Catfish: Adaptive RDMA-enabled R-Tree for Low Latency and High Throughput

Mengbai Xiao†, Hao Wang†, Liang Geng†§, Rubao Lee‡, Xiaodong Zhang†
†Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
{xiao.736, wang.2721, geng.161}@osu.edu, zhang@cse.ohio-state.edu
§Department of Computer Science and Engineering, Northeastern University, China
‡United Parallel Computing Corporation, DE, USA, lirb@unipacc.com

***Abstract*—R-tree is a foundational data structure used in spatial databases and scientific databases. With the advancement of Internet and computer architectures, in-memory data processing for R-tree in distributed systems has become a common platform. We have observed new performance challenges to process R-tree as the amount of multidimensional datasets become increasingly huge. Specifically, an R-tree server can be heavily overloaded while the network and client CPU are lightly loaded, and vice versa.**

In this paper, we present the design and implementation of Catfish, an RDMA enabled R-tree for low latency and high throughput by adaptively utilizing the available network bandwidth and computing resources to balance the workloads between clients and servers. We design and implement two basic mechanisms of using RDMA for the client-server R-tree. First, in the fast messaging design, we use RDMA writes to send R-tree requests to the server and let server threads process R-tree requests to achieve low query latency. Second, in the RDMA offloading design, we use RDMA reads to offload tree traversal from the server to the client, which rescues the server as it is overloaded. We further develop an adaptive scheme to effectively switch an R-tree search between fast messaging and RDMA offloading, maximizing the overall performance. Our experiments show that the adaptive solution of Catfish on InfiniBand significantly outperforms R-tree that uses only fast messaging or only RDMA offloading in both latency and throughput. Catfish can also deliver up to one order of magnitude performance over the traditional schemes using TCP/IP on 1 Gbps and 40 Gbps Ethernet. We make a strong case to use RDMA to effectively balance workloads in distributed systems for low latency and high throughput.

I. Introduction

R-tree [1] is a fundamental data structure for storing and querying multidimensional data like rectangles and polygons, which are the essential data representations in scientific databases, spatial databases, and big data systems. In production systems, e.g., Google Maps [2], Yelp [3], and others, front-end web servers accept user requests such as "Search this area" and "find restaurants near me" from Internet, and send the spatial queries to back-end servers hosting an R-tree data structure. Figure 1 presents a typical system infrastructure in the client-server mode, where users send the requests of "searching nearby restaurants" via Web servers, and queries are processed in the back-end server with R-tree. Our observations on representative R-tree processing in the real-world motivating us for this work are described as follows.

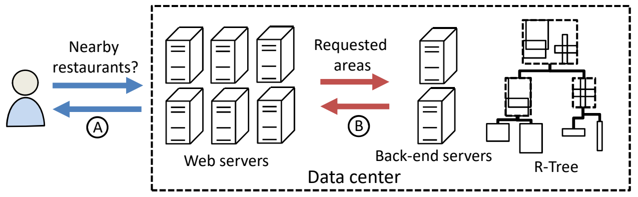
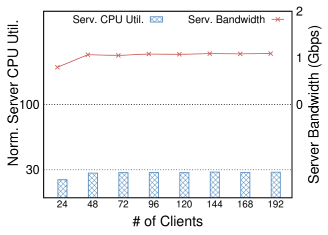
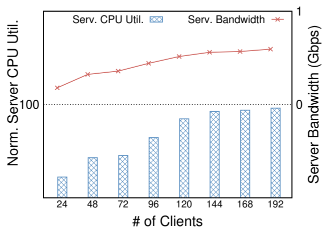


Fig. 1: An example of accessing spatial data in an R-tree, where Ⓐ represents an Internet connection and Ⓑ represents the intra-datacenter connection.

We carry out experiments on a small cluster to identify the locations of bottlenecks in the scenario shown in Figure 1. The experimental cluster uses 1 Gbps Ethernet to connect multiple compute nodes (Detailed setups are introduced in Section V). In the experiments, multiple clients send requests to a server and the server is responsible for searching the R-tree and returning the results. On the single server, an R-tree is pre-built with 2 million 2D rectangles, whose edges and locations are normalized in $[0, 1]$, which means that a square with edges equaling 1 covers the whole space. The clients launch independent threads (from 2 to 32) that continuously send 10,000 search requests to the server. The requested rectangles are designated with randomly generated locations and all overlapped rectangles in the R-tree are expected to be returned.We set various upper bounds over the edges of requested rectangles for simulating different types of searching workload. The experimental results are illustrated in Figure 2, in which the x-axis is the number of clients, the left y-axis is the normalized server CPU utilization and the right y-axis represents server bandwidth in Gbps. Figure 2(a) presents the results when setting the upper bound of requested rectangles to 0.01. In this case, the numbers of overlapped rectangles found in the R-tree are very large, and the bandwidth of server is easily saturated while only up to 28% server CPU cycles are used. This occurs when a user wants to monitor relevant objects in a large range of area, like how many properties would be impaired in an area that a hurricane would pass. Figure 2(b) shows the results when setting the edge upper bound of requested rectangles to 0.00001. Only few intersected rectangles are found in the R-tree. Until the server CPU utilization has been pushed up to 100%, only 65.8% bandwidth



(a) R-tree search with bandwidth bound



(b) R-tree search with CPU bound

Fig. 2: The normalized server CPU utilization and server bandwidth measured in different R-tree search workloads.

is consumed. Such a query in a small scope often happens, e.g., searching nearby restaurants or gas stations in Google Maps.

Our experiments indicate that when accessing a memory-resident R-tree, both the CPU and network bandwidth of the server could be saturated by highly simultaneous requests, becoming the performance bottlenecks. And such bottlenecks will be further aggravated by skew access patterns in real workloads [4]. The bottlenecks cannot be easily solved by updating hardware. For example, changing the network to 40 Gbps Ethernet does not help in the CPU-bound case as shown in Figure 2(b). Instead, the server CPU will be saturated more quickly as more R-tree requests will arrive at the server with the increased network bandwidth.

Bottlenecks of network and server CPU are observed phenomenon. However, having looked into the entire workload execution process, we identify three opportunities: 1) CPUs on the client side are often lightly loaded. 2) When server is heavily loaded, network is often lightly loaded. 3) When network is saturated, server may not be overloaded. These opportunities motivate us to develop a new system mechanism to balance the workloads between client and server by using available network bandwidth and idle CPU cycles on the client side, aiming for low latency and high throughput. Remote Direct Memory Access (RDMA) is an effective facility to help us to achieve this goal.

RDMA is widely deployed in data centers with modern interconnects. As opposed to the *send* and *recv* operations in TCP/IP, an RDMA node is able to directly *read* (RDMA Read [5]–[11]) and *write* (RDMA Write [12]–[14]) a remote address registered in RDMA hardware with higher bandwidth (100 - 290 Gbps) and lower latency (a few microseconds), and more importantly, without interrupting remote CPUs. As a result, the RDMA-based R-tree is a promising solution that can overcome the identified bottlenecks, especially the CPU-bound case. However, there are several challenges to directly apply RDMA Read/Write on R-tree. First, both the requests and responses could be delivered via RDMA Write, but this requires the server-side CPU to be aware of the incoming request and be involved with request processing. Thus, it cannot handle the CPU bound cases of R-tree search as shown in Figure 2 (b). Second, by using RDMA Read a client can directly read data in remote memory and the workload is offloaded from server to client. This method is able to free server-side CPU cycles, but it incurs multiple RDMA Read round trips in R-tree traversal, making the query latency unacceptable.

To address these issues, we propose Catfish, a distributed R-tree on RDMA that can adaptively use RDMA Write and RDMA Read to build a high performance R-tree. The reason we name our system as Catfish comes from a consideration for its strong adaptability, which is the goal in our design and implementation. Catfish has two unique advantage for its survival: 1) high skin sensitivity to timely detect dynamic changes in the environment and 2) high flexibility in turning its body to adopt the changes. Our Catfish system is outlined as follows. First, we implement the RDMA-based R-tree, where the read requests, i.e., searching a rectangle, are completed in either RDMA Write or RDMA Read. For R-tree write requests, such as insert, update, delete, and others, the RDMA-Write-based solution are always used. We refer to the scheme using RDMA Write for R-tree reads as *fast messaging* and the other one as *RDMA offloading*. Second, we observe that when server-side CPUs are not saturated, fast messaging is highly effective, since it only needs one RDMA round-trip time (RTT) and sever-side CPUs handle R-tree traversal with several local memory accesses; while, when the server is overloaded but with abundant bandwidth resources, RDMA offloading is more effective, since it can offload R-tree traversal to clients and utilize client-side CPUs. Therefore, we design a heuristic coordination mechanism to make every single client to determine its own R-tree access method autonomously. A client can adaptively switch to the best execution mode between fast messaging and RDMA offloading at runtime. Third, we also enhance fast messaging and RDMA offloading, respectively. We change to the event-driven mode on R-tree server to avoid CPU oversubscription, significantly improving the performance of fast messaging. We also overlap network RTTs of RDMA Read in RDMA offloading by a multi-issue technique: when traversing an R-tree, the client simultaneously sends multiple RDMA reads to query all valid child nodes.

We carry out our experiments on a cluster having 1Gbps Ethernet, 40Gbps Ethernet, and 100Gbps InfiniBand connections. For the workloads composed of 100% search requests, Catfish delivers up to 3.28x, 3.09x and 16.46x speedups of throughput and 3.25x, 3.07x, and 24.46x reductions of request latency over fast messaging, RDMA offloading, and TCP/IP-based schemes, respectively. Similar performance gains are also observed with the skewed search requests and the hybrid workloads having both R-tree search and insert requests. This paper makes the following contributions.

1) To the best knowledge of ours, this is the first design and implementation of R-tree on RDMA with high performance.
2) We propose an adaptive coordination mechanism to enable autonomous switch between fast messaging and RDMA offloading in clients, achieving the best overall performance.
3) We carry out extensive experiments to demonstrate the effectiveness of our designs, particularly the adaptive scheme. We make a strong case to use RDMA to effectively balance workloads in distributed systems for high performance.

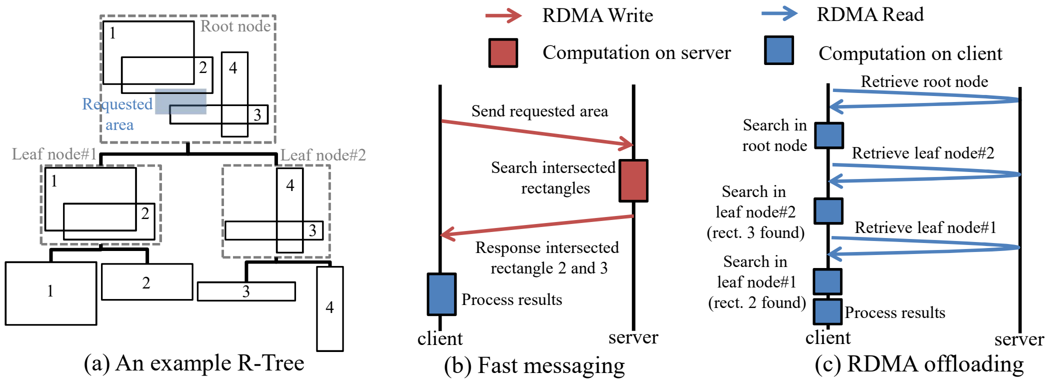


Fig. 3: The illustration of an R-tree organization, and how fast messaging and RDMA offloading complete a spatial query on the R-tree in the client-server mode.

II. Background and Motivation

A. R-Tree

An R-tree is a height-balanced tree containing spatial objects, e.g., rectangles and polygons, in its leaf nodes. Other than leaf nodes, the root node or an internal node of an R-tree includes the Minimum Bounding Rectangles (MBRs), or bounding boxes, of its child nodes. An MBR is the smallest rectangle that encloses a set of rectangles. In this work, we store 2-dimensional rectangles in leaf nodes, and each rectangle has four double precision floating-point variables to represent its coordinates, as min(x), max(x), min(y), and max(y). Figure 3 (a) gives an example of a two-level R-tree having four rectangles, and the dashed boxes are the MBRs.

An R-tree provides the basic index operations, e.g., search and insert. The *search* operation traverses all internal nodes whose MBRs intersect with the given rectangle until the leaf nodes containing overlapped rectangles are found or there is no such leaf node. Multiple search paths and qualified leaf nodes may exist. In Figure 3 (a), for the shadow box that represents the request rectangle, there are two search paths, and *rectangle 2* and *rectangle 3* will be found as the results.

The *insert* operation also starts from the root and tries to find a leaf node to contain the request rectangle. At each level, the insert algorithm selects the node whose MBR will have the minimum enlargement if it contains the request. If there is a tie, the algorithm will select the node having the minimum area. When finding a leaf node to insert, the algorithm will recursively update MBRs in the path from the leaf to the root. The insert operation may lead to the split of a node, which can be a leaf, an internal, or even the root. Different split algorithms will generate different structures of R-tree that contain the same set of rectangles, but vary the search performance. In this work, we use the mechanisms of R^*-tree [15] for the rectangle insertion and R-tree split. Because the search and insert can concurrently access an R-tree node, leading to the read-write and write-write conflicts, the lock mechanisms [16] are adopted for the concurrency control, which depends on the CPU to execute the low-level atomic instructions.

B. RDMA

RDMA is a network standard that provides high-bandwidth and low-latency by bypassing OS kernel processing and the remote CPU involvement in the communication. Figure 4 shows a comparison of using TCP/IP and RDMA.

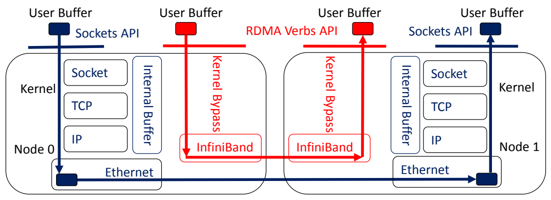


Fig. 4: TCP/IP *vs.* RDMA. RDMA can directly send data in a user buffer of the local node to a receive buffer at the remote node (or vice versa). The remote CPU and both side OS kernels are bypassed. While, TCP/IP needs OS kernels to process data packets, involving several internal memory copies. CPUs on both sides are also required.

Without involving the remote CPUs, an RDMA node can access data at another node via *RDMA Read* and *RDMA Write*. An RDMA node reads or writes a piece of remote memory according to a virtual address. Such virtual addresses are registered to network cards and are exchanged among nodes via TCP connections in advance. RDMA Read and RDMA Write are non-blocking, so the return of the operations does not guarantee the data written to remote memory or fetched back to the local memory. Polling is a commonly used technique for checking the completion of these operations. RDMA also supports the communication paradigm like the traditional TCP connection that both the sender and the receiver are aware of the data transmission, i.e., RDMA Send and Receive. But this method consumes more system resources and cannot achieve high performance as RDMA Read/Write [17]. In this paper, all communications are built upon RDMA Read and RDMA Write on the reliable connection (RC) of RDMA.

III. Basic R-Tree Designs over RDMA

In a TCP/IP-based client-server R-tree, a client first establishes a TCP connection with the server and then the client sends requests to the server via the TCP/IP connection. The R-tree server keeps listening on the established connection, accepts incoming requests, performs requested R-tree operations, and responds by returning the results to the client. Replacing TCP/IP with RDMA Read/Write, we can either follow the server-aware method by using RDMA Write, or switch to a server-unaware method for the R-tree search by exploiting RDMA Read. We name these two solutions as *fast messaging* and *RDMA offloading*.

A. Fast Messaging

In this design, the client and server pre-allocate their user-level buffers for containing request/response messages. In an R-tree access, the client directly writes the request to the server buffer via RDMA Write. The worker thread at server keeps polling the buffer to retrieve the request and then performs the corresponding operations, including search, insert, delete, and others. The results are directly written back to the client buffer also via RDMA Write. Figure 3(b) illustrates how a typical search works over a two level R-tree. In this example, the search results are retrieved after two RDMA writes and the R-tree traversal as the computation is carried out at server. With this method, RDMA is only used to accelerate data communication, and the R-tree operations are still handled by the server threads, similar to the TCP/IP-based solution.

The pre-allocated buffer at both sides are organized as a ring buffer for efficiency. Figure 5 shows the structure of the ring buffer, with several messages having variable lengths. A message in the ring buffer has a fixed format. The message size defines the whole size of the message. The receiver can check the message size, as shown in the figure "The other side is waiting here", to retrieve the size of current message. Following the size, the message type is designed for data segmentation, and the "request or response data" are for real payload. For variable sizes of response, the type field is flagged with "CONT" if more messages for the current response are arriving, and "END" ends the response.

There are two pointers for the ring buffer. The free pointer, also called tail, points to where the sender should write a new message. The processed pointer, also called head, points to where the receiver retrieve the earliest message. As shown in Figure 5, there are three messages, i.e., msg 0-2, having been sent from the sender to the receiver. The next message, e.g., msg 3, will arrive at the position pointed by the free pointer except that the message size exceeds the space between the tail and the head. The receiver is responsible for updating the processed pointer at the other side so that the sender can know if the previously-sent messages have been consumed. With the ring buffer mechanism, the client can send R-tree requests to the server, and the server can send R-tree responses to the client.

For the R-tree itself, we implement the R^*-tree [15] algorithm to split an R-tree node when it is full. To handle the concurrent accesses from multiple clients, we implement the concurrent lock [16] to avoid the read-write conflict and write-write conflict. Therefore, in the fast messaging design, R-tree itself is nearly the same to the original R-tree, while the communication between the server and the client is based on RDMA Write, instead of TCP/IP in existing client-server R-tree designs.

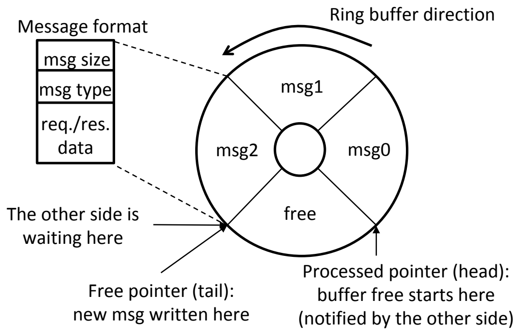

Message format
msg size
msg type
req./res. data
Ring buffer direction
msg1
msg2
msg0
free
The other side is waiting here
Free pointer (tail): new msg written here
Processed pointer (head): buffer free starts here (notified by the other side)

Fig. 5: Ring buffer design for RDMA Write

B. RDMA Offloading

In the second design, the client is able to directly read R-tree nodes from the server via RDMA Read, and traverses the tree at the client side. The client starts from fetching the root node of R-tree, and checking the root node to know which child nodes have the MBRs intersected with the request. The child nodes are retrieved via RDMA Read, and this process repeats until the client finds a set of (or none of) rectangles at leaf nodes intersected with the request. Figure 3(c) gives an example of R-tree traversal over RDMA Read. In this case, the client issues an RDMA read to fetch the root node, does the intersection check, and finds nodes 1 and 2 are intersected with the request. Then, the client issues two separate RDMA reads to fetch these nodes and gets rectangles 2 and 3 as the results. In practice, this method incurs multiple network RTTs, impairing the performance of R-tree accesses.

In RDMA offloading, we still use RDMA Write to send R-tree write requests, e.g., insert and delete, to the ring buffer of the server; and let the server threads handle the write requests. Note that such a design will not use RDMA Write to directly modify the R-tree. Therefore, the lock mechanism can prevent the write-write conflict as the write operations are enforced by the server threads.

In this design, we need a concurrency control mechanism at the client side for the R-tree read operations, because the retrieved R-tree node via RDMA Read is possibly being written by a server thread. Since server-side CPUs are totally bypassed in RDMA Read, we cannot depend on a conventional lock mechanism to prevent the read-write conflict. We apply a version number-based mechanism [6] to avoid the read-write conflict. This method inserts a version number to each cache line of an R-tree node when creating the node, and updates the version number in any write operation, e.g., insert or delete. When an RDMA read fetches a node back to the client, the client needs to check if all version numbers of this node are consistent. If not, there is a read-write conflict and the client has to retry this read. The concurrency control is ensured because both RDMA Read (from the client) and CPU Write (from the server) are cache-line atomic.

Besides the concurrency control, RDMA offloading needs an additional mechanism for the memory management. As the R-tree can be directly accessed by clients via RDMA Read in RDMA offloading, the memory address of R-tree needs to be registered at the server-side network card and propagated back to clients. In order to reduce the overhead of memory registration [18], [19], we allocate enough memory on server to hold the whole R-tree, and register the memory to the network card once. During the connection initialization, the registered memory address is returned to the client. We also divide this buffer into small chunks, each of which is for an R-tree node. Using the address of registered memory as the start point and the chunk ID as the offset, a client can use RDMA Read to fetch any R-tree node.

IV. Advanced R-Tree Designs over RDMA

There are drawbacks of only using a single solution at a time. Fast messaging achieves high performance because of its fast communication speed on RDMA Write and a single RTT required. However, as the server runs out its CPU cycles, the access latency gets much longer when there are increasingly more clients sending requests, even if the bandwidth are not used up. For RDMA offloading, it can reduce heavy loads in the server, however, the cost of multiple RTTs in a R-tree search must be a consideration for its usage. The larger the height of an R-tree, the higher its overhead is.

There are three sources of resources: server CPU cycles, client CPU cycles, and network bandwidth between the above two. Here are our observations on the dynamics of the three resources. When the server-side CPU is not a performance bottleneck, fast messaging can accelerate the communication for both request and response. Since RDMA offloading can complete an R-tree search without any server CPU resources, it is possible to improve the overall system throughput by using RDMA offloading as a complementary R-tree access method, in the case that the server CPUs are busy but the server bandwidth is abundant. However, it is challenging to design a comprehensive solution to best utilize all the resources, because clients are independent and may not choose the most proper action. An adaptive and automatic coordination mechanism is desirable to assist clients to determine which R-tree access method should be used.

A. Adaptive R-Tree Search

Our coordination mechanism is analogous to the binary exponential back-off (BEB) protocol in Ethernet [20] and Wireless LAN [21]. The mechanism is composed of two modules at the clients and the server, respectively. The server module is responsible for notifying the connected clients of the server CPU loads. The R-tree server periodically (10 ms in our setup) collects and embeds its CPU utilization statistics into a heartbeat that is sent to all active clients by the aforementioned ring buffer design. By analyzing the heartbeats, a client knows if there are available CPU cycles for future RDMA searches. Understanding the server status is not enough. If all clients choose RDMA offloading for their future requests when the server is busy, the server CPUs would be idle very soon and the system throughput would degrade. To avoid this side effect that all clients switch to RDMA offloading simultaneously, we design the client module to follow an adaptive rule: when a client finds the server is busy, it switches to RDMA offloading for the next n requests, where n is chosen randomly from $[0, N)$ and N is a predefined parameter. In this way, the clients will switch back to fast messaging at a different time, avoiding congestion again. The candidate interval will further back off to $[N, 2N)$, if the server CPUs are found still busy after two consecutive requests. The back-off continues without an upper bound. So in the extreme case, all R-tree searches of a client are completed with RDMA offloading. Note that, in our design, R-tree write requests are always sent to the server via the ring buffer and processed by the sever CPUs. This is the major reason why we allow all R-tree searches to be processed by RDMA offloading.

Algorithm 1 The adaptive solution with the back-off algorithm

```
1:  procedure RTREE-SEARCH-ADAPTIVE($req$)
2:      Input: $Inv$, $u_{serv}$, $N$, $T$, $r_{busy}$, $r_{off}$, getTimeOfDay($\cdot$)
3:       FastMessaging($\cdot$), RDMAOffloading($\cdot$), predUtil($\cdot$)
4:                          $\triangleright$ $N$, $T$ and predUtil($\cdot$) are configurable
5:      $U \leftarrow 0$
6:      $t \leftarrow$ getTimeOfDay()
7:      if $t - t_0 > Inv$ and $u_{serv} \neq 0$ then
8:          $U \leftarrow$predUtil($u_{serv}$)
9:          memset($u_{serv}, 0$)
10:         $t_0 \leftarrow$ getTimeOfDay()
11:     end if
12:     if $U > T$ and $r_{off} \leq r_{busy} \cdot N$ then
13:         $r_{busy} \leftarrow r_{busy} + 1$
14:         $r_{off} \leftarrow$ rand() $\% N + (r_{busy} - 1) \cdot N$
15:     else
16:         $r_{busy} \leftarrow 0$
17:     end if
18:     if $r_{off} > 0$ then
19:         $r_{off} \leftarrow r_{off} - 1$
20:         RDMAOffloading($req$)
21:     else
22:         FastMessaging($req$)
23:     end if
24: end procedure
```

Our adaptive search algorithm is shown in Alg. 1. Variable Inv stands for the heartbeat interval and u_{serv} is the memory region where the heartbeats are written. These two variables are agreed by the client and server when the RDMA connection is established, and allowed to be different in multiple client-server pairs. T is a predefined threshold for identifying

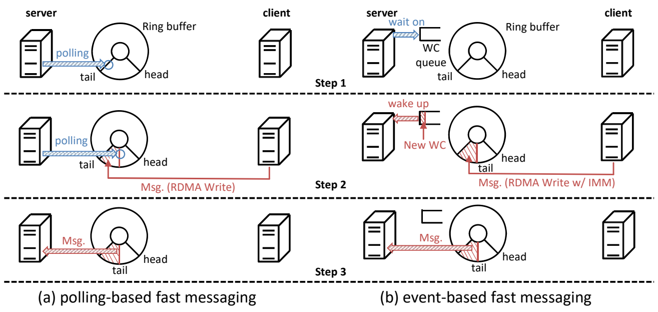


Fig. 6: Polling- *vs.* event-based fast messaging. (a) For polling-based fast messaging, the server keeps polling the tail of the ring buffer for detecting message arrival (**step 1**). As a message is starting to be delivered to the ring buffer via RDMA Write, the server changes the polling position to know when delivery is completed (**step 2**). Eventually, the server can retrieve the complete message from the ring buffer (**step 3**). (b) For event-based fast messaging, the server waits on a Work Completion (WC) queue and yields the CPU (**step 1**). If a message has arrived via RDMA Write with Immediate Data (RDMA Write w/ IMM), the RDMA NIC will also generate a notification in the WC queue and wake up the thread waiting on it (**step 2**), which then retrieves the message (**step 3**).

if the server CPU is busy. r_{busy} shows in how many continuous rounds the server is identified as busy and r_{off} means how many rounds the R-tree searches should be offloaded to the client, where a round means a complete RDMA search. predUtil$(\cdot)$ predicts the server CPU utilization based on the server CPU information sent to the client. Currently, we use the most recent CPU utilization as the predicting value. The algorithm first checks the server CPU information in u_{serv}. It ignores that no heartbeat has arrived, because the reason of the delay could be the saturated server bandwidth. In this case, switching to RDMA offloading will consume more bandwidth. Second, our algorithm determines if it switches to RDMA offloading and how many rounds this should last. If the current communication method is RDMA offloading but the server is still busy, the offloading rounds will be extended. Finally, the client processes the R-tree search request by using fast messaging or RDMA offloading according to the determination. Besides the coordination mechanism that adaptively switch the R-tree search methods, we also optimize the fast messaging and RDMA offloading for R-tree.

B. Event-based Fast Messaging

Conventionally, for handling RDMA Write, the server thread needs to poll a piece of memory region agreed by both sides to recognize the message arrival. Figure 6 (a) illustrates how the polling-based fast messaging works. However, the server-side CPU cycles on such busy polling may increase linearly with the number of active connections. Even worse is the case of CPU oversubscription, i.e., the number of connections is larger than the number of CPU cores. Because the OS kernel is responsible for thread scheduling, a thread that is not scheduled by OS is delayed in polling, even if its request has arrived; and other scheduled threads may be wasting the CPU cycles if there are no incoming requests in their connections. To investigate how severe this problem is,

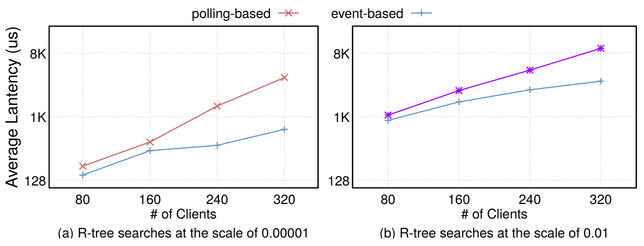


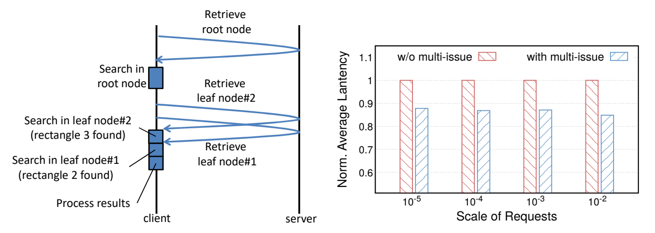
Fig. 7: Performance comparisons of polling- *vs.* event-based fast messaging on 100 Gbps InfiniBand.

we conduct an experiment in which the server-side CPUs are saturated by R-tree searches. The experimental setup is the same as that in Section I, except that the nodes are connected by InfiniBand for RDMA and the number of clients varies from 80 to 320.

As shown in Figure 7 (a), the polling-based fast messaging has the average search latency at 203.96 μs, when there are 80 clients sending the requests at the scale of 0.00001 (the scale of 0.00001 corresponds to the case of server CPU-bound). But the average latency quickly reaches as high as 3712.35 μs (18.2x worse), when there are 320 clients. Figure 7 (a) also shows that when the R-tree accesses are bound to server-side CPUs, the polling-based fast messaging makes the search latency increases quadratically. Since we consider the use case of data center as shown in Figure 1, where the number of active clients to a single R-tree server can easily exceed the number of CPU cores on a server, a new design must address this issue for RDMA Write-based fast messaging.

We change the notification mechanism of RDMA Write on server from being polling-based to being event-based. This event-based fast messaging is described in Figure 6 (b). Specifically, other than polling, a server thread blocks on its RDMA connection and yields the occupied CPU until the arrival of next message. We modify our ring buffer design on both the server and the client for an event-based connection as below. In the client, we change the regular `RDMA Write` to `RDMA Write with Immediate Data`. This method will generate a `Work Completion` carrying the immediate data in a completion queue at the server when the request message has been written to the ring buffer of the server. In the server, each thread creates an event channel and registers it over the completion queue. As a result, the server thread yields the CPU by waiting on the event channel and is awaken if a request arrives.

The effects of using the event-based mechanism are shown in Figure 7, in which 80 clients has the average search latency of 152.50 μs for search requests at the scale of 0.00001. By increasing the client number to 320, the average search latency linearly increases to 680.47 μs. The similar performance trend can be observed if the search requests are sets at the scale of 0.01. As a result, the event-based design can make the R-tree accesses more scalable on RDMA.



(a) RDMA offloading with multi-issue (b) Latency of RDMA offloading w/ and w/o multi-issue on 100 Gbps InfiniBand

Fig. 8: RDMA offloading with multi-issue and its performance

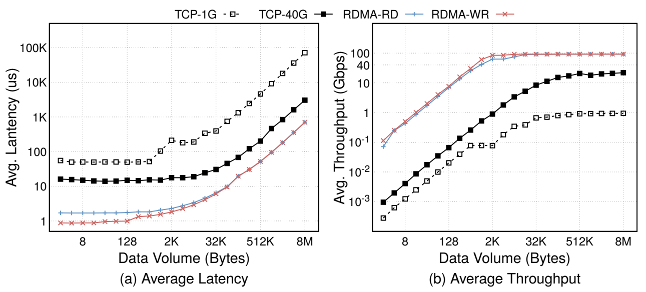


Fig. 9: Micro benchmark of different communication methods

C. *RDMA Offloading with Multi-issue*

A major concern of RDMA offloading in R-tree search is its multiple network RTTs, where each R-tree node access corresponds to an RDMA read. To address this issue, our basic idea is not to retrieve R-tree nodes one by one during the traversal; and instead, we simultaneously dispatch multiple RDMA reads to fetch as many as possible R-tree nodes. The network RTTs can be hidden in a pipeline. We name it as **multi-issue** that is quite suitable for the R-tree structure and traversal. First, there is no dependency between child nodes at the same R-tree level. Once we get a parent node, we can obtain multiple child node pointers. Second, different with the B-tree search, which goes along one path from the root to a leaf node, an R-tree search involves multiple paths since it needs to check if the request is intersected with all child nodes in the current MBR. As a result, after checking the intersection of the request and the current R-tree node, the multi-issue technique can issue multiple RDMA reads to fetch all intersected child nodes. Figure 8(a) shows two RDMA reads are issued to retrieve nodes 1 and 2 for the case in Figure 3(a). These two RDMA reads are pipelined on the client-side network card, the network connection, and the server-side network card. Moreover, once the RDMA read for retrieving leaf node 2 returns, the client can do the intersection check immediately, which further overlaps the following RDMA read retrieving leaf node 1.

We experimentally check the efficacy of multi-issue, as shown in Figure 8(b). The setups are similar as the experiment in Section IV-B, except that only one client is involved. We send R-tree search requests at four different scales from 0.00001 to 0.01. At all request scales, the multi-issue technique can effectively improve the R-tree search performance. Among all cases, the most search latency reduction of 15.13% appears at the scale of 0.01 that mimics the search on a large scope.

V. Experimental Results

We carry out our evaluations on a cluster including 9 compute nodes. Each node has a dual-socket Intel Xeon E5-2680 v4 (2×14-core Intel Broadwell, 2.40 GHz, 512 GB DDR4). There are three types of network cards installed on each node, including: (1) an Intel I350 1Gbps Ethernet controller, (2) a Mellanox MT27500 Family (ConnectX-3) adapter card, supporting 40Gbps Ethernet connectivity, and (3) a Mellanox MT27800 Family (ConnectX-5) adapter card, supporting EDR 100Gbps InfiniBand connectivity. We use one node as the server and remaining nodes for independent clients (up to 32 clients per node and 256 clients in total).

We label the socket-based R-tree solutions as "TCP/IP-1G" and "TCP/IP-40G" for 1Gbps Ethernet and 40Gbps Ethernet, respectively. We implement FaRM [6] for R-tree and label their methods as "Fast messaging" and "RDMA offloading" as the baselines of RDMA solutions. We implement our optimizations, including the event-driven fast messaging, multi-issue-based offloading, and our adaptive solution as "Catfish". All these methods are integrated with R^*-tree [15].

A. *Micro benchmark*

We implement a pair of TCP/IP server and client. The client keeps sending requests (1 Byte) to the server and the server responses the client with different sizes of data chunks. For InfiniBand, we use the *perftest* benchmark [22], in which data chunks are delivered over RDMA Read or RDMA Write. For both TCP/IP and RDMA communication, the size of data chunks ranges from 2 Bytes to 8M Bytes, and the transmission of a data chunk only begins if the previous one has finished. The results of transmission latency are shown in Figure 9(a). We can see that RDMA Write has the lowest average latency and TCP/IP over 1G Ethernet has the highest average latency. RDMA Read has higher transmission latency than RDMA Write, especially for the small data sizes. That is because RDMA Read needs a round trip of network communication, while RDMA Write is one direction. We also observe that the latency of all methods keeps nearly constant when sending small data ($<$ 2K), but when delivering medium and large data ($>$ 2K), the latency is determined by the bandwidth. We also measure the transmission throughput, and the results are shown in Figure 9(b). The TCP/IP over 1 Gbps Ethernet has the lowest throughput while the two RDMA connections have the highest throughput as expected. In all methods, the bandwidth can only be fully exploited when sending medium and large data ($>$ 2K).

B. *R-Tree Throughput and Latency*

To evaluate Catfish, we build an R-tree with 2 million rectangles, whose edges scale in the range of $(0, 0.0001]$ randomly. We put the R-tree in the main memory of the server.

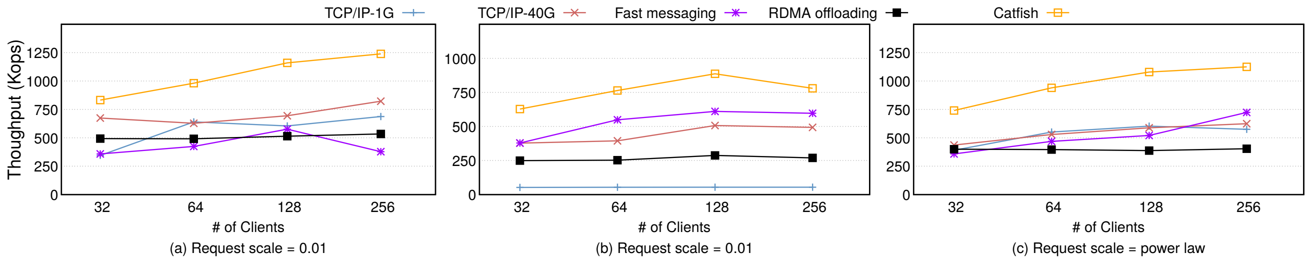


Fig. 10: The throughput of workloads composed of 100% search requests at various scales to an R-tree.

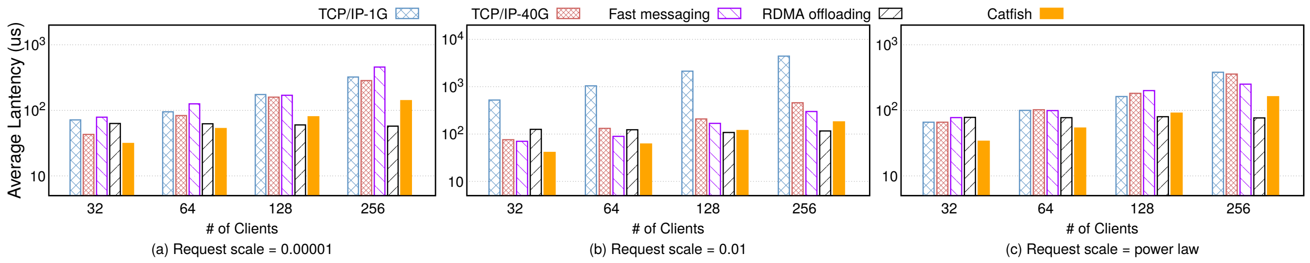


Fig. 11: The latency of workloads composed of 100% search requests at various scales to an R-tree.

The other 8 compute nodes host independent clients (from 4 to 32 per node) and each client issues 10000 search requests via different methods. The request scale is one of *0.00001*, *0.01*, or *power law*: The request scale of 0.00001 means the edges of a requested rectangle are randomly selected from $(0, 0.00001]$. These are CPU intensive workloads, representing a search in a geographically small scope. The scale of 0.01, on the other hand, is designed for bandwidth intensive cases, representing a search in a large scope. We also generate the requests according to a power law distribution, where the probability of request scales complies to $f(t) \propto t^{-0.99}$, and $t \in (0.00001, 0.01]$, resulting in much more requests to search in a small scope. This is for the skewed search cases that is general in the real world. For "Fast messaging" and Catfish, we allocate a ring buffer of 256 KB for each pair of connections. For Catfish, we set the parameter N to 8 and T to 95%, meaning that each client will use RDMA offloading for at most 8 consecutive requests when observing the CPU utilization on server is higher than 95%.

In the first group of evaluations, all requests are search operations (read only). The results are shown in Figure 10 and Figure 11. In Figure 10, the y-axis is the throughput in kilo-operations-per-second (Kops) and the x-axis is the number of clients. We can see that Catfish achieves the highest throughput in all cases. When there are 256 clients, Catfish handles the R-tree accesses at a rate of 1239.35 Kops (for request scale of 0.00001), 779.89 Kops (for request scale of 0.01), and 1124.84 Kops (for the power law distribution). In Figure 10 (a), we can see for the CPU-bound case, directly using RDMA, no matter RDMA-Write-based fast messaging or RDMA-Read-based offloading, cannot get good performance, even compared to the TCP/IP-based solution on 1Gbps Ethernet. Fast messaging has the worst throughput, because when the server-side CPUs become the bottleneck, the faster to send client requests to the server, the more overloaded the server becomes. As shown in the figure, RDMA offloading cannot get good performance either, because all search operations are executed at the clients with too many network RTTs. For the network-bound case in Figure 10 (b), RDMA offloading that trades the network bandwidth for the server-side CPU cycles cannot help. In this case, fast messaging is preferred. All these drawbacks of using RDMA are solved by Catfish, since the Catfish clients can adaptively switch to the enhanced fast messaging and RDMA offloading. In terms of throughput, Catfish outperforms fast-messaging and RDMA offloading by up to 3.28x and 3.09x, respectively. And the improvement of Catfish over the TCP/IP based schemes is up to 16.46x.

The results of latency are presented in Figure 11. Catfish has much lower request latency compared with fast messaging, because of the timely offloading when the server-side CPU is saturated and the event-based mechanism for detecting request arrivals. For the experiments with 256 clients, the average search latency of Catfish is 140.73 μs (0.00001), 180.66 μs (0.01), and 161.58 μs (power law), while fast messaging needs 299.10 μs, 321.52 μs, and 302.91 μs. RDMA offloading has constantly low search latency, and even better than Catfish in some cases, e.g., the latency of 256 clients at sale 0.00001 in Figure 11 (a). There are two major reasons for this case. First, each client of Catfish needs to run the back-off algorithm. The algorithm execution time is the overhead of Catfish. Second, our back-off algorithm is heuristic: only if a client switches back to fast messaging, it can find the CPU utilization on server is still high and needs to use RDMA offloading. When the server-side CPU are constantly overloaded, clients need

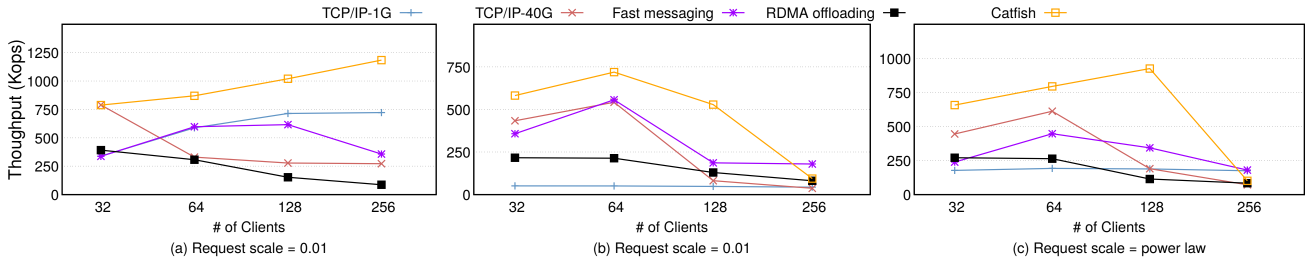


Fig. 12: The throughput of workloads composed of 90% search requests and 10% insert requests at various scales to an R-tree.

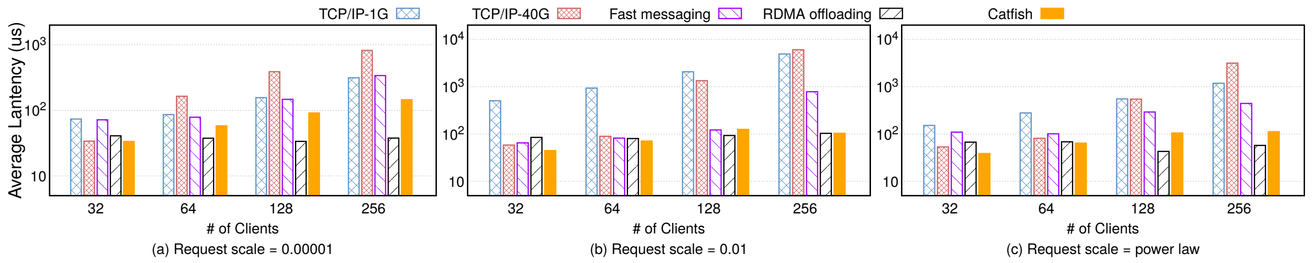


Fig. 13: The latency of workloads composed of 90% search and 10% insert requests at various scales to an R-tree.

to stay on offloading instead of switching back and force. A possible solution is that in a recent study [23], which uses machine learning methods to select the best configuration at the runtime. We leave this to our future work. As shown in the figure, both TCP/IP solutions have much higher average latency, since the messages have to go through all network stacks in OS kernel. Overall, Catfish can reduce the average latency by up to 3.25x (over fast messaging), 3.07x (over RDMA offloading), and 24.46x (over TCP/IP based).

We also evaluate Catfish with workloads that have both search and insert operations. The results are shown in Figure 12 and Figure 13. In the experiments, the clients independently generates 90% search requests and 10% insert requests. The rectangle scales of both search and insert requests are the same as those in the previous experiments. We select the locations for the insert rectangles as follows: 1) both the x and y coordinates are firstly selected following a power law distribution $f(t) \propto t^{-0.99}$, where $t \in (0.5, 1.0]$. 2) Then we randomly offset the insert position (x, y) to one of (x, y), $(1-x, y)$, $(x, 1-y)$ and $(1-x, 1-y)$. This represents the skewed insertion that mimics the geographical data updates more often happening in city areas. Catfish can still achieve the highest throughput, except 256 clients sending requests at the scales of 0.01 and the power law distribution. In these two cases, the insert operations have dominated the server-side CPUs and the adaptive algorithm can hardly help, because Catfish is for optimizing R-tree search operations. The performance of RDMA offloading slightly degrades when increasing the number of clients. This is because more inserts are done at the server, the higher probability the clients will find the read-write conflict in the offloading. For the query latency, we can observe the same trend as the search-only experiments. Overall, Catfish improves the throughput by up to 3.3x (over fast messaging), 13.67x (over RDMA offloading), and 14.22x (over TCP/IP based) and reduces the average latency by up to 7.55x (over fast messaging), 1.90x (over RDMA offloading), and 58.09x (over TCP/IP based).

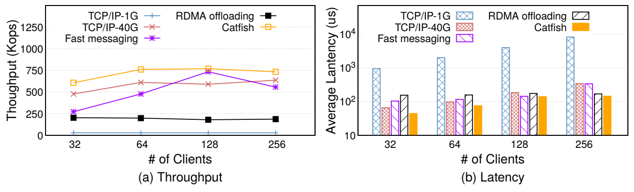


Fig. 14: The performance comparisons of R-tree search on the real dataset *rea02*.

C. Tests on a Real-world Dataset

We also evaluate Catfish by using a real world dataset *rea02* [24]. This dataset is composed of 1,888,012 rectangles representing street segments in California, US. The rectangles are grouped as sub-regions which have roughly 20,000 objects. These sub-regions are inserted in a random order while inside a sub-region, the rectangles are inserted in the row order, west to east. The rows are inserted from north to south. The dataset also provides search requests. The query file guarantees that on average 100 rectangles will be returned, and the actual number for a request randomly distributes from 50 to 150.

The experimental results are shown in Figures 14 (a) and (b). In the figures, we can observe the same performance trends as the search-only experiments, in which Catfish has achieved the highest throughput and the lowest search latency against the other schemes. Catfish improves the throughput by up to

2.23x (over fast messaging), 4.28x (over RDMA offloading), and 27.25x (over TCP/IP based). It reduces the average latency by up to 2.32x, 3.47x, and 56.09x.

VI. The Application Scope of Catfish

Although we focus on improving R-tree by Catfish in this paper, our RDMA-based system is designed for general-purpose, because Catfish consists of three major components that are applications independent, namely, fast messaging, offloading, and an adaptive mechanism between them. Catfish is a framework for accessing link-based data structures over RDMA, such as B+tree and Cuckoo hashing, and R-tree in data centers. In this paper, we make a strong case for using RDMA in this group of applications. While Catfish suggests an effective mechanism balancing computation and network resources, it is also possible to add more intricate functions in Catfish to maximize its efficiency. For example, the server can periodically predict the overloading period and the response latency for clients instead of CPU utilization in the current design. In this way, clients can make a more accurate decision to initiate offloading or not.

VII. Related Work

Besides scientific databases and spatial databases, R-tree is adopted by many big data systems deployed in data centers, including SpatialHadoop [25], Hadoop-GIS [26], Simba [27], LocationSpark [28], iSPEED [29], DITA [30], etc. All of these systems need to access R-tree or its variants, e.g., R^{*}-tree, R^{+}-tree, IR-tree, Hilbert R-tree, etc., with a client-server mode in the distributed environment. Catfish can be used to improve their performance by balancing server-side CPU resources and network bandwidth resources.

With the decreasing DRAM price, distributed in-memory systems [5]–[9], [12], [31], [32] are prevalent in data centers. Many researchers start to explore RDMA for reducing the communication overhead of these systems. Jose et al. [12] improve the *Put* performance of key-value store by using RDMA. The client sends the memory address of a key-value pair to the server, and the server uses an RDMA read to fetch the key-value at the client. Mitchell et al. [5] propose Pilaf, a key-value store on RDMA. The Pilaf client can directly read a key-value pair in the server via RDMA Read, and use RDMA Write to send write requests to the server. This is a hybrid solution using fast messaging and RDMA offloading, but without the adaptive design and optimizations proposed in this work. Kalia et al. [32] propose HERD, a hybrid key-value store, which uses RDMA Write to send both *Put* and *Get* requests to the server, but sends responses to the client via RDMA Message on UD (Unreliable Datagram). As UD is not a reliable connection, the key-value store system has to support the reliable transport, e.g., the packet retransmission, fragmentation, etc. Jose et al. [33] propose another hybrid solution of using RC and UD for Memcached, where RC is used for the high performance and UD is used for the large scale. Wang et al. [7] propose C-Hint, a client-side cache system on RDMA. C-Hint uses RDMA Write to report hot key-value pairs to the server, and uses a leasing-based mechanism to mange cached key-value pairs on the client. HydraDB [8] is a key-value store on RDMA, combining the cache system of C-Hint and the out-of-place update mechanism on the server. Dragojevic et al. [6] propose FaRM, a distributed memory system on RDMA. FaRM uses RDMA Read to read remote objects and uses RDMA Write to send write requests to sever for the object modification. The version number-based mechanism is proposed to detect the read-write conflict, making FaRM be a general solution of using RDMA for data-center applications.

Cell [10] is an RDMA-enabled B-tree, which balances the overhead between client and server by providing a client-side cache for top levels of B-tree. Hadoop RPC [14] uses RDMA to implement RPC in Hadoop, bypassing the JVM memory management for high performance. FaSST [34] is a fast and scalable RPC system that uses two-sided RDMA Message on the unreliable connected (UC) and unreliable datagram (UD) transports. This method requires the server to handle network communication. Su et al. [35] show the asymmetric in-bound and out-bound performance of RDMA network cards, and propose Remote Fetching Paradigm (RFP) to reduce the out-bound overhead on server for RDMA-enabled RPC. Liu et al. [36] use RDMA Message on UD to optimize the shuffle operators for database queries. Salama [11] design an over-lapping mechanism to overlap the RDMA read on network and local query processing. Several studies [37]–[40] use RDMA to speedup distributed transactions. These studies use RDMA Atomics to coordinate distributed transactions in the two-phase commit protocol. However, Kalia et al. [41] reveal the poor performance of RDMA Atomics. Dragojevic et al. [9] construct distributed transactions on the basis of FaRM [6] that uses RDMA Read and Write, and report better performance than RDMA Atomic-based solutions.

VIII. Conclusion

In this paper, we present Catfish, an RDMA-enabled R-tree to achieve low latency and high throughput. Besides enhancing the existing communication facilities of RDMA software, we have developed an adaptive scheme to effectively switch between fast messaging and RDMA offloading to balance the resource utilization for R-tree. Experiments show the effectiveness of Catfish, which is the only available RDMA-based R-tree (to our best knowledge). Our adaptive mechanism and load balancing method can also be applied in other RDMA-enabled applications to best utilize RDMA in a timely fashion.

IX. Acknowledgments

We would like to thank the anonymous reviewers for their encouragement, constructive comments and suggestions. This work has been partially supported by the National Science Foundation under grants CCF-1513944, CCF-1629403, and IIS-1718450.

References

[1] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84. New York, NY, USA: ACM, 1984, pp. 47–57.

[2] Google, "Google maps," 2005. [Online]. Available: https://maps.google.com

[3] Yelp, "Yelp inc." 2004. [Online]. Available: https://www.yelp.com

[4] A. P. Iyer and I. Stoica, "A scalable distributed spatial index for the internet-of-things," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 548–560.

[5] C. Mitchell, Y. Geng, and J. Li, "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*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 103–114.

[6] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 401–414.

[7] Y. Wang, X. Meng, L. Zhang, and J. Tan, "C-hint: An effective and reliable cache management for rdma-accelerated key-value stores," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 23:1–23:13.

[8] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng, "Hydradb: A resilient rdma-driven key-value middleware for in-memory cluster computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 22:1–22:11.

[9] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: Distributed transactions with consistency, availability, and performance," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 54–70.

[10] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li, "Balancing cpu and network in the cell distributed b-tree store," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. Berkeley, CA, USA: USENIX Association, 2016, pp. 451–464.

[11] A. Salama, C. Binnig, T. Kraska, A. Scherp, and T. Ziegler, "Rethinking distributed query execution on high-speed networks," *Data Engineering*, p. 27, 2017.

[12] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda, "Memcached design on high performance rdma capable interconnects," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 743–752.

[13] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance rdma-based design of hdfs over infiniband," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 35:1–35:35.

[14] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-performance design of hadoop rpc with rdma over infiniband," in *Proceedings of the 42nd International Conference on Parallel Processing*, ser. ICPP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 641–650.

[15] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90. New York, NY, USA: ACM, 1990, pp. 322–331.

[16] M. Kornacker and D. Banks, "High-concurrency locking in r-trees," in *Proceedings of the 21th International Conference on Very Large Data Bases*, ser. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 134–145.

[17] A. Dragojevic, D. Narayanan, and M. Castro, "Rdma reads: To use or not to use?" *IEEE Data Eng. Bull.*, vol. 40, no. 1, pp. 3–14, 2017.

[18] M. Li, K. Hamidouche, X. Lu, H. Subramoni, J. Zhang, and D. K. Panda, "Designing mpi library with on-demand paging (odp) of infiniband: challenges and benefits," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 37.

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

[20] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM*, vol. 19, no. 7, pp. 395–404, 1976.

[21] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification*, IEEE Std. 802.11, 1997.

[22] M. Technologies, "Performance tests (perftest) package for ofed," 2015. [Online]. Available: https://github.com/linux-rdma/perftest

[23] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 1009–1024.

[24] N. Beckmann and B. Seeger, "A benchmark for multidimensional index structures," 2008. [Online]. Available: http://www.mathematik.uni-marburg.de/~seeger/rrstar/index.html

[25] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 1352–1363.

[26] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: A high performance spatial data warehousing system over mapreduce," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1009–1020, Aug. 2013.

[27] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 1071–1085.

[28] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1565–1568, Sep. 2016.

[29] Y. Liang, H. Vo, J. Kong, and F. Wang, "ispeed: An efficient in-memory based spatial query system for large-scale 3d data with complex structures," in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. SIGSPATIAL'17. New York, NY, USA: ACM, 2017, pp. 17:1–17:10.

[30] Z. Shang, G. Li, and Z. Bao, "Dita: Distributed in-memory trajectory analytics," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 725–740.

[31] A. Qin, M. Xiao, J. Ma, D. Tan, R. Lee, and X. Zhang, "DirectLoad: A Fast Web-scale Index System across Large Regional Centers," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019.

[32] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 295–306.

[33] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda, "Scalable memcached design for infiniband clusters using hybrid transports," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 236–243.

[34] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 185–201.

[35] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu, "Rfp: When rpc is faster than server-bypass with rdma," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 1–15.

[36] F. Liu, L. Yin, and S. Blanas, "Design and evaluation of an rdma-aware data shuffling operator for parallel database systems," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 48–63.

[37] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using rdma and htm," in *Proceedings of the 25th*

Symposium on Operating Systems Principles, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 87–104.

[38] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using rdma and htm," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 26:1–26:17.

[39] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 528–539, Mar. 2016.

[40] E. Zamanian, C. Binnig, T. Harris, and T. Kraska, "The end of a myth: Distributed transactions can scale," *Proc. VLDB Endow.*, vol. 10, no. 6, pp. 685–696, Feb. 2017.

[41] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance rdma systems," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. Berkeley, CA, USA: USENIX Association, 2016, pp. 437–450.