (YCSB D and E). The throughput of every system is impacted by dynamic workloads due to the contention between reads and inserts. For DrTM-Tree and EMT, the contention happens on the tree-based index. For XSTORE and Cell, the performance slowdown is mainly due to cache invalidations. However, Cell only caches the top four levels, where node split is rare. The overhead in XSTORE mainly comes from two parts: (1) cache invalidations would increase RDMA operations due to fallbacks (RDMA-based RPC) and speculative execution (50% one more RDMA READ); (2) a dynamic dataset is always harder to learn than a static dataset due to the randomly inserted new keys; the prediction error would stably increase to 8.3 for YCSB D.[9] Fortunately, the clients can still use stale learned cache for most read requests, and model retraining is also very fast. Thus, for YCSB D, XSTORE can provide up to 3.5× and 3.2× (from 2.7× and 1.9×) speedup and achieve 53M and 48M reqs/s throughput for Uniform and Latest distributions, respectively. For YCSB E, the performance is dominated by scanning a large range of KV pairs. Thereby the difference is relatively small, and XSTORE outperforms other systems by up to 1.8× (from 1.4×).

Fig. 11 further shows the timelines for YCSB D with Uniform and Latest workloads. The optimal throughput of tree-based index cache can only achieve about 25M reqs/s, more than 3× lower than its read-only throughput (78M reqs/s), and suffers from severe performance fluctuations due to frequent cache invalidations, especially for Uniform distribution. For Latest distribution, each client will focus on a small range of KV pairs (latest inserted by itself), which significantly reduces cache misses and invalidations due to accessing internal nodes split by other clients. XSTORE preserves relatively high throughput and has steady cache invalidation rates, 5% for Uniform, and 21% for Latest. It is mainly because stale learned cache can still provide a correct prediction for most read requests. The speculative execution also helps to halve the rate (from 10% to 5%). In addition, in Latest distribution, each client will frequently access KV pairs just inserted. If the insert incurs a node split, XSTORE might not fetch a new model immediately (wait for model retraining) and would increase cache misses.

***CPU utilizations of* XSTORE.** Note that XSTORE uses two auxiliary threads to retrain XMODEL for dynamic work-

[9] The data distribution of dynamic workloads (i.e., YCSB D and E) is close to noised linear (NL). Hence, XSTORE can only achieve 61M reqs/s for YCSB D with 2M models even no inserts (see Fig. 14b and Fig. 15d).

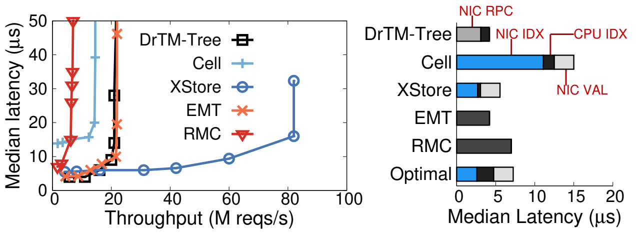


Fig. 12. Comparison of (a) throughput-latency and (b) end-to-end median latency at low load for YCSB-C with a uniform distribution.

loads, causing increased server CPU usage. Yet, XSTORE still saves server CPUs compared to server-centric KVs (e.g., DrTM-Tree) due to handling read requests in the clients. For example, DrTM-Tree saturates all CPUs (24 × 100%) for YCSB D, while XSTORE just consumes under half for serving insert requests and retraining sub-models.

End-to-end latency. Fig. 12a shows the throughput-latency curves for YCSB C with a uniform distribution. Due to space limitations, we omit other workloads that are similar. When using few clients (low load), server-centric KVs have lower latency, as one RPC round trip is faster than two one-sided RDMA operations, namely DrTM-Tree (NIC_RPC) vs. XSTORE (NIC_IDX and NIC_VAL) in Fig. 12b. However, the throughput of them (e.g., DrTM-Tree) is saturated by CPUs much earlier (about 20M reqs/s), and the latency would rapidly collapse. On the other hand, the latency of Cell is limited by multiple RDMA READs for each lookup (NIC_IDX) even at low load. In contrast, XSTORE only needs one RDMA READ, thanks to the learned cache. As a reference, we provide the latency of using whole-index cache (Optimal) that also takes just one RDMA READ. However, traversing tree-based index locally still takes more time (2.14µs in CPU_IDX) due to many random memory accesses, compared to XSTORE (0.35µs). Moreover, XSTORE can keep low latency at much high load (82M reqs/s with median latency of 16µs) by eliminating CPU bottleneck at the server.

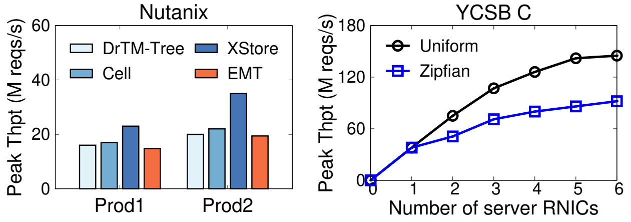


Fig. 13. (a) Performance comparison with production workloads. (b) Scalability of XSTORE on YCSB C with the increase of RNICs.

7.3 Production Workload Performance

Fig. 13a shows the peak throughput of XSTORE and other systems on two write-intensive production workloads, similar to YCSB A. The performance is also mainly bottlenecked by server CPUs due to 57% of writes. In the first workload (Prod1), XSTORE outperforms DrTM-Tree, EMT, and Cell by 1.44×, 1.55×, and 1.35×, respectively. The speedup in the second workload (Prod2) increases to 1.75×, 1.80×, and 1.60× since this workload is more skewed.

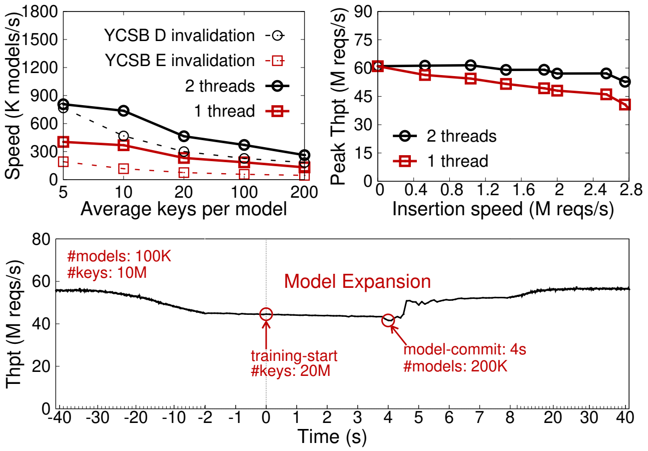


Fig. 14. (a) Comparison between sub-model retraining and invalidation speed. (b) Performance of XSTORE with the increase of insertion speed. (c) Performance timeline with model expansion.

7.4 Scale-out Performance

Fig. 13b shows the scalability of XSTORE with up to 6 server RNICs (3 server machines). We scale XSTORE by range-based partitioning a YCSB dataset with 600M keys into different numbers of RNICs. The performance is measured using up to 13 client machines (26 RNICs) with a read-only workload. For a uniform request distribution, XSTORE achieve a peak throughput of 145M reqs/s, which is limited by the number of client machines. Note that, on our testbed, XSTORE needs about eight client RNICs to saturate one server RNIC. XSTORE scales to 1.97× and 2.81× by using 2 and 3 server RNICs, respectively. For a skewed request distribution (Zipfian), XSTORE just reaches 92M reqs/s by using 6 server RNICs since most requests (more than 35%) are sent to one RNIC. It throttles the entire system.

7.5 Model (Re-)Training and Expansion

Fig. 14a shows the throughput of training models using one or two threads and model invalidation speed for dynamic workloads (YCSB D and E). Empirically, using two threads for model retraining is sufficient for XSTORE to reach a throughput of 53M reqs/s (YCSB D). XSTORE can retrain sub-models individually and takes 8µs on average to retrain a model with 200 keys. Note that the insertion speed reaches about 2.65M reqs/s for YCSB D (5% inserts). For dynamic workloads, the throughput of XSTORE would decrease when stale sub-models can not retrained in time. To quantify the performance overhead, we evaluate XSTORE with the increase of insertion speed, similar to YCSB D (except that one client is dedicated to insert key-value pairs with a given speed, and the rest of clients still issue reads). As shown in Fig. 14b, the throughout drops below 40% (61M vs. 37M reqs/s) under the peak insertion speed (2.8M reqs/s, limited by server CPUs) when using a single retraining thread. Further, when using two threads, the performance degradation is limited to 13%.

Finally, the growing size of KV pairs in the ML model

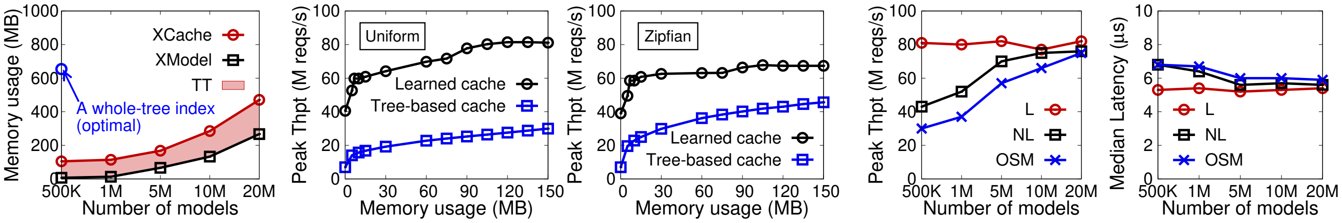


Fig. 15. (a) Memory usage of learned cache (XCache). Comparison of peak throughput between learned cache and tree-based cache with different memory footprint at the client for YCSB C using (b) Uniform and (c) Zipfian distributions. Comparison of (d) peak throughput and (e) median latency on XStore with the increase of models for various data distributions (see Table 2).

will likely increase the prediction error, resulting in performance degradation. XStore supports model expansion to increase models in the background if needed. As shown in Fig. 14c, starting from 10M keys and 100K models, several clients continuously insert KV pairs, and the performance of XStore slowly degrades for read requests. When the average number of keys per model exceeds 200 (a user-defined threshold), the server starts to train a new XModel with double sub-models (200K) in the background from 0s to 4s, with negligible overhead. After that, the server will commit the new model, and clients could individually fetch new sub-models on demand. The performance resumes rapidly in 2s.

7.6 Memory Footprint of XCache

Fig. 15a presents the memory usage of XCache with the increase of sub-models for 100M KV pairs. Note that the entire XTree has 654MB internal nodes. The size of TT depends on the number of leaf nodes. Since each leaf node has 16 slots for KV pairs, TT occupies around 98MB as the tree-based index is half-full. Thus, TT would dominate the memory usage for a small XModel since each sub-model is 14B large. To achieve peak throughput, XModel with 500K sub-models is enough for read-only workloads (YCSB C) with 100M KV pairs, while it needs 2M sub-models for dynamic workloads (YCSB D) with 250M KV pairs.

As shown in Fig. 15b and Fig. 15c, compared to conventional tree-based index cache, XStore can provide competitive performance with much lower memory footprint at the clients, even (almost) no memory footprint. XCache prefers to store XModel, which may only occupy 1% memory (6.8M vs. 654MB). It means that, for YCSB C with Uniform and Zipfian distributions, XStore can achieve 74% and 87% of optimal throughput (a whole-index cache), where the client uses one additional RDMA READ to fetch several 8-byte TT entries for each lookup. Even if the client only stores a 16-byte top model, XStore can still achieve about 40M reqs/s by using one RDMA READ to fetch a 14-byte sub-model first.

7.7 Data Distribution

We further evaluate XStore on a 100M-key dataset with different data distributions in Table 2 using a read-only workload (YCSB C). The throughput of XStore is sensitive to the prediction error due to bandwidth amplification for retrieving more keys. Thus, XStore requires more simple sub-models (e.g., LR) to learn complex data distributions (e.g., OSM) for the same prediction error. For example, as shown in Fig. 15d, XStore requires about 20M sub-models for OSM to achieve a peak throughput of 80M reqs/s. However, as shown in Fig. 15e, the median latency at a low load is relatively stable for various data distributions, as the latency of RDMA is insensitive to payload sizes when the network is not saturated [39].

Table 3: The impact of durability on throughput (M reqs/s).

YCSB /Uniform	A	B	C	D	E	F
w/o logging	41	80	82	53	10.2	36
w logging	31	78	82	51	9.9	33

7.8 Durability

To study the overhead of logging for durability, we evaluate the peak throughput of XStore for various YCSB workloads with logging to SSD enabled. As shown in Table 3, the performance drops by up to 24% for update-heavy workloads (e.g., YCSB A) due to additional writes to SSD for write operations (e.g., Update). On the other hand, it does not degrade the performance of read-heavy workloads much (e.g., YCSB C). First, XStore executes read operations (e.g., Get) using one-sided RDMA primitives, bypassing the logging threads thoroughly. Second, XStore flushes the logs in a batched manner [33], which hides the impact of slow storage (§5.4).

7.9 Variable-length Value

By default, XStore directly stores the value in leaf nodes (*inline value*). To support variable-length values, XStore stores a 64-bit fat pointer (the size and the position of value) in leaf nodes (*indirect value*). Consequently, the client needs an additional RDMA READ to retrieve the variable-length value (Line 13 in Fig. 7). Fig. 16a shows the performance of XStore by using inline and indirect value. Using *indirect value* causes up to 43% (from 8%) performance degradation, compared to using *inline value*. The performance gap is closing with the increase of values (e.g., 1KB) since the cost of one additional RDMA READ becomes trivial.

7.10 Application Performance

To demonstrate the effectiveness of XStore in application workloads, we have integrated it into DrTM+H [51], a state-

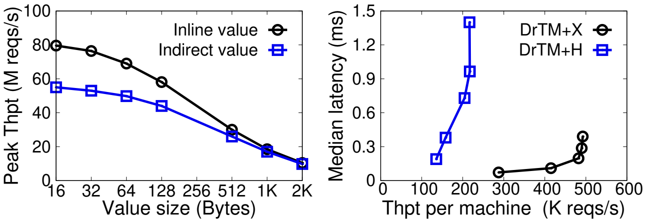


Fig. 16. (a) Performance of XStore by using inline and indirect value. (b) Comparison of DrTM+H on TPC-C w/ and w/o XStore.

of-the-art distributed OLTP system that leverages RDMA-enabled KVS to store tuples. The vanilla DrTM+H only performs unordered index lookups (hash table) by using one-sided RDMA primitives [52]. DrTM+H with XStore (called DrTM+X) can further perform ordered index lookups (B+tree) through one-sided RDMA operations.

Experimental setup. We use TPC-C [45] to compare the performance of DrTM+H and DrTM+X. Note that both of them run in an asymmetric setting, which is widely adopted in cloud databases [55, 47, 7].[10] More specifically, we deploy 96 warehouses on four data servers and use the rest of the machines in our testbed as clients. Both DrTM+H and DrTM+X rely on the data server to update tuples, while DrTM+X uses one-sided RDMA READs to retrieve tuples from the data server. Therefore, we use a read-heavy TPC-C workload in the experiment, which consists of New-Order transactions (10%) and Order-Status transactions (90%). NewOrder transaction inserts a new order with five to fifteen order lines; OrderStatus transaction retrieves the recently inserted orders first and then scans related order lines.

Performance. As shown in Fig. 16b, XStore improves the peak throughput of DrTM+H by 2.27×, reaching 490K reqs/s. DrTM+H is bottlenecked by server CPUs since the data server traverses the index and performs the read request locally. Consequently, the read requests of OrderStatus transactions would compete CPUs with the write requests of NewOrder transactions at the servers. Differently, DrTM+X relies on RNICs at the clients to lookup and retrieve tuples for OrderStatus transactions. It relaxes the burden on server CPUs and improves performance significantly.

8 Related Work

RDMA-enabled key-value stores. XStore continues the line of research of RDMA-based in-memory key-value stores [31, 34, 25, 16, 52, 35, 43, 57, 8, 48], but explores a new design point, namely *learned cache*, that leverages machine learning (ML) models as index cache for RDMA-based, tree-backed key-value store. There have been many efforts to investigate RDMA-based unordered in-memory KVs which focus on such as improving the communication layer (e.g.,RPC) [25, 24, 10], selecting appropriate hash tables [34, 16, 52], supporting index caching [52, 48], and enabling in-network processing [31, 40].

There is an increasing interest in optimizing tree-backed in-memory key-value stores with RDMA. Cell [35] allows clients to traverse server's B+Tree using RDMA READs and caches the top three levels of tree index. FaRM B-Tree [17] caches B-tree's internal nodes at each server to accelerate lookups using RDMA, while it is costly and error-prone for dynamic workloads [38]. Ziegler et al. [57] studies different RDMA-based design alternatives for tree-based index, including how the tree should be distributed and the choices of RDMA primitives for tree operations.

Learned indexes and their applications in systems. Kraska et al. [29] argue that all existing index structures can be replaced with machine learning (ML) models, which are termed "learned index", and further propose several example learned indexes to replace various index structures, including tree-based range index. There have been several recent efforts of adapting learned indexes to handle dynamic workloads [44, 15, 36]. XIndex [44] adds a delta index to each sub-model in a learned index and proposes a new concurrent compaction scheme to split models. ALEX [15] uses a gapped array to accommodate new key-value pairs, similar to the leaf node of XTree. However, it is non-trivial to enable the gapped array in a distributed system since it requires complex coordinations when expanding the array upon full. Bourbon [14] is a log-structured merge (LSM) tree that leverages the learned index to speedup lookups. FITing-Tree [18] is a form of a learned index to balance prediction error and memory cost. It uses extra sorted buffers to store inserts and merges them back when reaching a threshold. SIndex [49] is a concurrent learned index for variable-length string keys. Differently, XStore proposes a hybrid architecture to leverage ML models as RDMA-based index cache, instead of replacing or augmenting traditional index structures.

9 Conclusion

This paper presents XStore, an RDMA-based in-memory ordered key-value store with a new *hybrid* architecture to leverage ML model as RDMA-based index cache. Our experimental results show the high performance of XStore.

10 Acknowledgment

We sincerely thank our shepherd Andrea C. Arpaci-Dusseau and the anonymous reviewers for their insightful suggestions. We also thank Zhaoguo Wang, Chuzhe Tang, Zhiyuan Dong and Youyun Wang for sharing their experience on *learned* index, and Xiating Xie for the valuable feedback. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010164003), the National Natural Science Foundation of China (No. 61772335, 61925206, 61732010), the High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), and a research grant from Huawei Technologies. Corresponding author: Rong Chen (`rongchen@sjtu.edu.cn`).

[10] Prior work [51] has shown that using (two-sided) RDMA-based RPC is a better choice for Get operations in a symmetric setting [17].

References

[1] Memcached. https://memcached.org/.

[2] OpenStreetMap (OSM) on AWS. https://aws.amazon.com/public-datasets/osm, 2020.

[3] AGUILERA, M. K., KEETON, K., NOVAKOVIC, S., AND SINGHAL, S. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2019), HotOS '19, Association for Computing Machinery, p. 120–126.

[4] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.

[5] BLUNDELL, C., LEWIS, E. C., AND MARTIN, M. M. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters 5*, 2 (2006).

[6] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H. C., ET AL. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference* (2013), pp. 49–60.

[7] CAO, W., LIU, Z., WANG, P., CHEN, S., ZHU, C., ZHENG, S., WANG, Y., AND MA, G. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment 11*, 12, 1849–1862.

[8] CASSELL, B., SZEPESI, T., WONG, B., BRECHT, T., MA, J., AND LIU, X. Nessie: A decoupled, client-driven key-value store using rdma. *IEEE Transactions on Parallel and Distributed Systems 28*, 12 (2017), 3537–3552.

[9] CHEN, H., CHEN, R., WEI, X., SHI, J., CHEN, Y., WANG, Z., ZANG, B., AND GUAN, H. Fast in-memory transaction processing using rdma and htm. *ACM Trans. Comput. Syst. 35*, 1 (July 2017).

[10] CHEN, Y., LU, Y., AND SHU, J. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[11] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 26.

[12] COOPER, B. F. YCSB Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[13] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC'10, ACM, pp. 143–154.

[14] DAI, Y., XU, Y., GANESAN, A., ALAGAPPAN, R., KROTH, B., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. From wisckey to bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation* (2020), OSDI '20, USENIX Association.

[15] DING, J., MINHAS, U. F., YU, J., WANG, C., DO, J., LI, Y., ZHANG, H., CHANDRAMOULI, B., GEHRKE, J., KOSSMANN, D., LOMET, D., AND KRASKA, T. Alex: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 969–984.

[16] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14, USENIX Association, pp. 401–414.

[17] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP'15, ACM, pp. 54–70.

[18] GALAKATOS, A., MARKOVITCH, M., BINNIG, C., FONSECA, R., AND KRASKA, T. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1189–1206.

[19] GRAEFE, G. Write-optimized b-trees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases* (2004), VLDB '04, VLDB Endowment, p. 672–683.

[20] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM'16, ACM, pp. 202–215.

[21] GUPTA, M., COTTER, A., PFEIFER, J., VOEVODSKI, K., CANINI, K., MANGYLOV, A., MOCZYDLOWSKI, W., AND VAN ESBROECK, A. Monotonic calibrated interpolated look-up tables. *J. Mach. Learn. Res. 17*, 1 (Jan. 2016), 3790–3836.

[22] HIGH-PERFORMANCE BIG DATA (HIBD). RDMA-based Memcached (RDMA-Memcached). http://hibd.cse.ohio-state.edu.

[23] JONAS, E., SCHLEIER-SMITH, J., SREEKANTI, V., TSAI, C.-C., KHANDELWAL, A., PU, Q., SHANKAR, V., CARREIRA, J., KRAUTH, K., YADWADKAR, N., ET AL. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[24] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. Datacenter rpcs can be general and fast. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 1–16.

[25] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), SIGCOMM'14, ACM, pp. 295–306.

[26] Kalia, A., Kaminsky, M., and Andersen, D. G. Fasst: fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association, pp. 185–201.

[27] Kaminsky, A. K. M., and Andersen, D. G. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference* (2016), p. 437.

[28] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (USA, 2018), OSDI'18, USENIX Association, p. 427–444.

[29] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data* (2018), ACM, pp. 489–504.

[30] Lepers, B., Balmau, O., Gupta, K., and Zwaenepoel, W. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 447–461.

[31] Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., and Zhang, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 137–152.

[32] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 429–444.

[33] Mao, Y., Kohler, E., and Morris, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys'12, ACM, pp. 183–196.

[34] Mitchell, C., Geng, Y., and Li, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), USENIX ATC'13, USENIX Association, pp. 103–114.

[35] Mitchell, C., Montgomery, K., Nelson, L., Sen, S., and Li, J. Balancing cpu and network in the cell distributed b-tree store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).

[36] Nathan, V., Ding, J., Alizadeh, M., and Kraska, T. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 985–1000.

[37] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al. Scaling memcache at facebook. In *nsdi* (2013), vol. 13, pp. 385–398.

[38] Shamis, A., Renzelmann, M., Novakovic, S., Chatzopoulos, G., Dragojević, A., Narayanan, D., and Castro, M. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 433–448.

[39] Shi, J., Yao, Y., Chen, R., Chen, H., and Li, F. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 317–332.

[40] Sidler, D., Wang, Z., Chiosa, M., Kulkarni, A., and Alonso, G. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[41] Sowell, B., Golab, W., and Shah, M. A. Minuet: A scalable distributed multiversion b-tree. *Proc. VLDB Endow. 5*, 9 (May 2012), 884–895.

[42] Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Gonzalez, J. E., Hellerstein, J. M., and Tumanov, A. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow. 13*, 12 (jul 2020), 2438–2452.

[43] Su, M., Zhang, M., Chen, K., Guo, Z., and Wu, Y. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 1–15.

[44] Tang, C., Wang, Y., Dong, Z., Hu, G., Wang, Z., Wang, M., and Chen, H. Xindex: A scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2020), PPoPP '20, Association for Computing Machinery, p. 308–320.

[45] The Transaction Processing Council. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/.

[46] Tsai, S.-Y., and Zhang, Y. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 306–324.

[47] Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., and Bao, X. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 1041–1052.

[48] Wang, Y., Meng, X., Zhang, L., and Tan, J. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SoCC'14, ACM, pp. 23:1–23:13.

[49] Wang, Y., Tang, C., Wang, Z., and Chen, H. Sindex: A scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2020), APSys '20, Association for Computing Machinery, p. 17–24.

[50] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys'14, ACM, pp. 26:1–26:15.

[51] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation* (2018), OSDI '18, pp. 233–251.

[52] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[53] XIE, X., WEI, X., CHEN, R., AND CHEN, H. Pragh: Locality-preserving graph traversal with split live migration. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 723–738.

[54] YOU, S., DING, D., CANINI, K., PFEIFER, J., AND GUPTA, M. Deep lattice networks and partial monotonic functions. In *Advances in neural information processing systems* (2017), pp. 2981–2989.

[55] ZAMANIAN, E., BINNIG, C., HARRIS, T., AND KRASKA, T. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow. 10*, 6 (Feb. 2017), 685–696.

[56] ZHANG, H., ANDERSEN, D. G., PAVLO, A., KAMINSKY, M., MA, L., AND SHEN, R. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data* (2016), ACM, pp. 1567–1581.

[57] ZIEGLER, T., TUMKUR VANI, S., BINNIG, C., FONSECA, R., AND KRASKA, T. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, ACM, pp. 741–758.

A Artifact Appendix

A.1 Abstract

This artifact provides the source code of XStore and scripts to reproduce the main experimental results. XStore is an RDMA-based ordered key-value store that adopts the client-server model (network-attached) and range index structures (tree-backed). To reproduce the results, we provide instructions to build binaries (§A.3) and run experiments (§A.4). The source code of XStore can be retrieved from a public open-source repository (§A.2.1). The repository also contains scripts to generate the main results in §7 (see Table 4). Though the scripts target our testbed (§7.1), readers can simply change them for other platforms (§A.6).

A.2 Artifact Check-list

- **Program:** `fserver`, `ycsb`, and `micro`.
- **Compilation:** `g++` and `cmake`.
- **Hardware:** Intel CPU with RTM and Mellanox NIC with RDMA.
- **Execution:** Python scripts.
- **Metrics:** Throughput and median latency.
- **Expected experiment run time:** 1 minute each experiment.
- **Public link:** `https://github.com/SJTU-IPADS/xstore`.
- **Code licenses:** Apache License 2.0.

A.2.1 How to Access

The artifact is publicly available at our Github repository.

```
$ git clone https://github.com/SJTU-IPADS/xstore
$ git checkout c9f38188
```

A.2.2 Hardware Dependencies

To reproduce the experiment results, each machine must have at least one Mellanox RDMA network card (e.g., Mellanox ConnectX-4 MT27700 100Gbps InfiniBand NIC), and the server machine must have Intel processors with Restricted Transactional Memory (RTM) (e.g., Xeon E5-2650 v4). It should be noted that the throughput of read operations (e.g., gets) is mainly bottlenecked by the RDMA network, while the throughput of write operations (e.g., updates) is mainly bottlenecked by the server CPU.

A.2.3 Software Dependencies

Operating system: `Ubuntu` ≥ `16.04`.

Compile toolchain: `g++` ≥ `5.4.4` and `cmake` ≥ `3.5.1`.

Other software dependencies: `Intel MKL`, `Mellanox OFED`, `boost 1.6.1`, and `jemalloc`.

A.3 Installation

Intel MKL (Math Kernel Library).

```
$ apt-get install -y intel-mkl-2019.1-053
```

Listing 1: A sample evaluating script (`sample.toml`).

```
[[pass]]
host = server_host    ## host name of the server
path = /cock/fstore
cmd = "./fserver -db_type ycsb -model_config=
    ycsb-model.toml"

[[pass]]
host = client_host    ## host name of the client
path = /cock/fstore
cmd = "./ycsb -threads 1 -server_host
    server_host"

[[pass]]
host = master_host    ## host name of the client
path = /cock/fstore
cmd = "./master -client_config cs.toml -epoch 60
     -nclients 1"
```

Mellanox OFED.

```
$ wget latest_ofed_for_the_OS.
$ ./mlnxofedinstall -without-iser-dkms
-without-srp-dkms -without-srptools -force
```

Boost and jemalloc.

```
$ cd path_to_xstore
$ pip3 install -r requirements.txt
$ ./magic.py config -f build-config.toml
$ cmake .
$ cd deps/jemalloc
$ autoconf
$ cd path_to_xstore
$ make boost jemalloc
```

XStore.

```
$ make fserver ycsb micro master
```

A.4 Experiment Workflow

Launch XStore server.

```
$ ssh server_host
$ ./fserver -db_type ycsb -model_-
config=ycsb-model.toml
```

Launch XStore clients.

```
$ ssh client_host
$ ./ycsb -threads 1 -server_host server_host
```

Launch a master to collect results from clients.

```
$ ssh master_host
$ ./master -client_config cs.toml -epoch 60
-nclients 1
```

Automatic experiment workflow. Optionally, readers could use our script (`bootstrap.py`) to automate the above three steps. It takes a configuration file (e.g., Listing 1) to execute the above three steps required for the experiments. Specifically, the following command should launch the server, the clients, and the master accordingly:

```
$ ./bootstrap.py -f sample.toml
```

Table 4: The evaluating scripts to reproduce the results in §7. Note that the scripts are in the ae_scripts folder in our repository.

Figure	Description	Evaluating script
Fig. 10	YCSB A	ycsba.toml
Fig. 11	YCSB B	ycsbb.toml
Fig. 12	YCSB C	ycsbc.toml
	YCSB D	ycsbd.toml
	YCSB E	ycsbe.toml
Fig. 14c	Model expansion	expan.toml
Fig. 15d,e	Data distribution	ln.toml
Fig. 15b,c	Memory footprint	cached_ycsbc.toml

A.5 Evaluation and Expected Result

The experimental results mainly include throughput and meidan latency. By default, the master is responsible for printing the results. It should be noted that it is difficult to compare the performance results across different machines. Therefore, we only show the reported numbers on our testbed as an example here. For instance, to evaluate YCSB C on XSTORE, readers could run the script as follows:

```
$ ./bootstrap.py -f ae_scripts/ycsbc.toml
```

The master would print throughput (*thpt*) and median latency (*lat*) per second:

```
...
At epoch 1 thpt: 79.4M/s, ..., lat: 19.5 us
At epoch 2 thpt: 79.3M/s, ..., lat: 19.6 us
At epoch 3 thpt: 79.8M/s, ..., lat: 19.4 us
...
```

A.6 Experiment Customization

Table 4 lists the configuration files used to produce the experimental results in our paper. However, the scripts mainly target our testbed (§7.1). To execute them on other platforms, readers need to make minor changes to the scripts. The README in our repository provides detailed information about how to customize them for the experiments.

A.7 Notes

The source code and scripts for the artifact evaluation are used to reproduce the main results in XSTORE. To use XSTORE in your research, we recommend the main branch of our repository (§A.2.1), which would be maintained by members of the Institute of Parallel and Distributed Systems.

A.8 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts