Fat-B$^+$Tree: Fast B$^+$tree Indexing with In-Network Memory

Yikai Zhao[*], Yuanpeng Li[*], Zicang Xu[*], Tong Yang[*], Kaicheng Yang[*], Li Chen[†], Xin Yao[‡], Gong Zhang[‡]
[*] Peking University, China [†] Zhongguancun Laboratory, China [‡] Huawei Theory Lab, China

***Abstract*—In-memory database in the data center plays an indispensable role in many fields. B$^+$tree is the most recognized index in in-memory database, but its indexing latency has become the bottleneck that prevents the database from achieving higher performance. The existing work reduces latency by caching B$^+$tree nodes using slow DRAM on computing clients, or directly caching data using fast SRAM on programmable switch. However, no existing work can meet all three key requirements: (1) Efficiency: providing enough fast memory to accelerate indexing. (2) Compatibility: compatible with multiple architectures and query types. (3) Adaptability: adapting to various workloads and database scales. Inspired by structrual similarity between the network topology of modern data centers and the B$^+$tree structure, we propose Fat-B$^+$Tree, which embeds the B$^+$tree into the in-network fast memory provided by the hierarchical connected programmable switches, thus meeting all design requirements. We have fully implemented the Fat-B$^+$Tree prototype, and the experimental results show that compared with the baseline system, it reduces query latency by up to 76% and improves throughput by up to 3.93 times. The source codes of Fat-B$^+$Tree are open-sourced at GitHub.**

I. Introduction

In-memory databases are the infrastructure for a vast number of fields [1]–[4]. B$^+$tree is the most widely used index in in-memory database [5]. In-memory databases often improve the performance by separating computing and memory into different machines in the data center [6]. The two most promising architectures are client-server [7], [8] and memory disaggregation [9]–[16]. In the client-server architecture, memory servers perform the indexing through the local B$^+$tree, so the indexing performance of the B$^+$tree directly determines the throughput. In the memory disaggregation architecture, the computing clients remotely access the B$^+$tree in the disaggregated memory provided by the memory servers through the RDMA (Remote Direct Memory Access) [17], [18], so that the indexing time becomes the main component of the query latency.

Optimizing the index performance of B$^+$tree is challenging because of its scattered structure connected by pointers [19]. For local memory, every access from the root to the leaf has a high probability of causing cache misses, which increases latency. For disaggregated memory, each access requires one RDMA read, resulting in a microsecond-level round trip latency. We believe that an ideal system for optimizing the B$^+$tree should meet the following requirements: **(R1) Efficiency:** it can provide large capacity of fast memory (*e.g.*, SRAM) to accelerate the data indexing. **(R2) Compatibility:** it is compatible with both client-server and memory disaggregation, and it is compatible with common operations of B$^+$tree. **(R3) Adaptability:** it can adapt to the dynamic changes of workload, and it can adapt to the scale of the database.

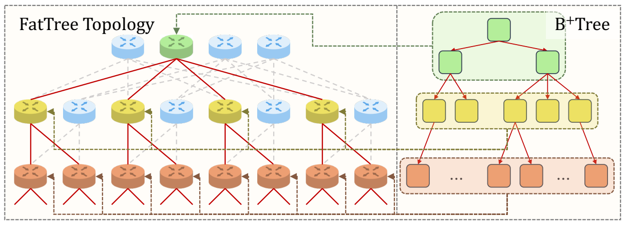


Figure 1: Exploiting the structural similarity between FatTree topology and B$^+$tree.

The existing systems for optimizing data indexing cannot meet the requirements simultaneously. We loosely divide them into two categories. The first category optimizes the structure of the B$^+$tree for RDMA network, and reduces latency by caching nodes in the DRAM of hundreds MB in computing clients [8], [17], [20]–[22]. However, they fail to meet R1 because the SRAM available on the computing client is only 1$\sim$2MB per core. The second category uses the programmable switch to directly cache the information of the hottest keys with tens MB of SRAM in the data plane of edge switch [23]–[26]. However, they fail to meet R2 because they do not support range query and RDMA network. They also fail to meet the R3 because they can only cache the information of a few keys, so they are sensitive to workload changes [23].

Figure 1 shows the FatTree topology and the data structure of the B$^+$tree: there is a three-layer tree consisting of 13 switches in the FatTree topology with $k = 4$, and these switches can provide a total of more than 100 MB of fast memory (SRAM and TCAM[1]). Based on this observation, we propose Fat-B$^+$Tree that embeds part of the B$^+$tree into the hierarchical connected programmable switches. Fat-B$^+$Tree meets the requirements at the same time. (**R1**) It is efficient: it can leverage hundreds MB or even several GB of fast on-chip memory in the switches to perform in-network indexing. Compared with the baseline system, it can reduce query latency by up to 76% and improve throughput by up to 3.93 times. (**R2**) It is compatible: it embeds B$^+$tree into the data plane to support common operations of B$^+$tree, and it can be deployed in Ethernet and RoCE networks to support both architectures. (**R3**) It is adaptive: it completely stores the nodes in the top layers of the B$^+$tree, which can always reduce the indexing latency under any workload and any scale.

[1]Ternary Content Addressable Memory.

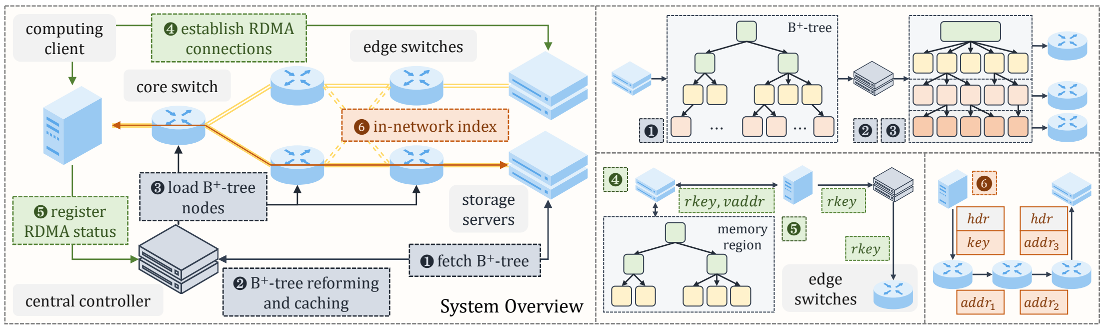


Figure 2: Overview and workflow of Fat-B$^+$Tree.

The programmability of modern switches enables us to perform the in-network indexing while forwarding the query packet in the data plane. However, embedding the B$^+$tree and the corresponding index operation into the data plane still faces several challenges: (**C1**) Implement the data structure and operations of the B$^+$tree through the limited programming model provided by the data plane. (**C2**) Bridge the gap between the Mega-byte level of memory in the data plane and the Giga-byte B$^+$tree of large-scale databases. And (**C3**) Modify the packets of RDMA operators on the data plane that only supports Ethernet. To address these key challenges, our proposed system Fat-B$^+$Tree designs several key techniques and mechanisms according to the following methodologies.

Extending supported key range: *using components with low bit-width to achieve the function of high bit-width.* To support the in-network indexing of 64-bit key, we need to implement the comparator and range matching table of 64-bit data in the data plane. Therefore, we design a method to implement a high bit-width comparator with multiple low bit-width compactors and matching tables in the data plane (§III-A), and also propose an algorithm to encode n high bit-width ranges to $O(n)$ low bit-width sub-tables and table entries (§III-B). With the expanded programming capabilities, we can implement B$^+$tree nodes with SRAM or TCAM respectively in the data plane.

Query acceleration with selective node caching: *reforming the B$^+$tree to make full and reasonable use of memory resources.* Although there is a gap of several orders of magnitude between the SRAM resources of data plane and the memory volume required by the large-scale B$^+$tree, due to the highly skewed distribution of query keys, caching only a few B$^+$tree nodes can cover a large proportion of queries. Therefore, based on the B$^+$tree structure and the pipeline architecture of the data plane, we design a layer-by-layer node caching mechanism to select nodes to be loaded into the data plane (§III-C). On the other hand, we also design a greedy B$^+$tree reforming algorithm (§III-B). By using TCAM resources, it can not only make full use of data plane memory resources, but also reduce the depth of B$^+$tree to improve performance.

Native RDMA integration: *using the protocol independence of the data plane to support RDMA protocol.* Although the current programmable data plane is designed for Ethernet rather than Infiniband network, RoCE (RDMA over Converged Ethernet) mechanism [27], [28] enables RDMA messages to be transmitted in Ethernet. Benefiting from the protocol independence of PISA (Protocol Independent Switch Architecture) of programmable data plane, we design processing logic of RDMA packets to enable RDMA packets to carry query keys and perform in-network indexing (§III-D).

Key contributions: In this paper, we make the following contributions: (1) We propose a novel in-network indexing system Fat-B$^+$Tree to accelerate database queries under the memory disaggregation architecture. As far as we know, it is the first system to embed tree index structure into programmable data plane. It meets all design requirements. (2) We implement the prototype system of Fat-B$^+$Tree, which supports both RDMA network and Ethernet. We also design network-wide deployment mechanisms to enable Fat-B$^+$Tree to support hierarchical connected switches, read-write mixed workloads, and parallel-accelerated range query. (3) We perform a comprehensive evaluation of Fat-B$^+$Tree in both RDMA network and Ethernet scenarios through abundant experiments, and the experimental results show that Fat-B$^+$Tree is superior to the baseline system in many aspects. The source codes of Fat-B$^+$Tree are open-sourced at GitHub [29].

II. Design Overview

In this section, we use Figure 2 to show the overview and workflow of Fat-B$^+$Tree, which includes six key steps that can be divided into three parts:

- The first part of Fat-B$^+$Tree is to select and reform the nodes of the B$^+$tree to load them into the data plane of switches. This part consists of step ❶ to ❸, and involves memory servers, the central controller, and all switches: ❶ The memory server builds the B$^+$tree index structure based on the stored data, and transmits the information of the B$^+$tree nodes to the central controller as required. ❷ The central controller reforms the B$^+$tree through the built-in greedy algorithm, and selects nodes with high access frequency to construct a layer-by-layer cache. ❸ The central controller loads the selected nodes into the data plane of switches layer by layer.
- The second part of Fat-B$^+$Tree is to establish the RDMA connection between clients and servers, and register the RDMA status. This part consists of step ❹ and ❺, and involves the computing client, memory servers, the central controller, and edge switches: ❹ The computing client establishes RDMA connections with the memory servers

through CM (Communication Manager) protocol, and obtains metadata of the memory regions where the B$^+$tree is located on the memory servers. ❺ The computing client transmits the RDMA connection metadata to the controller, and the controller sends it to the corresponding edge switch.

- The third part of Fat-B$^+$Tree is to provide in-network indexing for database queries. This part consists of step ❻, and involves the computing client, switches, and memory servers: ❻ The computing client constructs the RDMA-read operator sent to the memory server and embeds the query key in the packet; The switches on the forwarding path perform the in-network indexing and update the target memory address of the RDMA-read operator according to the query key and the B$^+$tree nodes stored in the data plane; As a result, the computing client can perform subsequent remote indexing directly from an internal node of the B$^+$tree instead of the original root node.

Our proposed Fat-B$^+$Tree supports both RDMA-based queries (for memory disaggregation architecture) and TCP/IP-based queries (for client-server architecture). For queries based on TCP/IP connection, the system only contains the first and third parts, in which the third part will be modified according to the application layer protocol. Memory servers decide whether to perform the indexing from the root node or an internal node of the B$^+$tree according to the query address carried by the received query packets.

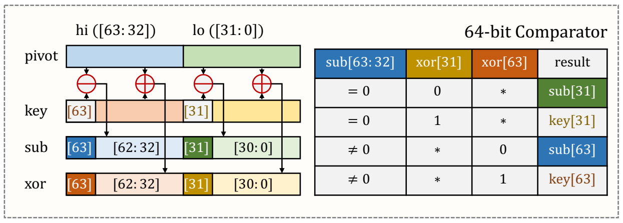
III. System Design in the Switch

The design of our in-network index system Fat-B$^+$Tree is mainly composed of the following four components or mechanisms: 1) SRAM-based regular B$^+$tree nodes, 2) TCAM-based fat-root generation and corresponding B$^+$tree reform mechanism, 3) layer-by-layer node caching mechanism, and 4) data plane RDMA encapsulation mechanism. We describe the design of these components and mechanisms in detail in this section.

A. *Regular B$^+$tree Nodes*

When accessing a K-branch B$^+$tree node supporting 64-bit keys on the CPU platform, we need to fetch $K - 1$ internal pivot keys from memory according to the node address, and compare the input key and these pivot keys to determine the address of the next node to be accessed. However, since the programmable data plane neither supports the comparison between 64-bit data nor provides the address space as large as that of the CPU platform, we need to adjust the above steps to adapt the special programming model. We first introduce how to use 32-bit arithmetic units and match-action tables to achieve comparison between higher bit-width data, and then show how to use tables to fetch internal keys and the next address in the data plane pipeline.

Design of 64-bit comparator: As shown in Figure 3, given two 64-bit data $key[63 : 0]$ (input key) and $pivot[63 : 0]$ (internal key), their comparison consists of two phases. In the first phase, we use 32-bit arithmetic units to perform



sub[63: 32]	xor[31]	xor[63]	result
= 0	0	*	sub[31]
= 0	1	*	key[31]
≠ 0	*	0	sub[63]
≠ 0	*	1	key[63]

Figure 3: An example of 64-bit data comparison using 32-bit arithmetic units and match-action tables.

operations on the high 32 bits and low 32 bits of data respectively, and get:

$$\begin{aligned} sub[63 : 32] &= pivot[63 : 32] - key[63 : 32] \\ xor[63 : 32] &= pivot[63 : 32] \oplus key[63 : 32] \\ sub[31 : 0] &= pivot[31 : 0] - key[31 : 0] \\ xor[31 : 0] &= pivot[31 : 0] \oplus key[31 : 0]. \end{aligned}$$

In the second phase, a value $sub[63 : 32]$ and a total of 6 bits ($key[63]$, $key[31]$, $sub[63]$, $sub[31]$, $xor[63]$, $xor[31]$) will be used to determine whether the input key is larger than the internal key. Specifically, we use a match-action table as shown in Figure 3, in which $sub[63 : 32]$ and 2 bits ($xor[63]$, $xor[31]$) are used as matching keywords, and the other 4 bits ($key[63]$, $key[31]$, $sub[63]$, $sub[31]$) are used as possible assignment results. For example, given key = 0x59A4737582A775AB and $pivot$ = 0xDDAEE7764589DB15, the match-action table uses the keywords $sub[63 : 32]$ = 0x840A7401, $xor[31] = 1$, $xor[63] = 1$ to determine that the fourth action needs to be executed, and then uses $key[63] = 0$ as the comparison result, which means that input key is not larger than pivot key. This method is not only applicable to the comparison between 64-bit data, but also can be easily extended to the comparison between data with higher bit-width.

Design of 16-branch regular nodes: In the programmable data plane with pipeline architecture, our system Fat-B$^+$Tree uses 6 stages to implement 16-branch regular nodes. Given n regular nodes and assuming that each node records 16 internal pivot keys, implementing these nodes requires 16 match-action tables (`get_pivot_i`) with n entries, 16 very small tables (`cal_result_i`), and a table `get_child_address` with $16 \times n$ entries. Specifically, the table `get_pivot_i` records the i-th internal pivot key of each node. This table uses the 64-bit address as the matching keyword, the corresponding action fetches the pivot key associated with the address, and obtains the temporary variables sub_i and xor_i. The table `cal_result_i` uses $sub_i[63 : 32]$, $xor_i[31]$, and $xor_i[63]$ as matching keywords, and obtains a 1-bit comparison result, which is recorded in $result[i]$. The table `get_child_address` records the addresses of $16 \times n$ child nodes of n regular nodes. This table uses the 64-bit initial address and the 16-bit comparison result $result[15 : 0]$ as matching keywords, and the corresponding action fetches 64-bit child node address. The comparison result is 16-bit data starting with a continuous 1 bit. For example, when $result[15 : 0]$ = 0b1111110000000000, the action will fetch the address of the sixth child node.

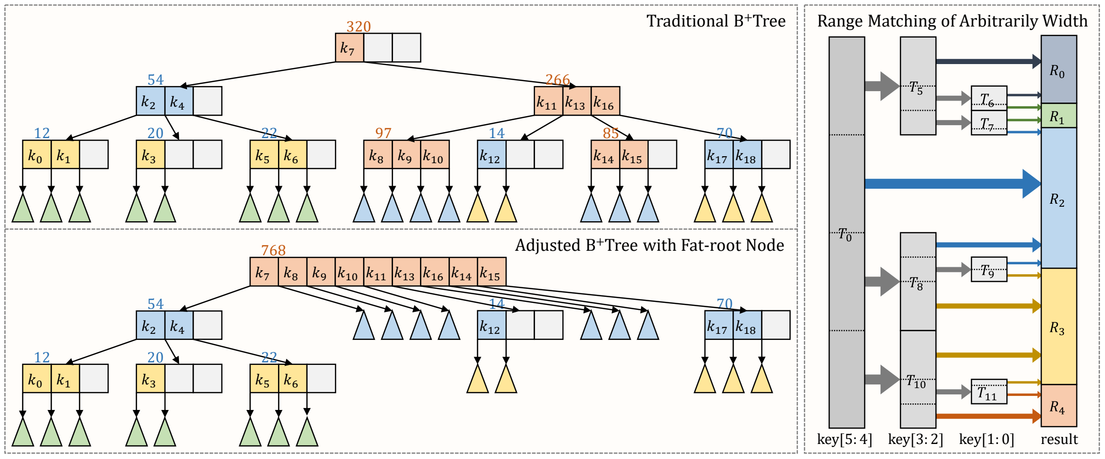

Traditional B+Tree
Adjusted B+Tree with Fat-root Node
Range Matching of Arbitrarily Width
key[5:4]
key[3:2]
key[1:0]
result

Figure 4: i) An example of reforming a traditional B$^+$tree into an adjusted B$^+$tree with a fat-root node, in which the numbers above the nodes represent the access frequencies. ii) An example of implementing range matching of high bit-width (*e.g.*, 6-bit) data through multiple low bit-width (*e.g.*, 2-bit) range matching tables.

B. *Fat-root Generation and B$^+$tree Reform*

Although Fat-B$^+$Tree can deploy many B$^+$tree nodes within 6 stages, each packet can only access one of these nodes due to the pipeline processing constraints of the data plane. Considering that for a B$^+$tree, each index operation will also access only one node in each layer, an intuitive idea is to deploy one layer of the B$^+$tree in every 6 stages of the data planes. However, this idea is not feasible due to the tree structure of B$^+$tree. The top layers of a B$^+$tree contain only a few nodes, which cannot make full use of the resources of the data plane. In contrast, the bottom layers of a B$^+$tree contain too many nodes, which cannot be fully deployed to the data plane due to resource constraints. In this section, we propose a technique called *fat-root generation* to reform the transitional B$^+$tree to alleviate the first issue.

As shown in Figure 4(i), the rationale of *fat-root generation* technique is to merge multiple nodes in the top layers of the B$^+$tree into a fat-root node, which makes the entire B$^+$tree "fat", that is, increases the number of nodes in the top layers of the B$^+$tree. Constructing a fat-root node with many internal pivot keys can not only make full use of the switch resources, but also reduce the number of access to nodes required by index operations. For example, if all the internal keys in the top 3 layers of the B$^+$tree are added to the fat-root node, each index operation can reduce the access to the nodes twice. The reason why this technique is not used in the B$^+$tree of the CPU platform is that containing too many internal keys will significantly increase the number of comparisons. For similar reasons, fat-root nodes cannot be implemented with SRAM in the data plane as regular B$^+$tree nodes, especially when a stage supports up to 16 tables. Fortunately, TCAM (ternary content addressable memory) resources and TCAM-based range matching tables in the data plane give us the possibility to find the correct one from a large number of ranges at line-rate. We first analyze which nodes should be selected to merge into fat-root node, and then introduce how to implement range matching of high bit-width (*e.g.*, 64-bit) data through low bit-width (*e.g.*, 16-bit) range matching tables provided by the programmable data plane.

The reason why we need to carefully select which nodes to merge into the fat-root node rather than randomly is that, from the perspective of reducing the number of access to nodes, the nodes are not equal. For example, suppose that the root node has only two child nodes, and 80% of the index operations pass through the left child node, while the remaining 20% of the index operations pass through the right child node. This skewed distribution is common due to the Zipf's law [30]. If we can only merge one node into the fat root node, merging the left child node can benefit more index operations, thus improving the overall index performance.

Based on this observation, we propose a greedy-based node selection algorithm. Assuming that we know the frequency of each node being accessed, we maintain a priority queue from the node with most access to the node with the least access. We start by adding the original root node to the queue. Every time a node is taken from the queue, we merge all the internal keys of this node into the fat-root node, and add all the child nodes of this node to the queue. This routine runs until the queue is empty or the size of the fat-root node reaches the limit K. If each node is full, this algorithm can find the optimal node set, that is, the set that minimizes the total number of access of nodes. An implementation example of the algorithm is shown in Figure 4(i). The algorithm finally selects the four nodes $\{k_7\}$, $\{k_{11}, k_{13}, k_{16}\}$, $\{k_8, k_9, k_{10}\}$, and $\{k_{14}, k_{15}\}$ with the highest access frequency, and merges them into a fat-root node $\{k_7, k_8, k_9, k_{10}, k_{11}, k_{13}, k_{14}, k_{15}, k_{16}\}$ containing 9 internal keys.

Implementation of fat-root node: Given a fat-root node containing $n - 1$ internal pivot keys and n child nodes, the index operation on the fat-root node is to find which of the n ranges the query key falls within. Since the Fat-B$^+$Tree supports 64-bit keys, the index operation depends on 64-bit range matching tables, while the programmable data plane generally only supports range matching table with a lower bit-width, such as no more than 20 bits. To address this issue, we propose Algorithm 1[2], a method to encode high bit-wide range into multiple low bit-wide range matching tables.

Algorithm 1: Routines for constructing range matching tables and generating table entries.

```
Func Construct_1 ($Ranges[1 \dots n]$):
    $r \leftarrow 1$;
    $next\_tab[0 \dots 2^{16} - 1] \leftarrow [0, \cdots, 0]$;
    $next \leftarrow n + 1$;
    for $i = 0 \rightarrow 2^{16} - 1$ do
        $range \leftarrow [i \ll 48, (i+1) \ll 48)$;
        while $range \cap Ranges[r] = \emptyset$ do
            $r \leftarrow r + 1$;
        if $range \subseteq Ranges[r]$ then
            $next\_tab[i] \leftarrow r$;
        else
            $next\_tab[i] \leftarrow next$;
            $next \leftarrow$
              Construct_2($Ranges, next, i \ll 48$);
    Generate_1($0, next\_tab$);
    return $next$;
Func Generate_1 ($tab, next\_tab[0 \dots 2^{16} - 1]$):
    $range \leftarrow \emptyset$;
    $next \leftarrow next\_tab[0]$;
    for $i = 0 \rightarrow 2^{16} - 1$ do
        if $next\_tab[i] = next$ then
            $range \leftarrow range \cup \{i\}$;
        else
            Table-add-entry($tab, range, next$);
            $range \leftarrow \{i\}$;
            $next \leftarrow next\_tab[i]$;
    Table-add-entry($tab, range, next$);
```

Taking 64-bit range and 16-bit range matching tables as an example, and assuming that the 64-bit key space is divided into n ranges $R_1, \cdots, R_n$, the algorithm first uses function `Construct_1` to construct a 16-bit range matching table to match the high 16 bits data $key[63 : 48]$ of query key key. The algorithm first traverses 2^{16} ranges $r_1, \cdots, r_{2^{16}}$ with a length of 2^{48}, where $r_i = [(i-1) \times 2^{48}, i \times 2^{48} - 1]$. If the range r_i is completely covered by the range R_j, the algorithm sets the matching result of the 16-bit range $[i, i]$ to R_j; If the range r_i intersects multiple ranges $R_{j_1}, \cdots, R_{j_m}$, the algorithm sets the matching result of the 16-bit range $[i, i]$ as another table T_i in the next stage. After getting the matching results of each 16-bit range, algorithm uses function `Generate_1` to merge the adjacent ranges with the same matching results, and obtain $O(n)$ matching table entries. The algorithm uses functions `Construct_2`, `Construct_3`, and `Construct_4` to recursively construct the tables in the next three stages, and uses them to match the remaining three parts $key[47 : 32]$, $key[31, 16]$, and $key[15, 0]$ of the query key. These three functions are similar to `Construct_1`. When deploying these matching tables in the data plane, multiple tables in a stage can be implemented as a large table, and each entry of this table contains an index of the original table and a 16-bit range.

[2]Due to space limitations, we only show the pesudo-code for functions `Construct_1` and `Generate_1`, the implementation of the other functions is similar.

Algorithm 1 only constructs n 16-bit matching tables at most in each stage, and the total number of entries in all matching tables in each stage does not exceed $2 \times n$. The computational complexity of Algorithm 1 is $O(2^{16} \times n + n^2)$. The complexity of algorithm can be optimized to $O(2^{16} + n)$, but the current version is efficient enough to complete the construction of tables in one second. Figure 4(ii) shows an example of implementing range matching of 6-bit data through eight 2-bit range matching tables. In this example, the 6-bit key space is divided into five ranges $R_1 = [0, 10]$, $R_2 = [11, 14]$, $R_3 = [15, 37]$, $R_4 = [38, 56]$, $R_5 = [57, 63]$. Algorithm 1 constructs 2-bit range matching tables in three stages. The first stage consists of 1 table (T_0) containing 4 table entries in total, the second stage consists of 3 tables (T_5, T_8, T_{10}) containing 9 table entries in total, and the third stage consists of 4 tables (T_6, T_7, T_9, T_{11}) containing 8 table entries in total.

C. *Layer-by-layer Node Caching*

As mentioned in Section III-A, deploying a B$^+$tree in the data plane faces two practical issues. The *fat-root generation* technique addresses the first issue that there are too few nodes in the top layers of the B$^+$tree to make full use of the resources in the data plane. In this section, we propose the *layer-by-layer node caching* technique to address the second issue that we cannot load all the nodes in the bottom layers of the B$^+$tree into the data plane.

Although we assume that Fat-B$^+$Tree can obtain the access frequency of each node in the B$^+$tree, the tree structure of the B$^+$tree prevents us from directly caching the nodes with the highest access frequency in each layer to the data plane, because if a node wants to be accessed, all its ancestors must be cached to the data plane. Based on this limitation, we propose a layer-by-layer node caching technique. Suppose we can cache up to K nodes in each layer to the programmable data plane. After reforming the B$^+$tree through the fat-root generation technique, we first add the K nodes with the highest access frequency among the children of the fat-root node to the first layer cache. Later, we select K nodes with the highest access frequency from the children of all nodes in the first layer cache, and add them to the second layer cache, although the access frequency of these nodes may be lower than that of some other nodes in the same layer. We construct each subsequent layer of cache according to the same routine.

D. *Query Encapsulation for RoCE*

After deploying the B$^+$tree into the data plane, the remaining issue is how to fetch the query key from the packet and fill in the query address. If the client and server communicate through TCP/IP, we can flexibly customize the application layer protocol. However, the current communities are keen to introduce RDMA into the database to improve the performance. For the traditional B$^+$tree, the client first uses the RDMA-read to fetch the root node of the B$^+$tree, then obtains the address of the child node locally through comparison, and then gradually fetches the nodes through the RDMA-read, and finally fetches the data. The introduction of RDMA brings challenges to Fat-B$^+$Tree, because the data packet of RDMA-read has no application layer load, and the fields that the sender can modify are only three files in the RDMA Extended Transport Header: 64-bit va, 32-bit $rkey$, and 32-bit $length$.

We design a query information encapsulation mechanism in the data plane to solve this challenge. We divide the switches participating in Fat-B$^+$Tree into three types: the first-hop switch, the intermediate switch, and the last-hop switch. For each index or query operation, the client construct a *special* RDMA-read by setting the 64-bit va filed as the query key, the 32-bit $rkey$ filed as 0, and the $length$ field as the size of the B$^+$tree node. The first-hop switch identifies special operators according to whether $rkey$ is 0, attaches a temporary header to record the query key originally stored in the va, performs in-network index to obtain the query address, and fills the address in the va field. Intermediate switches all perform the index according to the query key in the temporary header and the va filed, and update the va with the result. The last-hop switch needs to remove the temporary header. The server takes out the B$^+$tree node pointed to by the query address filled by the switch, and sends the data back to the client. After receiving an internal node of the B$^+$tree obtained by the *special* RDMA-read, the client can use the conventional logic to obtain the subsequent child nodes trough the *normal* RDMA-read until the final data.

IV. Implementation

A. Network-wide Deployment

Deployment on a single switch: As mentioned in Section III-B and III-A, we design two types of B$^+$tree nodes, namely the fat-root node based on TCAM, and the 16-branch regular node based on SRAM. We fully implement the fat-root node supporting up to 1024 child nodes and the node set supporting up to 4096 16-branch regular nodes in the commercial programmable switch. The implementation of fat-root node and regular node set only requires 6 consecutive stages, while the commercial programmable data plane provides at least 12 stages. Therefore, we can implement a fat-root node and a regular node set in a switch, or two regular node sets in a switch, where each regular node set contains some cached nodes in the same layer of the B$^+$tree.

Deployment on multiple switches: Take the data center network with the FatTree topology with k pods as an example. To deploy our Fat-B$^+$Tree in this network, we first arbitrarily select a core switch S_c. Switch S_c is connected with k aggregation switches $S_{a_1}, \cdots, S_{a_k}$ in the k pods. Switch S_{a_i} is connected with $\frac{k}{2}$ edge switches $S_{e_{i,1}}, \cdots, S_{e_{i,k/2}}$ in the pod, and these edge switches connect the servers in their respective racks. For the distributed database, data is sequentially stored in multiple memory servers, and each memory server maintains its own B$^+$tree. To build a unified B$^+$tree in the data plane, we only need to build a super-root node with the root node of each B$^+$tree as the child node, and add this super-root node as the initial node to the queue to generate a fat-root node. *Core switch:* We deploy the fat-root node of the unified B$^+$tree in the core switch S_c. In addition to giving the address of the child node corresponding to each range, the fat-root node also needs to give the memory server ID corresponding to the data in each range, and the memory server ID is carried in the temporary header mentioned in Section III-D. Switch S_c forwards the packet to one of the k aggregation switches according to the server ID. *Aggregation switches:* We deploy two regular node sets in the aggregation switch S_{a_i}. Switch S_{a_i} forwards the packet to one of the $\frac{k}{2}$ edge switches according to the server ID. *Edge switches:* We deploy two regular node sets in the edge switch $S_{e_{i,j}}$. Switch $S_{e_{i,j}}$ forwards the packet to one of the $\frac{k}{2}$ servers according to the server ID.

B. Read-write Mixed Workload

We currently adopt a lazy B$^+$tree structure update mechanism to handle the read-write mixed workload. Since the insertion/deletion operations change the B$^+$tree structure from the leaf node and affect the parent node layer by layer, once we need to modify an internal node with a memory address of r and its parent node is loaded into the data plane, we regard it as a temporary root node rather than an internal node. The subtree with this node continues to take this node as the root, or a newly generated node whose memory address is still r as the root, without affecting the loaded parent node.

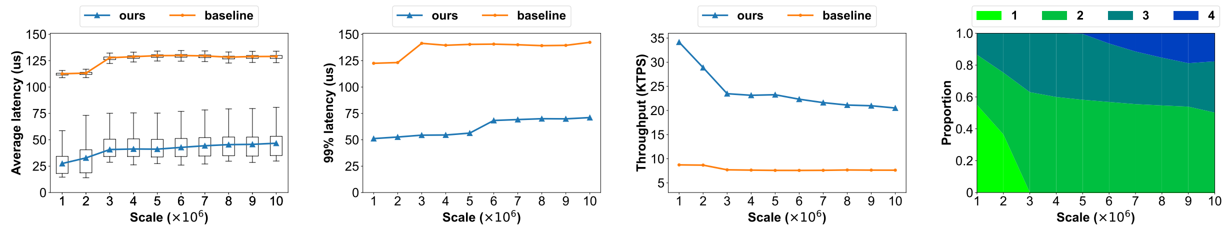
C. Parallel Accelerated Range Query

When performing a range query for the range $[l, r]$, the traditional B$^+$tree needs to first query the minimum data falling within the range, and then continuously traverse the data through the linked list between leaf nodes. A simple approach to accelerate range query is to divide the range $[l, r]$ into n consecutive sub-ranges $[l_1 = l, r_1], \cdots, [l_n, r_n = r]$, and perform sub-range queries in parallel. However, for traditional B$^+$tree index, this approach will bring significant index overhead. For example, if the range contains 100 valid keys, assuming that each indexing requires 7 memory access, dividing the original range into 10 sub-ranges will bring $59\%(170/107)$ additional overhead. With Fat-B$^+$Tree, assuming that each indexing can perform 5 memory access in the data plane, and requires only 2 memory access in the servers, the additional overhead can be reduced to $18\%(120/102)$. Therefore, using parallel-accelerated range query in Fat-B$^+$Tree can significantly reduce query latency at the cost of only sightly affecting throughput.

V. Experiment

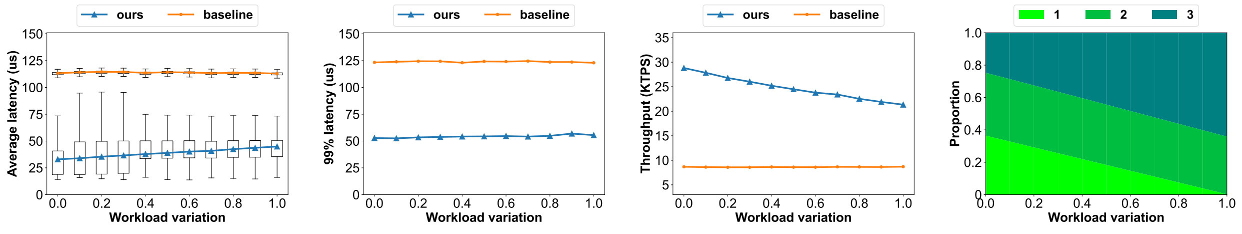
A. Experimental Setup

Benchmark: We evaluate our system Fat-B$^+$Tree through the open source YCSB benchmark [31], [32]. We use the workload provided by YCSB as the computing client, use the B$^+$tree written by about 1000 lines of C++ code as the memory server, and use the communication protocol written by about 1100 lines of DPDK code [33]. We deploy the programmable data plane program according to the mechanism proposed in Section IV-A, which contains about 1300 lines of P4 code and 1000 lines of control plane code.



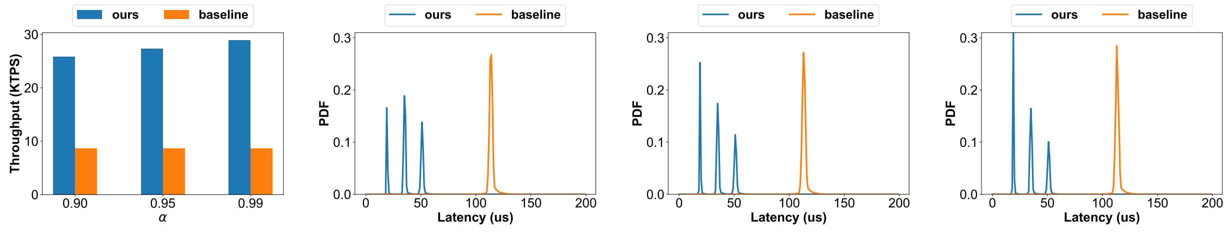
(a) Average latency (b) 99% latency (c) Throughput (d) # round trips

Figure 5: The impact of database scale on performance.



(a) Average latency (b) 99% latency (c) Throughput (d) # round trips

Figure 6: The impact of query workload variation on performance.



(a) Throughput (b) Latency ($\alpha = 0.90$) (c) Latency ($\alpha = 0.95$) (d) Latency ($\alpha = 0.99$)

Figure 7: The impact of query workload distribution on performance.

Hardware platform: We evaluate our system Fat-B$^+$Tree in RDMA network and Ethernet respectively. For RDMA network, we emulate memory disaggregation architecture. We use a server to emulate the computing client, and another server to emulate the memory server, each of which is configured with a 4.2GHz CPU, 128GB DRAM, and a Mellanox ConnectX-5 NIC. These servers are connected through two Tofino programmable switches. For Ethernet, we emulate client-server architecture. We use one server to emulate the computing client, and four servers to emulate memory servers, each of which is configured with a 2.1GHz CPU, 256GB DRAM, and a Mellanox ConnectX-3 NIC. These servers are located in a data center network with $K = 4$ FatTree topology. Four memory servers are in one pod, while the computing client is in another pod. All devices are connected through 40Gbps cables.

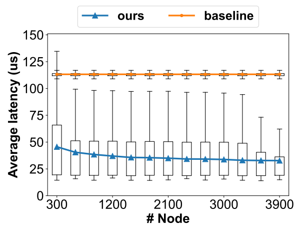
Baseline systems: For the RDMA network, in the baseline system, the computing client fetches the root node of the B$^+$tree remotely through the RDMA-read operator, and continuous to fetch the subsequent nodes until the data; For the Ethernet, in the baseline system, the memory servers are responsible for performing the indexing starting from the root of the B$^+$tree.

B. *Experiments on RDMA Network*

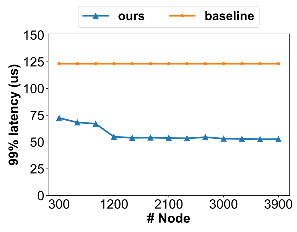
Default setting: In this section, we set the database scale to 2×10^6 keys, set the length of each value to 1000 bytes, set a single thread to emulate the computing client, and set the workload to be read-only (YCSB workload C), subject to the zipfian distribution of $\alpha = 0.99$. We load a fat-root node with a maximum of 600 ranges and three subsequent layers with a maximum of 3600 nodes into the data plane of two switches.

Performance *v.s.* database scale (Figure 5): We vary the database scale from 1×10^6 to 1×10^7, and evaluate the performance of our Fat-B$^+$Tree and the baseline. The average query latency of our Fat-B$^+$Tree varies from 27.5us to 46.7us, while that of the baseline varies from 112.6us to 129.0us. The 99% query latency of our Fat-B$^+$Tree varies from 51.1us to 71.0us, while that of the baseline varies from 122.5 to 142.4. The throughput of our Fat-B$^+$Tree varies from 34.2KTPS to 20.5KTPS, while that of the baseline varies from 8.7KTPS to 7.6KTPS. In summary, the latency of our Fat-B$^+$Tree can be reduced by up to 76%, and the throughput of our Fat-B$^+$Tree can be improved by up to 2.93 times. As shown in Figure 5(d), for Fat-B$^+$Tree, we observe two critical points 3×10^6 and 7×10^6. When the scale is 3×10^6, the total depth of the B$^+$tree is increased by one layer, so no query can fetch results after a single round trip. When the scale is 5×10^6, the subsequent layer of fat-root cannot be fully loaded, resulting in some queries that require four round trips. These two points can correspond to the change of the curve in Figure 5(a)-5(c)

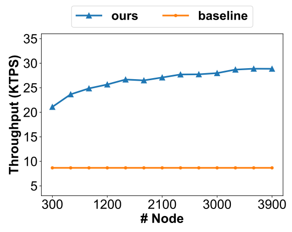
Performance *v.s.* workload variation (Figure 6): We make a difference between the query key distribution used to generate



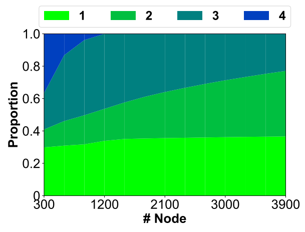
(a) Average latency



(b) 99% latency

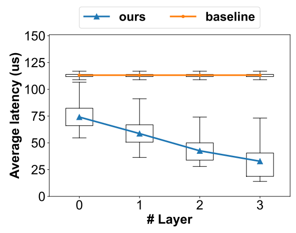


(c) Throughput

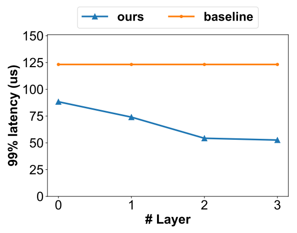


(d) # round trips

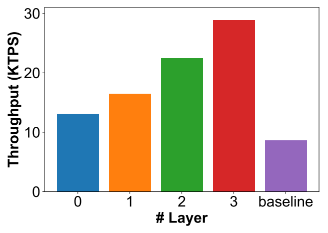
Figure 8: The impact of the number of loaded B$^+$tree nodes on performance.



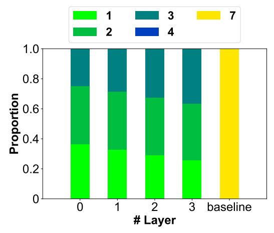
(a) Average latency



(b) 99% latency



(c) Throughput



(d) # round trips

Figure 9: The impact of the number of loaded B$^+$tree layers on performance.

the fat-root and the cached nodes and that used to perform the performance experiment, and evaluate the impact of the degree of variation on the performance. The average query latency of our Fat-B$^+$Tree varies from 32.9us to 44.9us, while that of the baseline is around 113us. The 99% query latency of our Fat-B$^+$Tree varies from 52.7us to 55.5us, while that of the baseline is around 123us. The throughput of our Fat-B$^+$Tree varies from 28.8KTPS to 21.3KTPS, while that of the baseline is around 8.7KTPS. In summary, the latency of our Fat-B$^+$Tree can be reduced by up to 71%, and the throughput of our Fat-B$^+$Tree can be improved by up to 2.31 times. As expected, all performance indicators change linearly, and even if the query key distribution of workload is completely changed, Fat-B$^+$Tree can still play a significant acceleration effect, but no query can be completed after a single round trip.

Performance *v.s.* workload distribution (Figure 7): We evaluate the performance of our Fat-B$^+$Tree and the baseline with zipfian distribution [30] workloads with parameters $\alpha = 0.9$, $\alpha = 0.95$, and $\alpha = 0.99$. The throughput of our Fat-B$^+$Tree varies from 25.8KTPS to 28.8KTPS, while that of the baseline is around 8.7KTPS. We also show the query latency distribution under different workload distributions. As shown in Figure 7(b)-7(d), Fat-B$^+$Tree performs better under a more skewed workload, which allows more queries to be completed after fewer round trips.

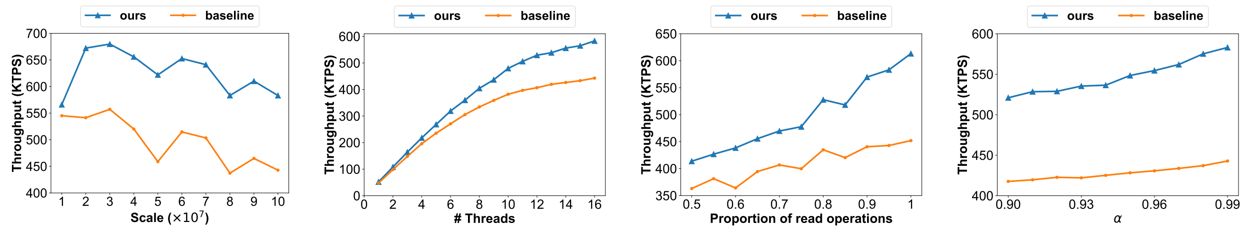
Performance *v.s.* the number of loaded nodes (Figure 8): We set the fat-root to contain a maximum of 600 nodes, and vary the maximum number of nodes that the subsequent layers can contain from 300 to 3900, and evaluate the performance of our Fat-B$^+$Tree and the baseline. The average query latency of our Fat-B$^+$Tree varies from 45.6us to 32.7us, while that of the baseline is 113.2us. The 99% query latency of our Fat-B$^+$Tree varies from 72.5us to 52.7us, while that of the baseline is 123.2us. The throughput of our Fat-B$^+$Tree varies from 21.1KTPS to 28.9KTPS, while that of the baseline is 8.7KTPS. In summary, the latency of our Fat-B$^+$Tree can be reduced by up to 71%, and the throughput of our Fat-B$^+$Tree can be improved by up to 2.32 times.

Performance *v.s.* the number of loaded layers (Figure 9): We loaded the fat-root fixedly, and vary the number of loaded subsequent layers from 0 to 3, and evaluate the performance of our Fat-B$^+$Tree and the baseline. The average query latency of our Fat-B$^+$Tree varies from 74.1us to 32.8us, while that of the baseline is 113.2us. The 99% query latency of our Fat-B$^+$Tree varies from 88.4us to 52.6us, while that of the baseline is 123.2us. The throughput of our Fat-B$^+$Tree varies from 13.1KTPS to 28.9KTPS, while that of the baseline is 8.7KTPS. In summary, the latency of our Fat-B$^+$Tree can be reduced by up to 71%, and the throughput of our Fat-B$^+$Tree can be improved by up to 2.33 times.

C. *Experiments on Large-scale Ethernet*

Default setting: In this section, we set the database scale to 1×10^8 keys, set the length of each value to 64 bytes, set 16 threads to emulate the 16 computing clients, and set the workload to 90% read and 10% write (modified YCSB workload B), subject to the zipfian distribution of $\alpha = 0.99$. We load a fat-root node with a maximum of 600 ranges and four subsequent layers with a maximum of 3600 nodes into the data planes (§IV-A).

Throughput *v.s.* database scale (Figure 10(a)): We vary the database scale from 1×10^7 to 1×10^8, and evaluate the performance of our Fat-B$^+$Tree and the baseline. The throughput of our Fat-B$^+$Tree has the maximum value of 679.8KTPS and the minimum value of 566.0KTPS, while those of the baseline is 557.1KTPS and 437.2KTPS. In summary, the throughput of our Fat-B$^+$Tree can be improved by up to 1.35 times. A critical point occurs when the scale is 2×10^7. This is because the resources of the switches are not fully used when the scale is 10^7, and the depth of the B$^+$tree is increased by one layer when the scale is 2×10^7, which makes full use of the resources of the switches to achieve higher performance.



(a) Throughput *v.s.* scale (b) Throughput *v.s.* # threads (c) Throughput *v.s.* read/write ratio (d) Throughput *v.s.* workload

Figure 10: The adaptability and scalability of the system in the Ethernet scenario.

Throughput *v.s.* the number of clients (Figure 10(b)): We vary the number of computing clients from 1 to 16, and evaluate the performance of our Fat-B$^+$Tree and the baseline. The throughput of our Fat-B$^+$Tree changes from 52.0KTPS to 583.1KTPS, while that of the baseline changes from 47.7KTPS to 442.8KTPS. In summary, the throughput of our Fat-B$^+$Tree can be improved by up to 1.32 times. It can be observed that the throughput growth rate of the baseline begin to slow down earlier than that of the Fat-B$^+$Tree.

Throughput *v.s.* read/write ratio (Figure 10(c)): We vary the read/write ratio from $0.5 : 0.5$ to $1 : 0$, and evaluate the performance of our Fat-B$^+$Tree and the baseline. The throughput of our Fat-B$^+$Tree changes from 613.2KTPS to 413.7KTPS, while that of the baseline changes from 451.9KTPS to 363.0KTPS. In summary, the throughput of our Fat-B$^+$Tree can be improved by up to 1.35 times.

Throughput *v.s.* workload distribution (Figure 10(d)): We vary the workload distribution from $\alpha = 0.9$ to $\alpha = 0.99$, and evaluate the performance of our Fat-B$^+$Tree and the baseline. The throughput of our Fat-B$^+$Tree changes from 521.0KTPS to 583.1KTPS, while that of the baseline changes from 417.7KTPS to 442.8KTPS. In summary, the throughput of our Fat-B$^+$Tree can be improved by up to 1.32 times. Unlike the baseline based on RDMA network, the baseline based on Ethernet can also benefit from a more skewed workload due to the existence of system cache.

VI. Related Work

Distributed tree index: Typical solutions include Cell [8], Sherman [20], and more [17], [22]. Cell [8] distributes a global B-tree of meganodes across machines for server-side searches, and organizes keys as a local B-tree of RDMA-friendly small nodes for client-side searches within each meganode. Sherman [20] proposes a write-optimized B$^+$-tree on disaggregated memory by combining RDMA hardware features and RDMA-friendly software techniques. To reduce remote access in the tree traversal, it caches and maintains the highest two level of nodes, and the nodes above the leaf nodes. The latency of Sherman is poor in the worst case: once the cache misses, it requires to traverse the B$^+$-tree from the high level.

In-network caching: SwitchKV [26] route queries to the right cache node to move the in-memory cache out of the data path to achieve high performance. NetCache [23] deploys a key-value cache on programmable data plane to cache hottest items. On the basis of NetCache, NetChain [24] builds a strongly-consistent, fault-tolerant key-value store in the switch data plane that provides scale-free sub-RTT coordination. DistCache [25] is a distributed caching mechanism that provides provable load balancing for large-scale storage systems. KV-Direct [34] chooses to offload the key-value operations to the programmable NIC with FPGA. In summary, none of the above efforts propose to cache the tree structure in-network.

RDMA on programmable switches: TEA [35] implements RDMA in the data plane to enable the programmable switches to access external DRAM without involving CPUs. SwitchML [36] implements a subset of RDMA in the switch, and uses RDMA to allow data to move directly between the switch and GPUs. Ribosome [37] use a programmable switch to delegate the packet payloads on RDMA servers by using RDMA write message, and only forward the headers to NF servers. Some works offload RDMA to other customized devices *etc.* [38]–[44].

VII. Conclusion

In this paper, we propose Fat-B$^+$Tree, a system that embeds part of the B$^+$tree into the hierarchical connected programmable switches in the data center network. To extend the ability of programmable data plane, we design SRAM-based regular node and TCAM-based fat-root that support 64-bit keys, we design a fat-root generation algorithm and a layer-by-layer node caching algorithm, we also design an in-network query information encapsulation mechanism for RDMA network. Fat-B$^+$Tree simultaneously meets all three design requirements: (**efficiency**) it can provide a large amount of fast on-chip memory, thus significantly reducing the indexing latency of B$^+$tree, (**compatibility**) it can support common operations of B$^+$tree and both client-server and memory disaggregation architectures, and (**adaptability**) it can always reduce the indexing latency under any workload and any scale. We have fully implemented the Fat-B$^+$Tree prototype, and the experimental results show that compared with the baseline system, it reduces query latency by up to 76% and improves throughput by up to 3.93 times. The source codes of Fat-B$^+$Tree are open-sourced at GitHub [29].

Acknowledgment

Tong Yang (yangtongemail@gmail.com) is the corresponding author. This work is supported by National Key R&D Program of China (No. 2022YFB2901504), and National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009, 623B2005).

REFERENCES

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX NSDI 12*, pp. 15–28, 2012.

[2] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1816–1827, 2015.

[3] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et al.*, "Scaling memcache at facebook," in *USENIX NSDI 13*, pp. 385–398, 2013.

[4] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, pp. 295–306, 2014.

[5] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 183–196, 2012.

[6] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu, and B. T. Loo, "Understanding the effect of data center resource disaggregation on production dbmss," *Proceedings of the VLDB Endowment*, vol. 13, no. 9, 2020.

[7] T. Lahiri, M.-A. Neimat, and S. Folkman, "Oracle timesten: An in-memory database for enterprise applications.," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 6–13, 2013.

[8] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li, "Balancing {CPU} and network in the cell distributed {B-Tree} store," in *USENIX ATC 16*, pp. 451–464, 2016.

[9] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM ASPLOS Conference*, pp. 79–92, 2021.

[10] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," in *Proceedings of the 27th ACM ASPLOS Conference*, pp. 417–433, 2022.

[11] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "Mind: In-network memory management for disaggregated data centers," in *Proceedings of the ACM SIGOPS 28th SOSP Conference*, pp. 488–504, 2021.

[12] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, "Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 868–880, IEEE, 2020.

[13] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, "Semeru: A memory-disaggregated managed runtime," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pp. 261–280, 2020.

[14] M. Zhang, Y. Hua, P. Zuo, and L. Liu, "{FORD}: Fast one-sided {RDMA-based} distributed transactions for disaggregated persistent memory," in *USENIX FAST 22*, pp. 51–68, 2022.

[15] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu, and B. T. Loo, "Understanding the effect of data center resource disaggregation on production dbmss," *Proceedings of the VLDB Endowment*, vol. 13, no. 9, 2020.

[16] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes, "Building an elastic query engine on disaggregated storage.," in *USENIX NSDI*, vol. 20, pp. 449–462, 2020.

[17] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska, "Designing distributed tree-based index structures for fast rdma-capable networks," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 741–758, 2019.

[18] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 202–215, 2016.

[19] J. Backus, "Can programming be liberated from the von neumann style? a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[20] Q. Wang, Y. Lu, and J. Shu, "Sherman: A write-optimized distributed b+ tree index on disaggregated memory," in *Proceedings of the 2022 International Conference on Management of Data*, pp. 1033–1048, 2022.

[21] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *USENIX NSDI 14*, pp. 401–414, 2014.

[22] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, and M. Castro, "Fast general distributed transactions with opacity," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 433–448, 2019.

[23] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th SOSP Conference*, pp. 121–136, 2017.

[24] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "{NetChain}:{Scale-Free}{Sub-RTT} coordination," in *USENIX NSDI 18*, pp. 35–49, 2018.

[25] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "{DistCache}: Provable load balancing for {Large-Scale} storage systems with distributed caching," in *USENIX FAST 19*, pp. 143–157, 2019.

[26] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with {SwitchKV}," in *USENIX NSDI 16*, pp. 31–44, 2016.

[27] I. T. Association *et al.*, "Supplement to infiniband architecture specification volume 1 release 1.2.1 annex a16: Rdma over converged ethernet (roce)," 2010.

[28] I. T. Association *et al.*, "Supplement to infiniband architecture specification volume 1 release 1.2.1 annex a17: Rocev2," 2014.

[29] "All related codes of our Fat-B$^+$Tree." https://github.com/Fat-BTree/Fat-BTree.

[30] D. M. Powers, "Applications and explanations of zipf's law," in *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*, Association for Computational Linguistics, 1998.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010.

[32] "Yahoo! cloud serving benchmark in c++, a c++ version of ycsb.." https://github.com/basicthinker/YCSB-C.

[33] "Data plane development kit.." http://doc.dpdk.org/guides-18.02/.

[34] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proceedings of the 26th SOSP Conference*, pp. 137–152, 2017.

[35] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "Tea: Enabling state-intensive network functions on programmable switches," in *Proceedings of the ACM SIGCOMM 2020 Conference*, pp. 90–106, 2020.

[36] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," in *USENIX NSDI 21*, pp. 785–808, 2021.

[37] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostic, and M. Chiesa, "A high-speed stateful packet processing approach for tbps programmable switches," in *USENIX NSDI 23*, 2023.

[38] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartnics using ipipe," in *Proceedings of the ACM Special Interest Group on Data Communication*, pp. 318–333, 2019.

[39] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism," in *Proceedings of the ACM SIGOPS 28th SOSP Conference*, pp. 756–771, 2021.

[40] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *USENIX NSDI 18*, pp. 51–66, 2018.

[41] M. Burke, S. Dharanipragada, S. Joyner, A. Szekeres, J. Nelson, I. Zhang, and D. R. Ports, "Prism: Rethinking the rdma interface for distributed systems," in *Proceedings of the ACM SIGOPS 28th SOSP Conference*, pp. 228–242, 2021.

[42] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy, "Xenic: Smartnic-accelerated distributed transactions," in *Proceedings of the ACM SIGOPS 28th SOSP Conference*, pp. 740–755, 2021.

[43] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso, "Strom: smart remote memory," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.

[44] J. Langlet, R. B. Basat, S. Ramanathan, G. Oliaro, M. Mitzenmacher, M. Yu, and G. Antichi, "Direct telemetry access," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023.