An RDMA-enabled In-memory Computing Platform for R-tree on Clusters

MENGBAI XIAO, School of Computer Science and Technology, Shandong University, China
HAO WANG and LIANG GENG, Dept. of Computer Science and Engineering, The Ohio State University, USA
RUBAO LEE, United Parallel Computing Corporation, USA
XIAODONG ZHANG, Dept. of Computer Science and Engineering, The Ohio State University, USA

R-tree is a foundational data structure used in spatial databases and scientific databases. With the advancement of networks 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 high. Specifically, an R-tree server can be heavily overloaded while the network and client CPU are lightly loaded, and vice versa.

In this article, 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 a client-server R-tree data processing system. 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 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.

CCS Concepts: • **Information systems** → **Multidimensional range search**; *Geographic information systems*; • **Networks** → *Cloud computing*

Additional Key Words and Phrases: RDMA, R-tree

This work has been partially supported by the National Science Foundation under Grants No. CCF-1513944, No. CCF-1629403, No. IIS-1718450, and No. CCF-2005884.

Authors' addresses: M. Xiao, School of Computer Science and Technology, Shandong University, Qingdao, Shandong, China, 266237; email: xiaomb@sdu.edu.cn; H. Wang, L. Geng, and X. Zhang, Dept. of Computer Science and Engineering, The Ohio State University, 2015 Neil Avenue, Columbus, OH, USA, 43210; emails: {wang.2721, geng.161}@osu.edu, zhang@cse.ohiostate.edu; R. Lee, United Parallel Computing Corporation, 6363 S Old State Rd, Lewis Center, OH 43035; email: lirb@unipacc.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
2374-0353/2022/02-ART15 $15.00
https://doi.org/10.1145/3503513

ACM Reference format:
Mengbai Xiao, Hao Wang, Liang Geng, Rubao Lee, and Xiaodong Zhang. 2022. An RDMA-enabled In-memory Computing Platform for R-tree on Clusters. *ACM Trans. Spatial Algorithms Syst.* 8, 2, Article 15 (February 2022), 26 pages.
https://doi.org/10.1145/3503513

1 INTRODUCTION

R-tree [16] 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 [15], Yelp [57], and others, front-end web servers accept user requests such as "Search this area" and "find restaurants near me" from the 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.

We carry out experiments on a small cluster to identify the locations of bottlenecks in the scenario shown in Figure 1, where the 1 Gbps Ethernet is used to connect compute nodes (detailed setups are introduced in Section 5). In the experiments, the 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 each sends 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 workloads. 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 area, like how many properties are about to 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 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 [19]. 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.

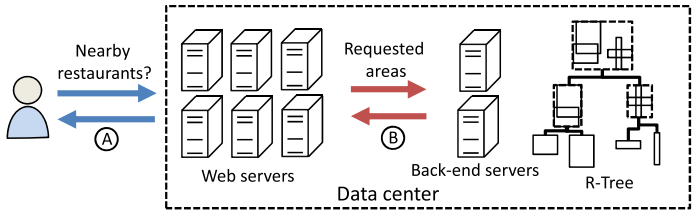
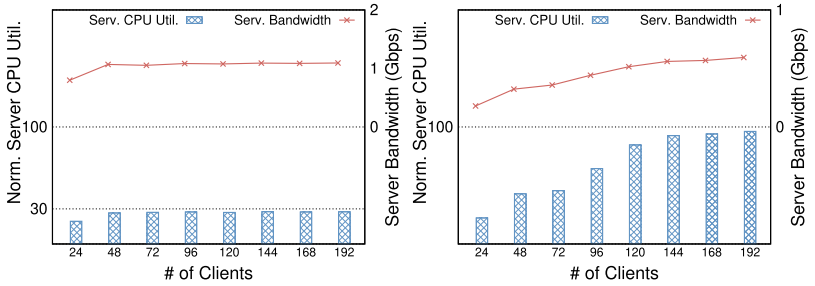


Fig. 1. An example of accessing spatial data in an R-tree, where **A** represents an Internet connection and **B** represents the intra-datacenter connection.



(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.

(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 [12, 13, 38, 39, 44, 52, 53]) and *write* (RDMA Write [18, 21, 35]) 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 1 Gbps Ethernet, 40 Gbps Ethernet, and 100 Gbps InfiniBand connections. For the workloads composed of 100% search requests, Catfish delivers up to 2.32×, 3.09×, and 16.46× speedups of throughput and 3.25×, 3.07×, and 24.46× 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 article makes the following contributions.

(1) To the best of our knowledge, 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.

Our previous research [55] discusses how the conventional TCP/IP network architecture could throttle the R-Tree performance, and proposes to use RDMA instead of Ethernet, along with mechanisms and algorithms developed for this emerging hardware. This work further reveals the poor scalability on the client node when accessing R-Tree with RDMA reads and presents a solution to this issue, followed by new experiments that prove its effectiveness. We also extends our evaluations to the basic R-Tree and the Hilbert R-tree [25], discovering that the optimization space of using RDMA varies with different R-Tree types.

2 BACKGROUND AND MOTIVATION

2.1 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 two-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.

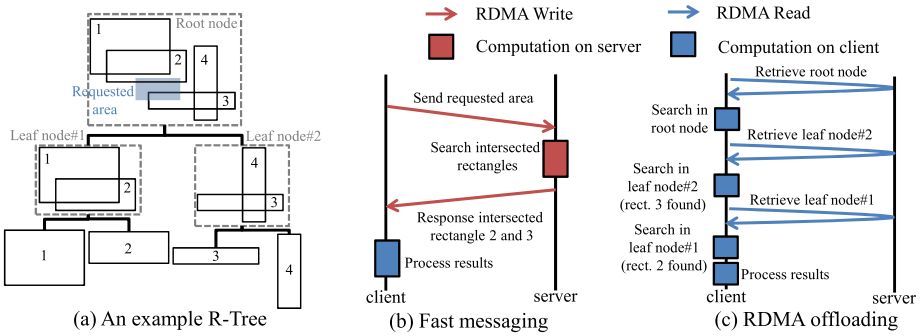


Fig. 3. The illustration of an R-tree organization and how fast messaging and RDMA offloading complete a spatial query on the R-tree in the client-server mode.

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, then 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, our RDMA-enabled design is evaluated on basic R-tree, R*-tree [4], and the Hilbert R-tree [25]. Because the search and insert can concurrently access an R-tree node, leading to the read-write and write-write conflicts, the lock mechanisms [28] are adopted for the concurrency control, which depends on the CPU to execute the low-level atomic instructions.

2.2 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.

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 [11]. In this article, all communications are built upon RDMA Read and RDMA Write on the **reliable connection (RC)** of RDMA.

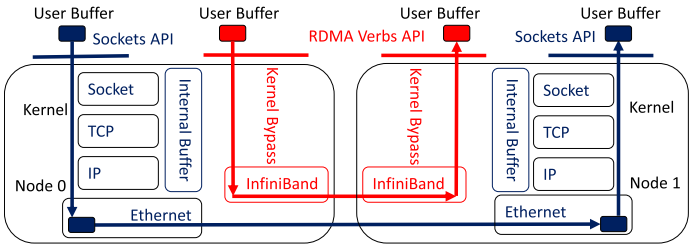


Fig. 4. TCP/IP vs. RDMA. RDMA can directly send data in a user buffer of the local node to a receive buffer at the remote node (or vice versa). The remote CPU and both side OS kernels are bypassed. While, TCP/IP needs OS kernels to process data packets, involving several internal memory copies. CPUs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

3 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*.

3.1 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. The detailed design and implementation of the ring buffer can be found in Reference [55].

For the R-tree itself, we implement the R^*-tree [4] algorithm to split an R-tree node when it is full. To handle the concurrent accesses from multiple clients, we implement the concurrent lock [28] 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.

3.2 RDMA Offloading

In the second design, the client directly reads 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 also 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, from the client to 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. However, we still 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 [12] to avoid the read-write conflict. Additionally, to reduce the overhead of memory registration [30, 50], we allocate enough memory on server to hold the whole R-tree, and register the memory to the network card once.

4 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 accelerates the communication for both request and response. Since RDMA offloading completes 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 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.

4.1 Adaptive R-Tree Search

Our coordination mechanism is analogous to the **binary exponential back-off (BEB)** protocol in Ethernet [37] and Wireless LAN [1]. 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 fast messaging 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, then 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

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(·), FastMessaging(·), RDMAOffloading(·),
    predUtil(·)   // N, T and predUtil(·) are configurable
3:
4:  U ← 0
5:  t ← getTimeOfDay()
6:  if t − t_0 > Inv and u_serv ≠ 0 then
7:     U ←predUtil(u_serv)
8:     memset(u_serv, 0)
9:     t_0 ← getTimeOfDay()
10: end if
11: if U > T and r_off ≤ r_busy · N then
12:    r_busy ← r_busy + 1
13:    r_off ← rand() % N + (r_busy − 1) · N
14: else
15:    r_busy ← 0
16: end if
17: if r_off > 0 then
18:    r_off ← r_off − 1
19:    RDMAOffloading(req)
20: else
21:    FastMessaging(req)
22: end if
```

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 fast messaging method and processed by the sever CPUs. This is the major reason why we allow all R-tree searches to be processed by RDMA offloading.

Our adaptive search algorithm is shown in Algorithm 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 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(·) 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 utilization 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, then the offloading rounds will extend. Finally, the client processes the R-tree search by using fast messaging or RDMA offloading according to the determination. Besides the coordination mechanism that adaptively switch the R-tree search, we also optimize fast messaging and RDMA offloading for R-tree.

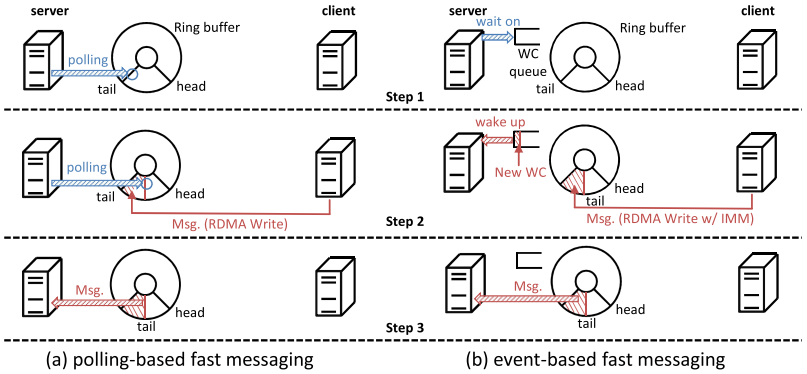


Fig. 5. Polling- vs. event-based fast messaging. (a) For polling-based fast messaging, the server keeps polling the tail of the ring buffer for detecting message arrival (**step 1**). As a message is starting to be delivered to the ring buffer via RDMA Write, the server changes the polling position to 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), then 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**).

4.2 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 5(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, 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 1, except that the nodes are connected by InfiniBand for RDMA and the number of clients varies from 80 to 320.

As shown in Figure 6(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 (18.2× worse), when there are 320 clients. Figure 6(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 5(b). The effects of using the event-based mechanism are shown in Figure 6, 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.

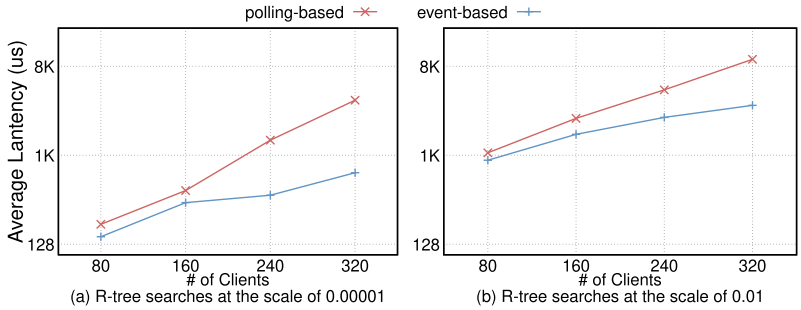


Fig. 6. Performance comparisons of polling- vs. event-based fast messaging on 100 Gbps InfiniBand.

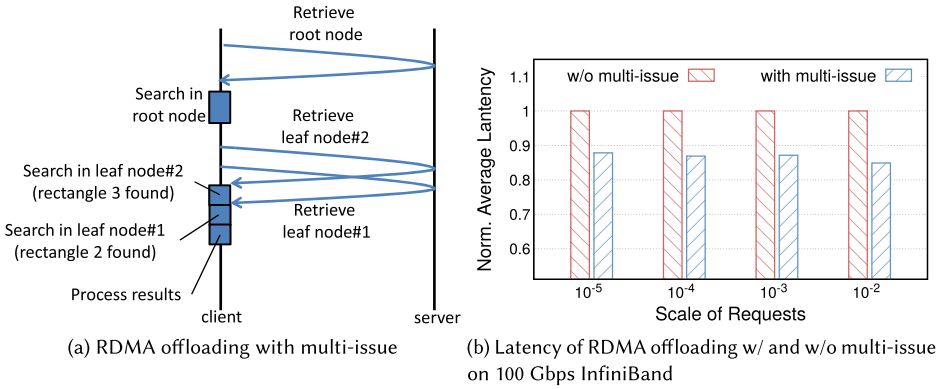


Fig. 7. RDMA offloading with multi-issue and its performance.

4.3 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 7(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 check the efficacy of multi-issue in Figure 7(b). The setups are similar as the experiment in Section 4.2, 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

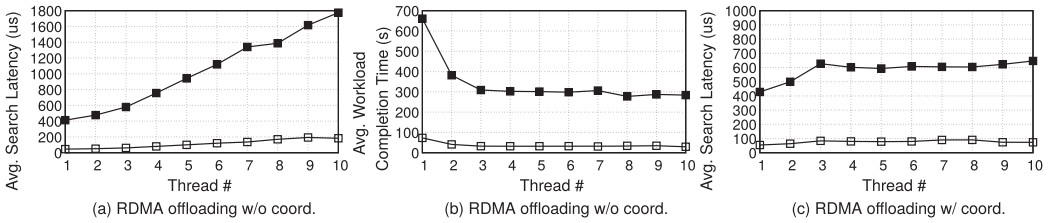


Fig. 8. Panel (a) is the average search latency of 12.8 million requests on 8 client nodes (1.6 million on each) to an R-Tree using RDMA offloading without thread coordination, and panel (b) is the average workload completion time among 8 client nodes. Panel (c) is the average search latency when enabling thread coordination, where conditional variables are used so that at most three threads can access the RDMA NIC on a client node when traversing the tree.

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.

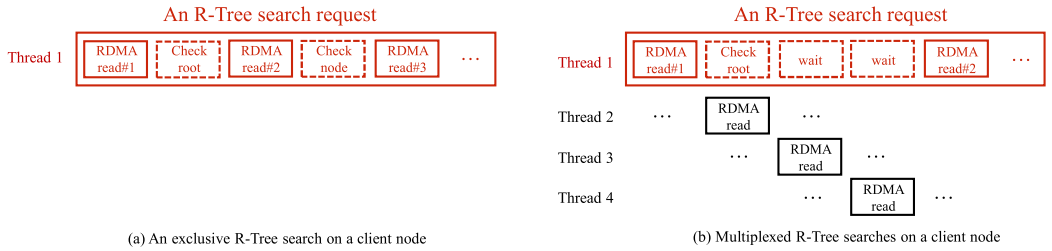
4.4 R-Tree Client Scalability

We also realize that simultaneously accessing an R-Tree with RDMA offloading from multiple clients locating at a same node could significantly degrade the performance. As shown in Figure 8, we build an R-Tree composed of 50 million rectangles with random sizes, and deploy 8 client nodes, on each 1.6 million search requests are averaged to 1–10 threads whose range randomly falls in $(0, 0.00001]$ or $(0, 0.01]$. On these client nodes, local barriers and MPI barriers are used to guarantee all threads start simultaneously.

In Figure 8(a), we measure the search latency as the time used from launching a request to collecting all qualified rectangles and observe how it varies along increasing thread number. The experimental results show that more clients are launched at a node, longer average access latency we will get, no matter which request scale is used. Despite the scalability issue of RDMA has been recognized earlier, which could come from the resource contention in server-side RDMA NIC [8] or the overwhelmed network [36], the degraded performance in our experiments should not be attributed to these factors, because the search latency keeps increasing even when the client number is small. We further investigate this issue by measuring the workload completion time, which is the time used to complete 1.6 million requests on a client node. From Figure 8(b), we find that when three or more threads are launched on a client node, the average workload completion time among all eight client nodes stays stable, indicating the RDMA connection has been saturated. This reveals that there is no resource contention happening in both client- and server-side RDMA NICs.

By looking into the details of RDMA offloading, we discover that the high latency of traversing the R-Tree comes from the design of how to use RDMA Read in our RDMA offloading mechanism. During an R-Tree search, multiple RDMA reads are issued by each thread to get the tree nodes from the root to the desired leafs, as well as the nodes on the search paths. Figure 9(a) shows an example of how a single thread complete an R-Tree traversal using RDMA offloading. If there are multiple threads on a client node, then the submitted RDMA reads would interleave with each other. An example with four threads launched at a client node is shown in Figure 9(b), where the execution time of an R-Tree search is prolonged, because more RDMA reads from other threads are inserted between the ones of this request.

To alleviate the deteriorated performance, we introduce a simple but effective concurrency control mechanism to coordinate concurrent clients that use RDMA offloading. A conditional variable



(a) An exclusive R-Tree search on a client node

(b) Multiplexed R-Tree searches on a client node

Fig. 9. One or four thread(s) launched on a client node to do R-Tree search with the RDMA offloading mechanism.

is used to control how many threads are allowed to concurrently access the RDMA NIC. Figure 8(c) shows the average search latency if at most three threads are allowed to do the R-Tree search with RDMA offloading on a client node. With the increasing thread number, the stable latency indicates that with the coordination mechanism RDMA offloading can scale. The number of threads that could simultaneously send R-Tree requests based on RDMA offloading can be empirically determined.

5 EXPERIMENTAL RESULTS

We carry out our evaluations on a cluster including nine 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 1 Gbps Ethernet controller, (2) a Mellanox MT27500 Family (ConnectX-3) adapter card, supporting 40 Gbps Ethernet connectivity, and (3) a Mellanox MT27800 Family (ConnectX-5) adapter card, supporting EDR 100 Gbps 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 1 Gbps Ethernet and 40 Gbps Ethernet, respectively. We implement FaRM [12] 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."

5.1 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 [49], 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 10(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 10(b). The TCP/IP over 1 Gbps Ethernet has the lowest throughput while the two RDMA

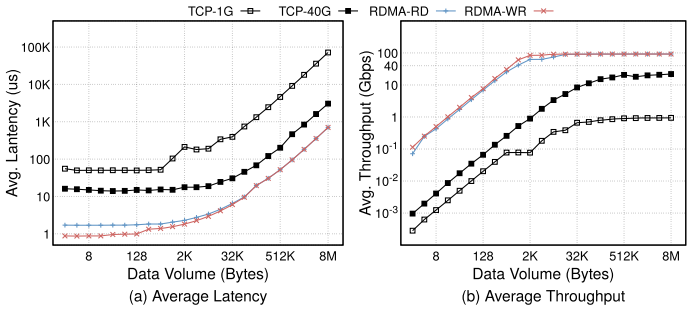


Fig. 10. Micro benchmark of different communication methods.

connections have the highest throughput as expected. In all methods, the bandwidth can only be fully exploited when sending medium and large data (>2K).

5.2 R-Tree Throughput and Latency

We evaluate Catfish and other peer solutions on three types of R-tree, the basic R-tree, the R*-tree [4], and the Hilbert R-tree [25]. Differing from the basic R-tree, the R*-tree is designed to minimize the overlapped areas of bounding boxes, thus the traversal paths of a search request is expected to be greatly reduced. However, additional overhead is introduced when inserting a new rectangle to the tree.

For every experiment, the tree is built with 2 million rectangles, whose edges scale in the range of $(0, 0.0001]$ randomly. We put the tree in the main memory of the server. The other eight compute nodes host independent clients (from 4 to 32 per node) and each client issues 10,000 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, however, 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 eight consecutive requests when observing the CPU utilization on server is higher than 95%. We will first present the experimental results of the R*-tree, and then the basic R-tree.

5.2.1 R-tree.* In the first group of evaluations, all requests are search operations (read only). The results are shown in Figures 11 and 12. In Figure 11, 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 1,239.35 Kops (for request scale of 0.00001), 779.89 Kops (for request scale of 0.01), and 1,124.84 Kops (for the power-law distribution). In Figure 11(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, especially compared to the TCP/IP-based solution on 40 Gbps Ethernet. Fast messaging has a lower throughput, because a large portion of time is used to poll if new messages arrive and the server-side CPUs become the bottleneck. As shown in the figure, RDMA offloading cannot get good performance either, because all search

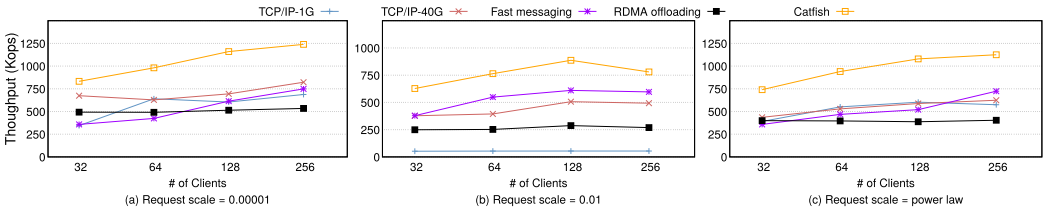


Fig. 11. The throughput of workloads composed of 100% search requests at various scales to an R*-tree.

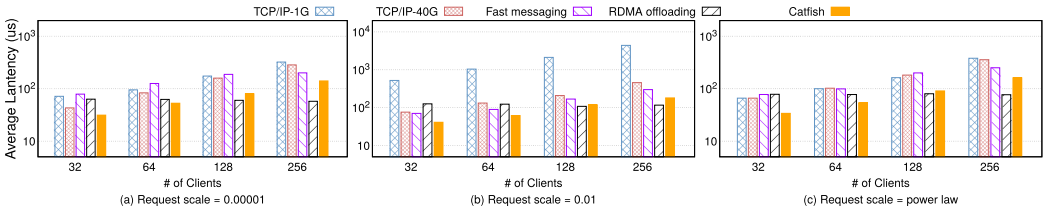


Fig. 12. The latency of workloads composed of 100% search requests at various scales to an R*-tree.

operations are executed at the clients with too many network RTTs. For the network-bound case in Figure 11(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. In terms of throughput, Catfish outperforms fast-messaging and RDMA offloading by up to 2.32× and 3.09×, respectively. And the improvement of Catfish over the TCP/IP-based schemes is up to 16.46×.

The results of latency are presented in Figure 12. 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 12(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 to stay on offloading instead of switching back and force. A possible solution is that in a recent study [51], 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 3.25× (over fast messaging), 3.07× (over RDMA offloading), and 24.46× (over TCP/IP-based).

We also evaluate Catfish with workloads that have both search and insert operations. The results are shown in Figures 13 and 14. 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 first 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

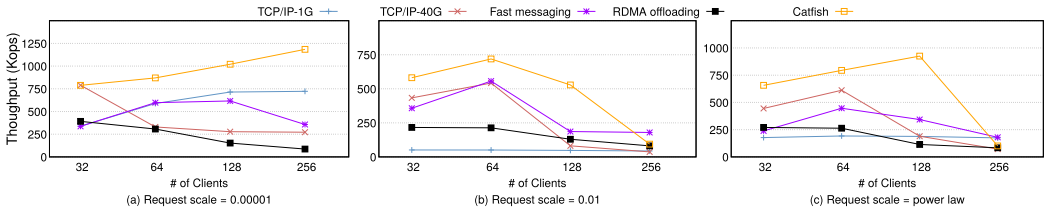


Fig. 13. The throughput of workloads composed of 90% search requests and 10% insert requests at various scales to an R*-tree.

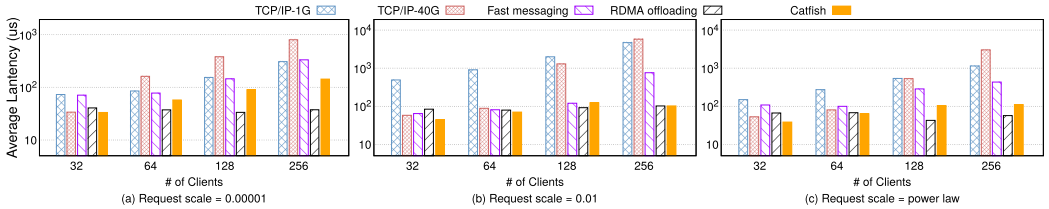


Fig. 14. The latency of workloads composed of 90% search requests and 10% insert requests at various scales to an R*-tree.

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 3.3× (over fast messaging), 13.67× (over RDMA offloading), and 14.22× (over TCP/IP-based) and reduces the average latency by up to 7.55×, 1.90×, and 58.09×, respectively.

5.2.2 Basic R-tree. We also evaluate Catfish and the other solutions on a basic R-tree with the same experimental configurations. The results of latency and throughput on the read-only workload is reported in Figures 15 and 16. Because a basic R-tree is expected to have much larger overlapped areas of bounding boxes than those in an R*-tree at the same scale, a search request has to traverse more nodes to find all qualified rectangles. This characteristic affects RDMA offloading most, since more nodes need to be fetched from the server to the client, leading to more RTTs and more network traffic in a request. When there are 256 clients at a node, RDMA offloading achieves the throughput of 193.75 Kops (for the request scale of 0.00001), 107.48 Kops (for the request scale of 0.01), and 147.58 Kops (for the power-law distribution), which is 2.75×, 2.50×, and 2.74× lower than the performance numbers on R*-tree, respectively. However, switching to the basic R-tree does not affect the performance of Fast messaging too much as the overhead comes from more memory accesses at server, which is much less than the network RTT overhead. The peak performance of using Fast messaging is 638.91 Kops (for the request scale of 0.00001), 553.67 Kops (for the request scale of 0.01), and 764.03 Kops (for the power-law distribution). They are similar to those measured on R*-tree, which are 575.53 Kops, 610.37 Kops, and 723.32 Kops, respectively. Catfish adopts both RDMA Read and RDMA Write for communication, so its access throughput is also impaired like what happens in RDMA offloading on the basic R-tree. In Figure 15, we observe that

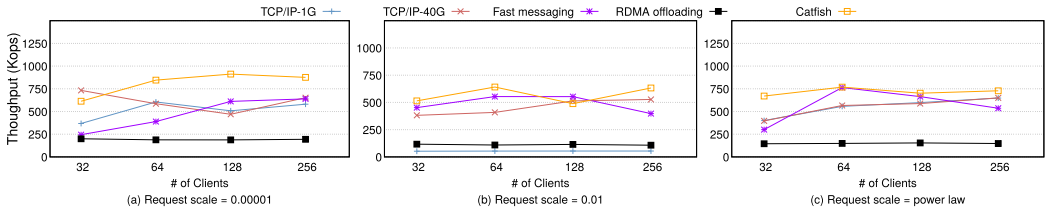


Fig. 15. The throughput of workloads composed of 100% search requests at various scales to a basic R-tree.

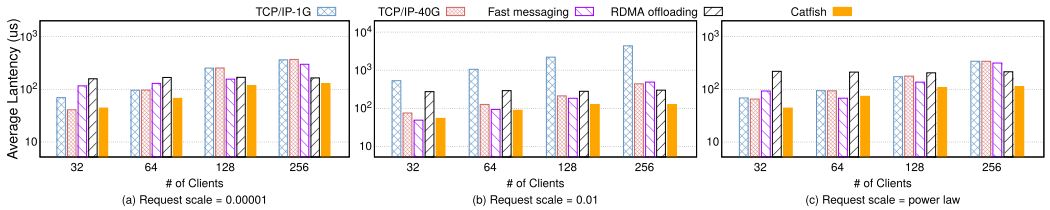


Fig. 16. The latency of workloads composed of 100% search requests at various scales to a basic R-tree.

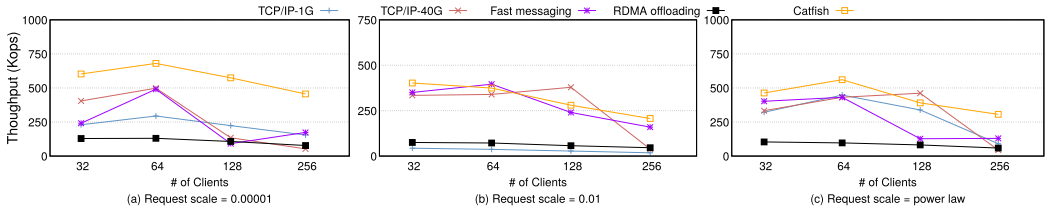


Fig. 17. The throughput of workloads composed of 90% search requests and 10% insert requests at various scales to a basic R-tree.

on the basic R-tree, Catfish can not get the similar performance benefit as on the R*-tree. In terms of access latency, Catfish has the lowest latency by taking advantage of the applied optimizations and the back-off algorithm. Overall, Catfish improves the throughput by up to 2.50× (over fast messaging), 5.88× (over RDMA offloading), and 11.92× (over TCP/IP-based). The average latency is reduced by up to 2.65×, 5.09×, and 35.87×, respectively.

The experimental results on the workloads composed of 90% search requests and 10% insert requests are shown in Figures 17 and 18. In terms of throughput, we observe that the same performance trend that Catfish can outperform the other schemes in the CPU-intensive cases (for the request scale of 0.00001), while if the bandwidth becomes the bottleneck (for the request scale of 0.01 and 256 clients) Catfish has the similar throughput as Fast messaging. For the average latency, Catfish always maintains it at a low level by switching a portion of RDMA Write-based requests to RDMA Read-based ones. Overall, the experiments on the mixed workloads and the basic R-tree show that Catfish achieves up to 6.28×, 5.28×, and 11.38× higher throughput than Fast messaging, RDMA offloading, and TCP/IP-based schemes, respectively. In addition, the average access latency of Catfish is up to 5.91×, 4.67×, and 41.67× lower than these compared schemes.

5.2.3 Hilbert R-tree. Another R-tree variant we evaluate the solutions on is the Hilbert R-tree [25]. As opposed to the R*-tree, the Hilbert R-tree inserts rectangles according to their Hilbert values, which are the one-dimensional coordinates on the space filling curve. The insertion operations in a Hilbert R-tree thus perform like the ones in the B-tree while the keys are the

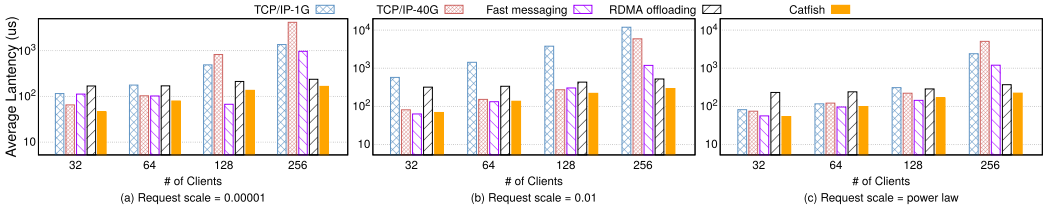


Fig. 18. The latency of workloads composed of 90% search requests and 10% insert requests at various scales to a basic R-tree.

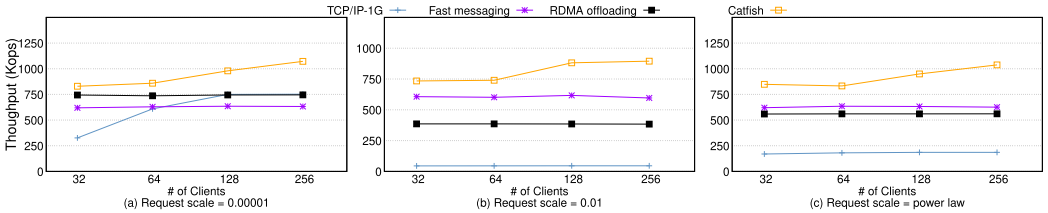


Fig. 19. The throughput of workloads composed of 100% search requests at various scales to a Hilbert R-tree.

Hilbert values. In our implementation, we introduce both the **largest Hilbert value (LHV)** and the **logical sequential number (LSN)** to a tree node. The LHV field is used for locating the insertion position while the LSN is added for the concurrency control [28]. We adopt the 2-to-3 split policy, since it is recognized a good compromise between the insertion cost and the search performance [25]. In the experiments, four schemes are evaluated, i.e., TCP/IP-1G, fast messaging, RDMA offloading, and Catfish.

Figure 19 shows the search throughput of the read-only workload. In the experiments, Catfish still has the best performance over the other schemes, because it exploits fast messaging and RDMA Offloading simultaneously. The peak performance of Catfish for requests at the scale of 0.00001, 0.01, and the power-law distribution are 1,072.06 Kops, 894.63 Kops, and 1,037.55 Kops. For the fast messaging scheme, it clearly outperforms RDMA Offloading in the bandwidth intensive cases, since it does not have to cost the RDMA bandwidth to deliver internal tree nodes. When the requests are at the scale of 0.01, the peak performance of fast messaging is 616.95 Kops, 1.60× higher than the peak performance achieved by RDMA offloading. We also notice that RDMA offloading approaches or even slightly wins fast messaging when more CPU intensive workloads are added. Furthermore, the overall search throughput from 256 clients via RDMA offloading reaches 745.67 Kops (for the request scale of 0.00001), 383.74 Kops (for the request scale of 0.01), and 560.94 Kops (for the power-law distribution). These are 1.40×, 1.43×, and 1.39× higher than the performance numbers on the R*-tree, respectively. By looking into the details of the query processing, we find that the superior performance of RDMA offloading in the Hilbert R-tree results from that less nodes are retrieved. Specifically, 12.15% less nodes are touched when we traverse the Hilbert R-tree, showing the excellent performance of the Hilbert R-tree composed of small rectangles. Figure 20 presents the search latency of the different schemes. The experimental results have a similar trend as the ones discovered when evaluating the R*-tree: (1) Catfish has a low search latency just like fast messaging when the workload is light; and (2) the access latency increases much slower than Fast messaging when the workload is heavy, since Catfish timely offloads the search operations to the clients. Overall, when searching on a Hilbert R-tree, the throughput is improved by up to 1.69× (over fast messaging), 2.33× (over RDMA offloading), and 19.71× (over TCP/IP-based) and the latency is reduced by 2.01×, 2.09×, and 24.20×, respectively.

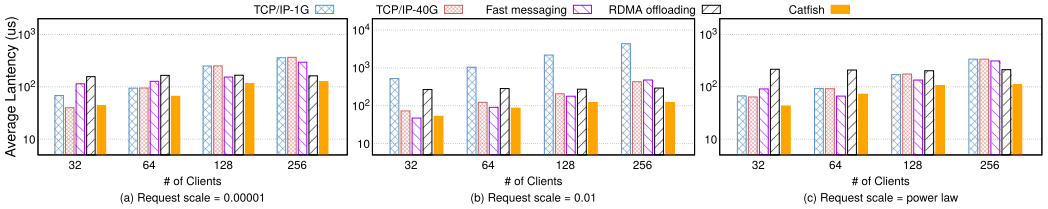


Fig. 20. The latency of workloads composed of 100% search requests at various scales to a Hilbert R-tree.

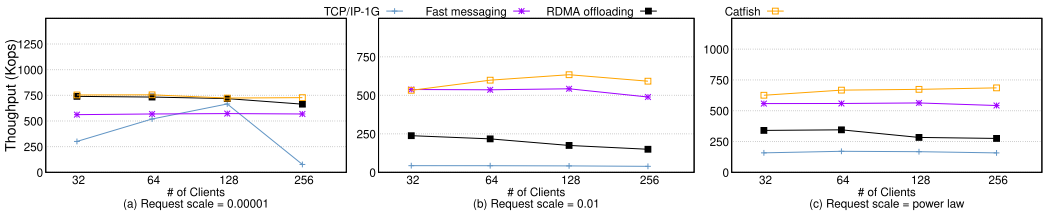


Fig. 21. The throughput of workloads composed of 90% search requests and 10% insert requests at various scales to a Hilbert R-tree.

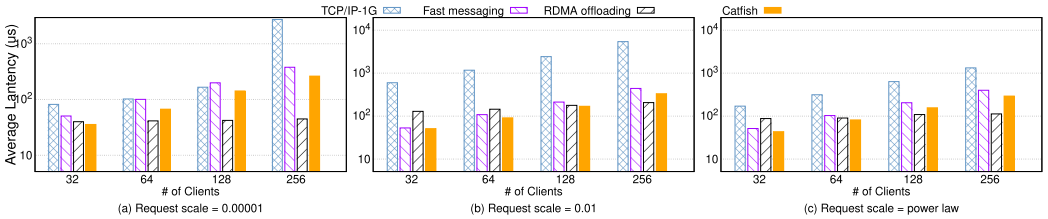


Fig. 22. The latency of workloads composed of 90% search requests and 10% insert requests at various scales to a Hilbert R-tree.

Catfish and the other schemes are also tested on the Hilbert R-tree with the workload of 90% search requests and 10% insert requests. The experimental results are shown in Figures 21 and 22. For the throughput, Catfish reaches the peak performance of 755.10 Kops, 633.84 Kops, and 685.80 Kops when the request scale is 0.00001, 0.01, and power-law distribution, respectively. Unlike the experiments conducted on the R*-tree, the throughput does not decrease when the number of clients increase. The reason is that the Hilbert R-tree inserts a rectangle only depending on its Hilbert value, incurring much less overhead than the R*-tree, which tries to minimize the relevant MBRs in the insert. Thus, even we have 256 clients connected to server and sending the requests, the server CPUs are still not saturated by the read/write contention. We can also observe the high performance of RDMA offloading in terms of throughput and latency, which benefits from the Hilbert R-tree design that is friendly toward small rectangles. In the experiments, Catfish increases the throughput by up to 1.34× (over fast messaging), 3.95× (over RDMA offloading), and 15.26× (over TCP/IP-based) and decreases the latency by up to 1.50×, 2.53×, and 16.51×, respectively.

5.3 Experiments with Varying Parameters

5.3.1 Varying Tree Sizes. To further investigate the performance of our RDMA-enabled R-tree, we measure the throughput and the access latency of requests to R-trees of different sizes. In the experiments, we build R*-trees with 10K, 100K, 1M, and 10M rectangles randomly generated in the

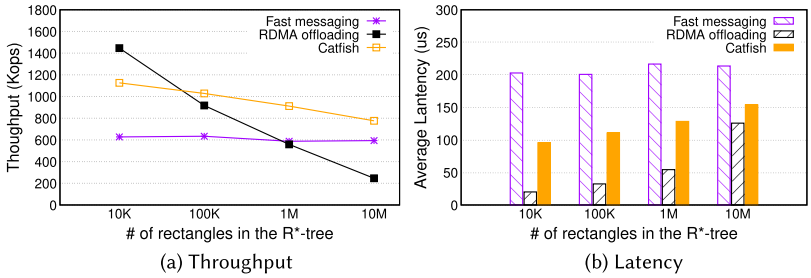


Fig. 23. The performance of Catfish, fast messaging, and RDMA offloading on the various sizes of R*-tree.

range of (0, 0.0001]. For each tree node, 30 entries could be contained before splitting. The heights of the R*-trees are hence 3, 4, 5, and 6. We also setup the requests at the scale of the power-law distribution as that in the prior experiments and launch 128 clients on 8 nodes evenly that send 20K search requests per client. Figure 23(a) presents the search throughput on varying sizes of R*-trees in Kops, from which we observe that RDMA offloading and Catfish have the decreasing search throughput as the tree size increases. Such a performance trend is due to more RTTs are required to retrieve the tree nodes from the larger trees. Since Catfish prefer processing the search requests on the server and only offload the tree traversal if the server CPUs are saturated, its read throughput declines much slower than RDMA offloading. For fast messaging, it maintains the stable throughput when the tree grows. This is because the major increasing overhead of fast messaging is the memory accesses on the server, which are relatively trivial in the complete processing pipeline. Figure 23(b) reports the average search latency of our experiments, where the results present the similar performance trend: With the tree size increases, RDMA offloading requires a higher latency to process a search request while fast messaging has a stable query completion time. Catfish makes a good compromise between them.

5.3.2 Varying Tree Node Sizes. The R-tree structure could change if the number of entries in a node varies, so we modify how many children (rectangles) are hold in an R-tree internal (leaf) node. We let a tree node contain 15, 30, 70, and 130 entries so that an R*-tree of 1M rectangles will grow to the heights of 6, 5, 4, and 3, respectively. The other configurations are same as those in Section 5.3.1, and we still test the performance numbers in terms of throughput and latency. Figure 24(a) presents the search throughput in Kops and Figure 24(b) shows the average latency of queries in microseconds. From the figures, we notice that changing the tree node size hardly affects fast messaging and Catfish. In these two schemes, most requests are handled at the server side, so the impacts of changing the traversal paths are constrained on the server and it contributes little to the overall performance. However, the performance of RDMA offloading adapts to the varying tree structures substantially. When we put more entries into one node and thus grow a shallower tree, the search throughput decreases and the latency increases. The reason is that RDMA offloading has to fetch the nodes with more irrelevant rectangles, and this consumes both additional bandwidth and time.

5.3.3 Queries with the Refinement Stage. In practical systems, the results of the window queries commonly need an additional refinement step to check if they really fulfill the query conditions [7]. This will also impacts the performance of our RDMA-enabled R-tree if the refinement operation is carried out on different places: For fast messaging, the refinement could be placed on the server node and only the reduced results are sent back. For RDMA offloading, we have to fetch and check all results on the client nodes. To evaluate the impacts of the refinement stage, we configure our

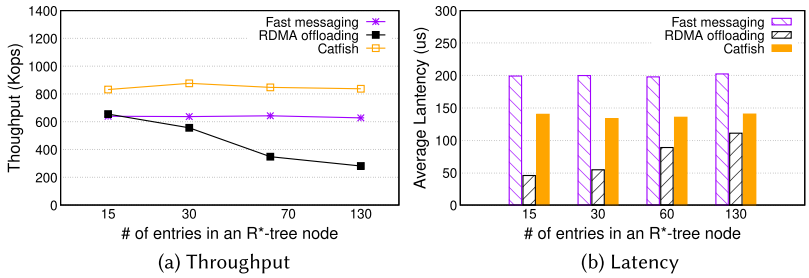


Fig. 24. The performance of Catfish, fast messaging, and RDMA offloading on the R*-trees with various node sizes.

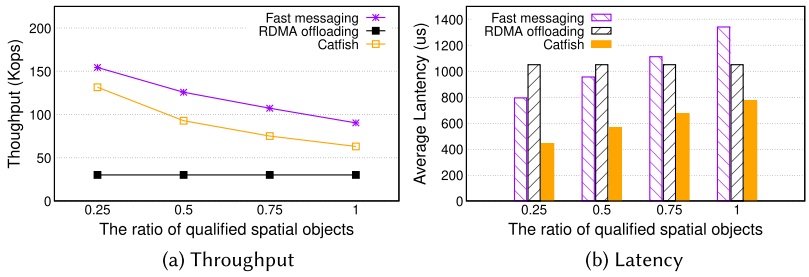


Fig. 25. The performance of Catfish, fast messaging, and RDMA offloading on the R*-trees with the refinement stage.

R*-tree of 1M rectangles to return partial results of the search queries, mimicking that only 25%, 50%, 75%, and 100% spatial objects could fulfill the query conditions. It is worth noting that this only affects fast messaging and Catfish. RDMA offloading still needs to read all relevant tree nodes. To highlight the impacts on the bandwidth from the refinement step, the search queries are randomly defined in the range of (0, 0.1], at most covering 1/10 of the whole space. The results are shown in Figures 25(a) and 25(b), which report the throughput and the latency, respectively. From the experimental results, we can see that fast messaging and Catfish have increasing bandwidth and decreasing latency if more spatial objects are recognized unqualified in the refinement stage. This is reasonable, since less bandwidth is consumed. Another interesting finding is that fast messaging outperforms Catfish. The inferior performance results from that Catfish determines to offload the queries if the server CPUs are busy, but it has no knowledge how much bandwidth will be cost in the offloading. In these extreme cases, the RDMA bandwidth is quickly saturated by the offloaded queries, which in turn blocks the queries handled by the server node.

5.4 Tests on a Real-world Dataset

We also evaluate Catfish by using a real world dataset *rea02* [5]. This dataset is composed of 1,888,012 rectangles for street segments in California, U.S. The rectangles are grouped as sub-regions, which have roughly 20,000 objects. The 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.

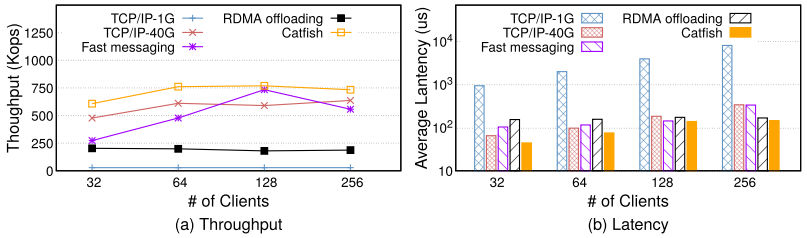


Fig. 26. The performance comparisons of R-tree search on the real dataset *rea02*.

The experimental results are shown in Figures 26(a) and 26(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 2.23× (over fast messaging), 4.28× (over RDMA offloading), and 27.25× (over TCP/IP-based). It reduces the average latency by up to 2.32×, 3.47×, and 56.09×.

6 THE APPLICATION SCOPE OF CATFISH

Although we focus on improving R-tree by Catfish in this article, 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 article, 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.

When building a distributed system storing data objects, two solutions could be considered: (1) Different objects are dispatched onto different servers according to the keys, and so are the queries; (2) The storage node holds all the data objects, i.e., keys and values, and queries are evenly handled by the servers that actually connects to the database and memcache the previous query results. Both the methods have to build the index structure, e.g., the R-tree for the spatial data, on each server node for accelerating the data accesses. In this article, we build our R-tree in a single-server–multiple-clients model for two reasons: (1) The index of geospatial data could be straightforwardly extended to multiple nodes. The geospatial data are naturally divided by their geographical coordinates and the partitions are kept on different nodes. Thus, the index structure could be extended to multiple servers as well. (2) Strong spatial locality is commonly found in the geospatial queries. For example, we mostly limit the query of finding restaurants in the neighboring areas. As a result, when straightforwardly extending the RDMA-enabled R-tree to multiple server nodes with the two solutions aforementioned, a large number of queries still concentrate in a single-server node, potentially leading to the performance issue that we strike to solve in this article.

As Catfish offloads the tree traversal to the clients, one can migrate the computation to the GPUs instead [26, 27, 40]. Despite that the GPU improves the R-Tree performance by about two orders of magnitude [40], the additional computation resources will be used up soon if the node holds “hot-spot” data or the dimension of data increases. We consider Catfish is orthogonal to the GPU-based techniques and they could be applied to R-trees simultaneously.

7 RELATED WORK

R-tree [16] as an essential indexing technique for spatial objects has been studied for decades and there are numerous R-tree variants proposed. R+-tree [45] guarantees that entries in intermediate nodes do not overlap each other. In this way, the point query is accelerated as the traversal path does not diverge. For the R*-tree [4], a new entry is always inserted to the MBR that leads to the minimum area expansion. One of space-filling curves, specifically the Hilbert curve, is used to linearly order the rectangles in Hilbert R-tree [25]. IR-tree [32] is designed to index both the textual and spatial content so that geographic queries with textual information are executed efficiently. PR-tree [3] promises that a window query can be answered within limited I/Os even in the worst case. Another research thread focuses on building an R-tree by packing spatial objects into leaf nodes in a bottom-up fashion. Ordering the high-dimensional objects is the key to this type of techniques. The sorting could be carried out according to one of the coordinates [29, 43] or the position on a space-filling curve [10, 17, 25]. Qi et al. [41] proposes a novel packing strategy that builds an R-tree with bounded window query performance as the PR-tree. This is achieved by mapping data points into a rank space before packing. Besides scientific and spatial databases, R-tree is adopted by many big data systems deployed in data centers, including SpatialHadoop [14], Hadoop-GIS [2], Simba [56], LocationSpark [48], iSPEED [33], DITA [46], and so on. All of these systems need to access R-tree or its variants 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 [12, 13, 21, 22, 38, 42, 52, 53] are prevalent in data centers. Many researchers start to explore RDMA for reducing the communication overhead. Jose et al. [21] improve the *Put* performance of key-value store by using RDMA, where 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. Pilaf [38] is a key-value store on RDMA. The client directly reads a key-value pair in the server via RDMA Read and sends write requests to the server via RDMA Write. This is a hybrid design using fast messaging and RDMA offloading, but without the adaptive design and the optimizations proposed in this work. HERD [22] is another key-value store on RDMA that uses RDMA Write to send both *Put* and *Get* requests to the server, but the server sends responses back via RDMA Message on **Unreliable Datagram (UD)**. As UD is not reliable, the key-value store itself needs to support the reliable transport. Jose et al. [20] 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. C-Hint [52] is a client-side cache system on RDMA. It 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 [53] combines C-Hint and the out-of-place update on server. FaRM [12] is a distributed memory system on RDMA. It 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 and resolve the read-write conflict on RDMA.

Cell [39] 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 [35] uses RDMA to implement RPC in Hadoop, bypassing the JVM memory management for high performance. HatRPC [31] uses a hint scheme to optimize heterogeneous RPC services and functions. FaSST [24] is a fast and scalable RPC system that uses two-sided RDMA Message on the unreliable connected and UD transports. This method requires the server to handle network communication. Su et al. [47] show the asymmetric in-bound and out-bound performance of RDMA network cards, and propose Remote Fetching Paradigm to reduce the out-bound overhead on server for RDMA-enabled RPC.

Liu et al. [34] use RDMA Message on UD to optimize the shuffle operators for database queries. Salama [44] design an overlapping mechanism to overlap the RDMA read on network and local query processing. Several studies [6, 9, 54, 58] 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. [23] reveal the poor performance of RDMA Atomics. Dragojevic et al. [13] construct distributed transactions on top of FaRM to get better performance than RDMA Atomic-based solutions.

8 CONCLUSION

In this article, we present Catfish, an RDMA-enabled R-tree to achieve low latency and high throughput. The R-tree data structure is commonly used in scientific and spatial databases. Billions of users frequently access these databases via the Internet in daily life to seek locations for shopping, travel, entertaining, and other activities. In practice, a customer or a user makes a request to a specific Web server via the Internet. The Web server connects to a datacenter where back-end servers use the R-tree data structure for data processing. In an in-memory computing environment, R-tree is memory-resident. Our work for this type of data processing environment focuses on the performance and efficiency of CPUs in both Web and back-end servers and the effectiveness of the network between the Web server and the back-end servers. These three factors determine the performance and throughput of the R-tree-based data processing.

By intensive experiments in a conventional R-tree data processing system, we have observed that both the CPU and network bandwidth of back-end servers can easily become bottlenecks for accepting and processing highly simultaneous requests from Web servers. In this environment, the Web server is a client to make requests to back-end servers. Our experiments show three insights into the poor performance behavior. First, CPUs on the client side are often lightly loaded, because they do not have heavy data processing duties. Second, when the back-end servers are heavily loaded, the network is often lightly loaded. Finally, when the network becomes a bottleneck, the server's load may be reasonable. Our findings motivate us to develop an RDMA-based R-tree data processing system. The Catfish system harnesses the idle CPU cycles on the client side (Web servers) and best utilizes the available network bandwidths (the connection between the Web servers and the back-end servers), which makes the R-tree data processing on the back-end servers in 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their encouragement, constructive comments, and suggestions.

REFERENCES

[1] A. M. Abdullah. 1997. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification. https://www.iith.ac.in/~tbr/teaching/docs/802.11-2007.pdf.

[2] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop GIS: A high performance spatial data warehousing system over mapreduce. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1009–1020.

[3] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. 2004. The priority R-Tree: A practically efficient and worst-case optimal R-Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. Association for Computing Machinery, New York, NY, 347–358.

[4] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. 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* (Atlantic City, New Jersey) *(SIGMOD'90)*. ACM, New York, NY, 322–331.

[5] Norbert Beckmann and Bernhard Seeger. 2008. A benchmark for multidimensional index structures. Retrieved from http://www.mathematik.uni-marburg.de/seeger/rrstar/index.html.

[6] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.* 9, 7 (Mar. 2016), 528–539.

[7] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1994. Multi-Step processing of spatial joins. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*. Association for Computing Machinery, New York, NY, 197–208.

[8] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the 14th EuroSys Conference 2019*. 1–14.

[9] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. ACM, New York, NY, Article 26, 17 pages.

[10] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. 1994. Client-Server paradise. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*. Morgan Kaufmann Publishers, San Francisco, CA, 558–569.

[11] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2017. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.* 40, 1 (2017), 3–14.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, 401–414.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 54–70.

[14] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A mapreduce framework for spatial data. In *Proceedings of the IEEE 31st International Conference on Data Engineering (ICDE'15)*. IEEE, 1352–1363.

[15] Google. 2005. Google Maps. Retrieved from https://maps.google.com.

[16] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'84)*. ACM, New York, NY, 47–57.

[17] Herman Haverkort and Freek V. Walderveen. 2008. Four-Dimensional hilbert curves for R-Trees. *ACM J. Exp. Algorithmics* 16, Article 3.4 (Nov. 2008), 19 pages.

[18] Nusrat S. Islam, Md. Wasi-Ur-Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. 2012. High performance RDMA-based design of HDFS over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, Los Alamitos, CA, Article 35, 35 pages.

[19] Anand Padmanabha Iyer and Ion Stoica. 2017. A scalable distributed spatial index for the internet-of-things. In *Proceedings of the Symposium on Cloud Computing (SoCC'17)*. ACM, New York, NY, 548–560.

[20] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md. Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K. Panda. 2012. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'12)*. IEEE Computer Society, Washington, DC, 236–243.

[21] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. 2011. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. IEEE Computer Society, Washington, DC, 743–752.

[22] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the ACM Conference on SIGCOMM (SIGCOMM'14)*. ACM, New York, NY, 295–306.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'16)*. USENIX Association, Berkeley, CA, 437–450.

[24] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. 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 (OSDI'16)*. USENIX Association, Berkeley, CA, 185–201.

[25] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*. 500–509.
[26] Mincheol Kim, Ling Liu, and Wonik Choi. 2018. A GPU-aware parallel index for processing high-dimensional big data. *IEEE Trans. Comput.* 67, 10 (2018), 1388–1402.
[27] Mincheol Kim, Ling Liu, and Woink Choi. 2021. Multi-GPU efficient indexing for maximizing parallelism of high dimensional range query services. *IEEE Trans. Serv. Comput.* (2021). https://www.computer.org/csdl/journal/sc/5555/01/09430517/1tzuflEV6Vy.
[28] Marcel Kornacker and Douglas Banks. 1995. High-Concurrency locking in R-Trees. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB'95)*. Morgan Kaufmann Publishers, San Francisco, CA, 134–145.
[29] S. T. Leutenegger, M. A. Lopez, and J. Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the 13th International Conference on Data Engineering*. 497–506.
[30] Mingzhe Li, Khaled Hamidouche, Xiaoyi Lu, Hari Subramoni, Jie Zhang, and Dhabaleswar K. Panda. 2016. 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, 37.
[31] Tianxi Li, Haiyang Shi, and Xiaoyi Lu. 2021. HatRPC: Hint-Accelerated thrift RPC over RDMA. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)*. Association for Computing Machinery, New York, NY, Article 36, 14 pages. https://doi.org/10.1145/3458817.3476191
[32] Zhisheng Li, Ken C. K. Lee, Baihua Zheng, Wang-Chien Lee, Dik Lee, and Xufa Wang. 2011. IR-Tree: An efficient index for geographic document search. *IEEE Trans. Knowl. Data Eng.* 23, 4 (2011), 585–599.
[33] Yanhui Liang, Hoang Vo, Jun Kong, and Fusheng Wang. 2017. 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 (SIGSPATIAL'17)*. ACM, New York, NY, Article 17, 10 pages.
[34] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*. ACM, New York, NY, 48–63.
[35] Xiaoyi Lu, Nusrat S. Islam, Md. Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. 2013. High-Performance design of Hadoop RPC with RDMA over infiniband. In *Proceedings of the 42nd International Conference on Parallel Processing (ICPP'13)*. IEEE Computer Society, Washington, DC, 641–650.
[36] Miao Luo, Dhabaleswar K. Panda, Khaled Z. Ibrahim, and Costin Iancu. 2012. Congestion avoidance on manycore high performance computing systems. In *Proceedings of the 26th ACM international conference on Supercomputing*. 121–132.
[37] Robert M. Metcalfe and David R. Boggs. 1976. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM* 19, 7 (1976), 395–404.
[38] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient Key-value store. In *Proceedings of the USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, 103–114.
[39] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and network in the cell distributed B-tree store. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference* (Denver, CO) *(USENIX ATC'16)*. USENIX Association, Berkeley, CA, 451–464.
[40] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. 2015. GPU-based parallel R-tree construction and querying. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 618–627.
[41] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2018. Theoretically optimal and empirically efficient R-Trees with strong parallelizability. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 621–634.
[42] An Qin, Mengbai Xiao, Jin Ma, Dai Tan, Rubao Lee, and Xiaodong Zhang. 2019. DirectLoad: A Fast Web-scale Index System across Large Regional Centers. In *Proceedings of the IEEE 35th International Conference on Data Engineering (ICDE'19)*. IEEE.
[43] Nick Roussopoulos and Daniel Leifker. 1985. Direct spatial search on pictorial databases using packed R-Trees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'85)*. Association for Computing Machinery, New York, NY, 17–31.
[44] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. 2017. Rethinking distributed query execution on high-speed networks. *Data Eng.* 40, 1 (2017), 27–37.
[45] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB'87)*. Morgan Kaufmann Publishers, San Francisco, CA, 507–518.
[46] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. 2018. DITA: Distributed in-memory trajectory analytics. In *Proceedings of the International Conference on Management of Data (SIGMOD'18)*. ACM, New York, NY, 725–740.
[47] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is faster than server-bypass with RDMA. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*. ACM, New York, NY, 1–15.

[48] Mingjie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A distributed in-memory data management system for big spatial data. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1565–1568.
[49] Mellanox Technologies. 2015. Performance Tests (perftest) package for OFED. Retrieved from https://github.com/linux-rdma/perftest.
[50] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 306–324.
[51] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17)*. ACM, New York, NY, 1009–1024.
[52] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. 2014. C-Hint: An effective and reliable cache management for RDMA-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'14)*. ACM, New York, NY, Article 23, 13 pages.
[53] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. 2015. 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 (SC'15)*. ACM, New York, NY, Article 22, 11 pages.
[54] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 87–104.
[55] Mengbai Xiao, Hao Wang, Liang Geng, Rubao Lee, and Xiaodong Zhang. 2019. Catfish: Adaptive RDMA-enabled R-Tree for low latency and high throughput. In *Proceedings of the IEEE 39th International Conference on Distributed Computing Systems (ICDCS'19)*. IEEE Computer Society, Washington, DC, 164–175.
[56] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient in-memory spatial analytics. In *Proceedings of the International Conference on Management of Data (SIGMOD'16)*. ACM, New York, NY, 1071–1085.
[57] Yelp. 2004. Yelp Inc. Retrieved from https://www.yelp.com.
[58] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.* 10, 6 (Feb. 2017), 685–696.

Received March 2021; revised September 2021; accepted December 2021