Information Sciences 502 (2019) 376–393

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

RDMA-driven MongoDB: An approach of RDMA enhanced NoSQL paradigm for large-Scale data processing

Bobo Huang[a], Li Jin[a], Zhihui Lu[a,e,*], Ming Yan[a,b,*], Jie Wu[a,b], Patrick C.K. Hung[c], Qifeng Tang[d]

[a] School of Computer Science, Fudan University, Shanghai 200433, China
[b] Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, Shanghai, China
[c] Faculty of Business and IT, University of Ontario Institute of Technology, Canada
[d] Shanghai Data Exchange Corporation, Shanghai, China
[e] Shanghai Blockchain Engineering Research Center, Shanghai, 200433, China

ARTICLE INFO

Article history:
Received 27 January 2019
Revised 4 June 2019
Accepted 16 June 2019
Available online 17 June 2019

Keywords:
RDMA
NoSQL
Big data
One-sided communication
RoCE
Kernel-bypass network

ABSTRACT

With the rapid development of big data and data center networks, NoSQL database has won great popularity for its excellent performance in accelerating the performance of many online and offline big data applications, such as HBase, Cassandra and MongoDB. However, due to massive and frequent Create/Update/Retrieval/Delete (CURD) operations, the traditional TCP/IP protocol stack has difficulty to provide the required request rates and response latency for the large-scale NoSQL system. For example, large-scale data migration or synchronization among multiple clusters in a data center results in competition for network bandwidth with high delay. To mitigate such transmission bottleneck, we propose an approach of RDMA-driven document NoSQL Paradigm' RDMA_Mongo, based on MongoDB. The performance of CURD operations is enhanced by one-sided Remote Direct Memory Access (RDMA) primitives (such as RDMA Read/Write) without involving the TCP/IP stack or CPU. Evaluation under RDMA-enabled network demonstrates that RDMA_Mongo significantly improves the CURD performance, compared with plain MongoDB. The results show that the average insert throughput increases by approximately 30%, the average delete throughput by over 30%, the update by up to 17% and the query throughput by 15% when facing large-scale data requests.

1. Introduction

With the rapid increase of large-scale data, which needs to be collected and analyzed online or offline in a data center, the distributed NoSQL (Not Only Structured Query Language) database systems such as MongoDB have been widely adopted in both academia and industry. NoSQL databases are designed not only to cope with the challenges in scalability and agility, which modern applications are confronted with, but also to take advantages of the commodity storage and computing power available today. As a typical implementation of NoSQL databases, MongoDB employs flexible document data model, auto-sharding, replica sets as core features, and is the only NoSQL database to support multi-document transactions currently. CURD document operations, replica sets and auto-sharding in MongoDB all rely on the underlying network communication.

* Corresponding author.
E-mail addresses: huangbb16@fudan.edu.cn (B. Huang), ljin18@fudan.edu.cn (L. Jin), lzh@fudan.edu.cn (Z. Lu), myan@fudan.edu.cn (M. Yan), jwu@fudan.edu.cn (J. Wu), patrick.hung@uoit.ca (P.C.K. Hung), keven@chinadep.com (Q. Tang).

https://doi.org/10.1016/j.ins.2019.06.048
0020-0255/

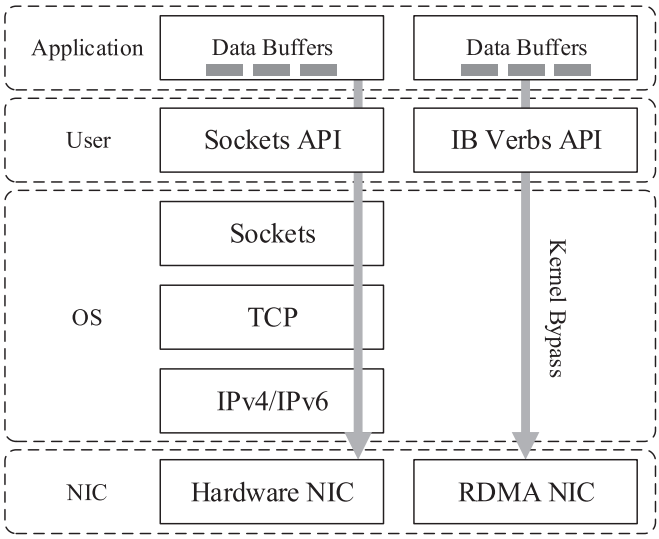

Fig. 1. The difference between TCP and RDMA.

Furthermore, networking for big data [50] is extremely vital to accommodate the demand for big data processing. In the case of high concurrency of big data processing, it is found in our evaluation that the transmission load using traditional network protocol stack has become a bottleneck of MongoDB.

Meanwhile, as an emerging underlying network technology in recent years, RDMA (Remote Direct Memory Access), especially RoCE (RDMA over Converged Ethernet), has been frequently adopted in many industrial datacenters [10,14,16,25,38,41,43]. It offers ideal high throughput, low latency and CPU-bypassing for big data applications like HDFS [13,36], memory file system [22], analysis and modeling on big data [5,33]. RDMA network protocol uses zero-copy technology and bypass OS kernel to directly access registered memory in remote host.

Fig. 1 shows the difference between TCP and RDMA. In terms of TCP, the data in application buffer is encapsulated by the sockets API in user space. Then the encapsulated data is written to network card through the sockets and TCP/IP layer protocols of OS kernel. However, in terms of RDMA, the data in application buffer is encapsulated by IB Verbs API in user space, and the encapsulated data is then directly written to the RDMA network card (RNIC) without any involvement of the operating system kernel.

The mainstream RDMA technologies include InfiniBand, RoCE and iWARP. RDMA primitives can be divided into message-oriented two-sided verbs (such as RDMA SEND/RECV) and memory-sharing one-sided verbs (such as RDMA READ/WRITE) [26,27]. The message-passing communication requires that the receiver should start the transmission session and register a communication buffer. After that, the sender prepares to modify the data in the remote receiver's memory buffer and accomplishes the data exchanges. However, the problem is, that in such a message-oriented mode, data copy between data memory buffer and communication memory buffer is still unavoidable. The RDMA shared-memory mode merges the data memory buffer and communication memory buffer by allowing sender and receiver sides to remain completely passive. Existing RDMA driven data center applications, called RDMA-Enhanced Paradigm, fall into two categories: RDMA-Enhanced System [4,13–20,22,28,29,36,38,39,43,44,46,47,49] and RDMA-Enhanced Algorithm [2,6,35,42]. For instance, Nessie [4] proposed a fully client-driven key-value store system using RDMA for high performance. APUS [42] proposed the first RDMA-based Paxos consensus algorithm. Existing RDMA-driven systems mainly focus on distributed file system [13,20,22,36], key-value store [4,14–16,28,38,43,44,49], parallel database [19], relational database [18,29], as well as memory transaction [17,46,47]. However, to the best of our knowledge, there have been no attempts to enhance document-based NoSQL databases in RDMA-enabled networks. Therefore, can we accelerate document-based NoSQL system with RDMA to mitigate the transmission bottleneck?

This paper presents RDMA_Mongo, which is the first RDMA driven document-based NoSQL paradigm. We first demonstrate a detailed analysis of MongoDB network transport designs. Through analyzing the key stages in the NoSQL CURD operations with high concurrency, we find that there are two key challenges: (1) to decrease frequent CPU context switching caused by data transmission in traditional network and (2) to reduce waiting time on the MongoDB client side.

To address the aforementioned challenges, specifically, RD-MA_Mongo automatically detects whether RDMA NICs and related libraries (such as libibverbs, librdmacm) are supported at both local and remote hosts, and then determines whether to use RDMA verbs or traditional network sockets. Second, based on the current system load (especially memory usage), RDMA_Mongo determines the appropriate buffer size on demand and registers RDMA communication memory regions.

Third, RDMA_Mongo introduces non-blocking data transmission with the power of RDMA Completion Queue (CQ). Without modification on the MongoDB existing network transport interface, RDMA_Mongo rewrites the socket implementation

exploiting RDMA one-sided shared-memory primitives, which can coexist with traditional network transport mode. Finally, indigenous MongoDB exploits the replica set mechanism to achieve high availability, that is, the MongoDB cluster has one primary node and multiple secondary nodes at the same time. The secondary nodes need to continuously pull oplogs (Operation Log) from the primary node and replay the idempotent oplogs for database synchronization. To accelerate such process, we optimize and redesign the oplogs synchronization protocol based on RDMA primitives.

The main contributions of our work can be summarized as follows.

- We analyze thoroughly all the various design choices of RDMA-driven paradigm and demonstrate an effective trade-off among different design options for high performance RDMA_Mongo.
- We put forward a RDMA context detection algorithm for determining TCP/IP-based or RDMA communication between two RDMA_Mongo nodes and also propose load-aware buffer registration mechanism for reasonable memory region management of RDMA channels.
- We redesign the oplogs synchronization protocol with RDMA primitives in RDMA_Mongo, including the optimization of oplogs initial sync and steady state replication, in which RDMA completion event channel is used to receive messages asynchronously.
- We have implemented RDMA_Mongo based on MongoDB version 4.1.1-59. The detailed evaluation in RDMA-enabled network demonstrates that RDMA_Mongo significantly improves the CURD performances in average insert by approximately 30%, update by 17%, query by 15% and delete throughput by over 30%, when facing large-scale data.

The following section, that is Section 2, describes the typical NoSQL categories, RDMA and recent efforts on RDMA-Enhanced Paradigm, then introduces the problem statement. Section 3 demonstrates the overview of RDMA_Mongo architecture and discusses the tradeoff among different design choices. In Section 4, RDMA context detection algorithm and load-aware buffer registration mechanism are proposed, and the detail of RDMA-enabled oplogs synchronization is given. In Section 5, we evaluate our system and compare it against the plain MongoDB. In Section 6, we make extensive investigations of recent works related to modern network hardware, kernel-bypass networking, RDMA protocol stack optimization, and RDMA-driven NoSQL. Section 7 comes to the conclusion of this paper and our expectation for future work.

2. Background and motivation

2.1. NoSQL

NoSQL(Not Only SQL) database generally refers to the non-relational database [32]. With the increase of data scale, traditional relational database is no longer capable to cope with such large amounts of data. Therefore, NoSQL database becomes more and more popular among developers [24]. The concept was first used in 1998 and was picked up in 2009 [11]. There are four mainstream data models of NoSQL database: Key-value, Column-oriented, Document and Graph [11,24]. Table 1 shows the NoSQL Category and RDMA-driven paradigm in recent years. We have found that many researches focus on key-value store and graph databases. However, there is not yet relevant researches on RDMA-driven document NoSQL system. Therefore, we propose a first approach of MongoDB Paradigm — RDMA_Mongo.

Key-value Data Model: A key-value database uses a key to match a value. Key-value store maintains a hash map or dictionary which contains a set of records. Different records are identified and retrieved by different keys. Key-value store providers fast retrieval, high scalability and concurrency. However, complex conditional queries cannot be carried out in key-value stores such as Redis, MemcacheDB and Totyo Tyran.

Column-oriented Data Model: In a column-oriented database, data is stored by columns. The advantages of this data model include:1) extremely high loading speed; 2) suitable for large amounts of data; 3) efficient compression ratio; 4) more suitable application on aggregation and data warehouse. However, it is not suitable for scanning small data, random updates, real-time deletes and updates [1]. HBase [8] is a commonly used column-oriented database.

Document Data Model: The structure of document database is similar to that of key-value database, but the query over large-scale data is more efficient. Document-oriented databases store semi-structured documents and encode data in specific formats including JSON, XML and YAML. In addition, a secondary index is supported in document-oriented NoSQLs to boost more efficient queries. MongoDB [31] is one of the typical document databases.

Table 1
Researchs on RDMA-driven NoSQL databases in recent years.

NoSQL Category	RDMA-driven Paradigm
Key-Value Data Model	HydraDB [44]
	Nessie [4]
	InnerCache [49]
	HERD [16]
	Hybrid Memcached [14,38]
Graph Data Model	Wukong [39]
Document Data Model	**RDMA_Mongo**

Graph Data Model: Flexible graph structures are exploited by graph DB for higher scalability and faster semantic queries. Graph databases employ nodes, edges and properties to represent and store data items. The relationships between different data entities are abstracted as edges, which boost faster queries over heavily inter-connected data. Graph models can be summarized into two categories: labeled-property graph and resource description framework (RDF). Typical graph databases include Neo4J [45] and Infinite Graph. RDF query with RDMA is proposed in Wukong [39].

2.2. RDMA

Emerging RDMA technology provides ultra-low latency and high throughput via zero-copy and bypassing the remote OS kernel [19]. It writes data directly into pre-registered remote memory regions through kernel-bypass network and moves data from local memory to remote memory without any CPU involvement. It eliminates the overhead of external memory replication and context switching.

Zero-copy technology allows RDMA NICs (RNIC) to transfer data directly with the registered memory regions, avoiding data copies between user space and kernel space and between kernel space and RNIC. The application sends commands to NIC without executing system calls. Bypassing OS kernel, RDMA requests are sent from user space to local NIC, then to remote NIC through inter-connect kernel-bypass network, which reduces the number of context switching between kernel memory space and user space.

There are two fast communication types on RDMA: one-sided read/write communication and two-sided send/recv communication. An one-sided RDMA can directly write from local memory to remote memory without involving OS kernel as mentioned above, while two-sided RDMA needs the participation of CPU. Therefore, one-sided RDMA is about 2X faster than two-sided RDMA [17]. Each link between local NIC and remote NIC will maintain a QP(Queue Pair), which contains a send queue and receive queue. As a global data structure, one QP can push the data into local memory within 0.2us [17]. With the increasing connections, the number of QP is also increased. Each QP maintains a CQ(Completion Queue), which is a FIFO (First Input First Output) queue and contains all completed Work Requests (WR). Every completed WR is called as a Completion Queue Entry (CQE) and is posted to the Work Queues (WQ). Generally, once a CQE is polled by an asynchronous daemon, it will be removed from the CQ.

2.3. RDMA enhanced paradigm

Fig. 2 shows many emerging research works on exploring the power of RDMA in recent years. The prior works can be roughly divided into two categories: RDMA-enhanced system [4,13–20,22,28,29,36,38,39,43,44,46,47,49] and RDMA-enhanced algorithm [2,6,35,42].

Regarding RDMA-enhanced systems, distributed file system for big data [13,20,22,36], key-value store [4,14–16,28,38,43,44,49], parallel database [19], relational database [18,29], graph computing [39], RPC library [17,40], distributed memory transaction [17,46,47] have employed RDMA to achieve high performance. Pilaf [28] achieves high-performance key-value store with one-sided RDMA Read, while proposing self-verifying data structures to guarantee cache consistency under concurrent memory modifications. Octopus [22] redesigns data/metadata operations mechanisms of distributed memory file system by closely combining RDMA and NVM. Wukong [39] is a RDMA-boosted distributed in-memory RDF graphs store that puts forward RDMA-driven graph exploration for more efficient and lower-latency RDF queries over large-scale data.

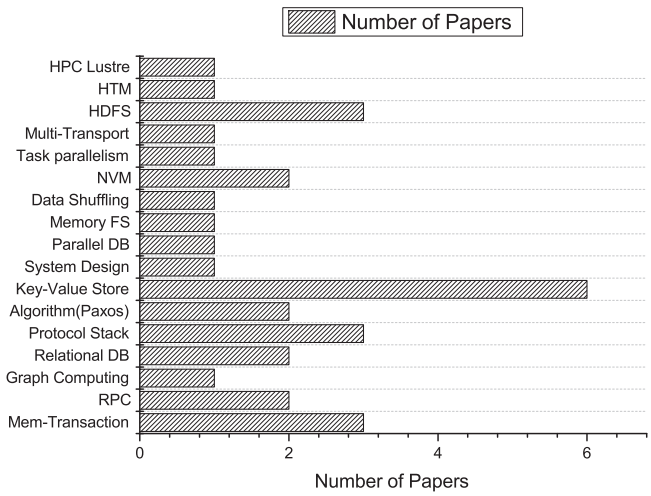

Fig. 2. The research point of RDMA papers published in OSDI, ATC or other famous conferences in recent years.

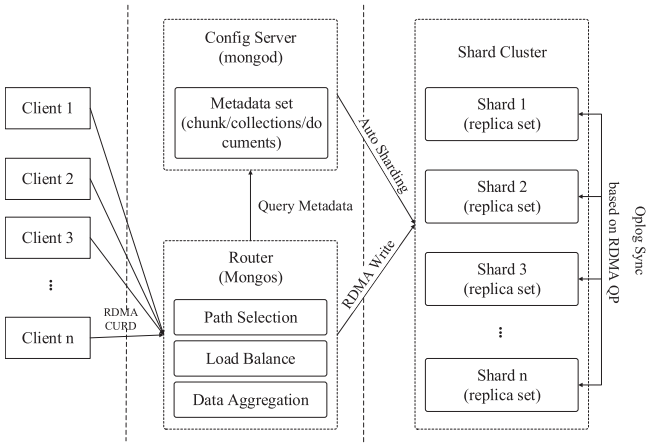

Fig. 3. RDMA_Mongo Architecture.

DrTM+H [46] presents a distributed memory transaction system which uses multiple RDMA primitives in a mixed manner to achieve optimal performance for each transactional execution phase.

For RDMA-enhanced algorithm, algorithms such as multi-tasks scheduling [2,6] and consensus algorithm [35,42] have been optimized on RDMA-enabled networks. Uni-Address [2] is a RDMA-based thread management scheme with scalable work stealing for high-performance distributed multi-tasks scheduling. DARE [35] leverages RDMA verbs to accelerate state machine replication protocol with wait-free log replication, which achieve higher request rates and lower latency for write/read requests. APUS [42] uses RDMA primitives to replace traditional TCP/IP in Paxos protocol targeting to lower consensus latency, which outperforms DARE [35] by 4.9X.

2.4. Motivation

In many data centers, big data processing platforms reply on document-based NoSQL as underlying data store and migration. However, data transmission based on traditional Ethernet under NoSQL involve excessive CPU overheads, and intolerable latency and throughput can't satisfy online transaction processing. Moreover, existing RDMA-enhanced NoSQL systems are mainly key-value store [4,14–16,28,38,43,44,49], column-oriented store [12] and graph-based RDF store [39]. There is still no RDMA optimization special for document-based NoSQL. Thus, inspired from the emerging RDMA-enhanced paradigm, we attempt to leverage RDMA primitives to accelerate data transmissions and oplogs synchronization in document-based NoSQL cluster for higher throughput and lower delay without CPU involvement.

3. RDMA_Mongo design

RDMA_Mongo is a RDMA-aware design for document-based MongoDB while effectively taking advantage of RDMA design space for MongoDB transport layer. In this section, we first demonstrate the overview of RDMA_Mongo, including system architecture and RDMA-involved key communication between different components. Afterwards, we describe the overall design space for RDMA-enhanced system optimization. Finally, we discuss the tradeoff among different design choices with the aim of higher throughput and lower latency.

3.1. Overview

RDMA_Mongo replaces the transport layer with RDMA verbs while keeping the original architecture unchanged. Fig. 3 shows RDMA_Mongo architecture. RDMA_Mongo involves three key components: mongos as router, mongod as config server and shard for shaded data. Among these components, there are mainly three stages in which the data transmission load is relatively large, including the CURD operations between client and mongos, data read/write between mongos and shard cluster, and oplog sync among shard nodes. RDMA_Mongo employs RDMA-aware transport layer to accelerate the aforementioned three critical transmission stages.

Next, we zoom into the design space and elaborate that how they work together can gain high performance with tradeoffs among different design choices.

3.2. RDMA-driven paradigm

Fig. 4 shows that RDMA can work on three types of networks: Infiniband, RoCE (including two versions: v1 and v2), and iWARP. In the user-space API layer, all paradigms are based on IB verbs API. Developers can either use some libraries

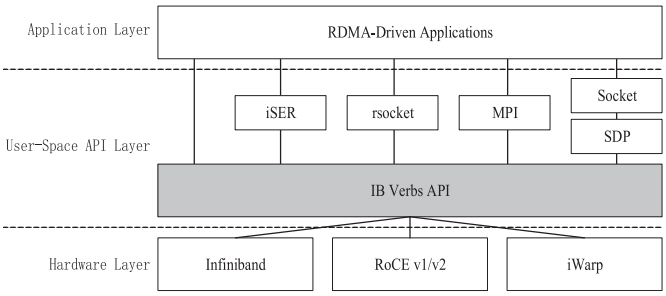

Fig. 4. RDMA-Driven Paradigm.

based on IB verbs such as iSER, rsocket, MPI and SDP, or use IB Verbs API directly to develop an RDMA-driven application. Although the libraries provide great convenience for development, the performance will be severely decreased due to the high overhead of socket-to-Verbs translation caused by these libraries. Therefore, to achieve the best performance, we use IB verbs API directly to develop our RDMA_Mongo without using any third-party libraries.

3.3. Design space

RDMA transfer progress includes RDMA device initialization, Queue Pair (QP) creation, memory registration (), QP metadata exchange, data send/receive/write/read and resource freeing. A QP is composed of a Send Queue (SQ) and a Receive Queue (SQ), and is associated with a Completion Queue (CQ). Work Requests (WRs) have to be posted asynchronously into QP for communication. Once the Work Request is serviced, a completion event will be trigged and pushed into associated CQ. The application then polls completion events from CQ for safely reusing memory regions. The associated design space can be mainly summarized into the following aspects.

RDMA transport types and functions: There are two transport service types for RDMA, including Reliable Connection (RC) and Unreliable Datagram (UD). The transport functions consist of message-oriented two-sided send/receive primitives and shared-memory read/write primitives. In connection-oriented RC mode, end-to-end transfer between n nodes needs Queue Pairs. In connectionless UD mode, end-to-end transfer between n nodes needs Queue Pairs only. RC service supports one-sided and two-sided communication while UD service only supports two-sided communication.

Message size for RDMA communication: In UD transport service, MTU is the upper limit of message size. In RC transport service, the maximum message size can be 1 GiB in Infiniband NIC. A smaller message size results in more Work Requests, which brings more CPU overhead during transfers. However, a larger message size requires more registered memory for RDMA communication, which leads to higher memory consumption. Hence, suitable message size is imperative for RDMA transfers.

Number of Queue Pairs per host: Determining the tailored number of Queue Pairs per host node requires a tradeoff between task parallelism and RDMA NIC (RNIC) on-chip memory consumption. Prior work [7] has proved that too many QPs per node can cause lower performance by up to 5x and easily overflow NIC on-chip cache in larger clusters. However, with high concurrency, too few QPs per node causes thread contention. With Reliable Connection mode, the number of QPs has to be equal to the number of connections and strictly limited by RNIC resource. With Unreliable Datagram mode, it is best practice to keep the number of QPs equal to that of CPU cores.

3.4. Design choices tradeoffs

RDMA two-sided mode vs. one-sided mode: Different RDMA transport modes have different overheads. With no acknowledgement packet, the Unreliable Datagram transport service type requires applications to handle out-of-order packets and error, which results in more CPU involvement. As every transmitted packet requires an ACK in RDMA NIC (RNIC), the Reliable Connection transport service type leads to high throughput with a simpler algorithm. When it comes to flow control, RDMA two-sided verbs (such as RDMA Send/Receive primitive) require applications to count transferred messages and continuously track the number of posted Send/Receive requests between the sender and the receiver. As a result, this causes frequent CPU context switching and multiple data copies between communication memory buffer and data memory buffer. Meanwhile, the one-sided verbs only need to inform the sender whenever the receiver's message buffer can be safely freed for incoming overwriting. Hence, the paper adopts the more intuitive one-sided RDMA verbs with Reliable Connection.

RDMA fixed vs. on-demand message size: Fixed message size for RDMA communication is hard to adapt to dynamic system load. Considering the balance between memory consumption and current network throughput, the load-aware on-demand message size for RDMA communication is strongly expected.

Extending vs. remaining origin interface: Extending MongoDB transport layer interface by RDMA verbs leads to great modification cost on upper layer invoke. Therefore, with remaining no changes on origin transport interface, RDMA_Mongo

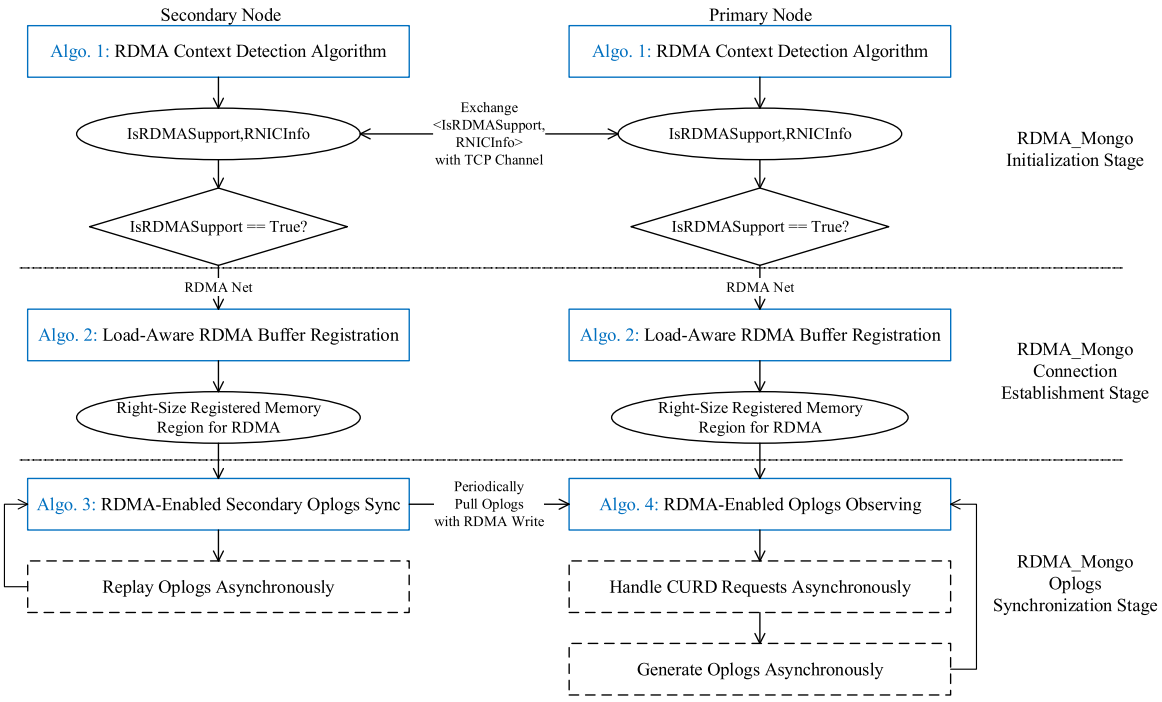

Fig. 5. The correlations among Algo. 1, 2, 3 and 4 employed by RDMA_Mongo.

replaces the socket implementation with RDMA one-sided Write/Read primitives, which can coexist with traditional TCP/IP network stack.

4. Implementation

This section presents a detailed introduction for RDMA-based context detection, buffer registration strategy, verbs selection and oplogs synchronization employed by RDMA_Mongo. RDMA context detection and load-aware buffer registration algorithm are executed during initialization and connection establishment phase, respectively. RDMA_Mongo oplogs synchronization phase contains RDMA-enabled oplogs fetching algorithm in the secondary node and oplogs observing algorithm in the primary node. The relationship between the aforementioned four algorithms employed by RDMA_Mongo is shown in Fig. 5.

4.1. RDMA Context detection algorithm

In RDMA-enabled transport, we should ensure that both client and server endpoints support interconnected RDMA hardware. A new configuration item named *is_rdma* is added to RDMA-_Mongo configuration file *mongod.conf*, in which *true* indicates RDMA hardware is supported and *false* indicates RDMA hardware is not supported. During RDMA_Mongo initialization, *mongod.conf file* is loaded by local node to extract user-defined configuration items. If *is_rdma* item is true, local node will query local available IB device name by *ibv_get_device_list()/ ibv_get_device_name()*, active device port by *ibv_query_port()*, gid (RDMA global address) index and bounded IP by *show_gids* command. Afterwards, one TCP channel between local and remote nodes will be established to synchronize local *is_rdma* and RNIC metadata to remote node, while remote RNIC metadata will be returned to local node. If the values of *is_rdma* for local and remote nodes are all true, RDMA channel between two sides will be established for data transfer. Otherwise TCP channel is used to transmit data. Algorithm 1 shows the aforementioned RDMA context detection algorithm.

4.2. Load-aware buffer registration algorithm

For RDMA_Mongo efficient transfers, the current memory usage and network load are crucial impact factors. We demonstrate an adaptive buffer registration algorithm for RDMA communication memory area, as shown in Algorithm 2. A slightly larger buffer size is desired with sufficient available memory and network while a smaller buffer size is expected with scarce free memory and network. In this paper, the product of *idle memory ratio* and *RNIC throughput ratio* is treated as overload factor to indicate the scarcity of resources. When this overload factor is less than a specified overload threshold *overloadThr*, we mark the load high and calculate the buffer size according to Formula 1. Otherwise, we mark the load low and use a

Algorithm 1 RDMA Context Detection Algorithm.

Input:
None
Output:
bool isRdmaSupport
struct remoteRnicInfo

1: *local_support, remote_support* ⇐ *Null*
2: *localRnicInfo, remoteRnicInfo* ⇐ *Null*
3: **if** local node rdma status is *True* **then**
4: *local_support* ⇐ *True*
5: *localRnicInfo.dev_name* ⇐ *local IB device name*
6: *localRnicInfo.ib_port* ⇐ *local IB device port*
7: *localRnicInfo.gid_idx* ⇐ *local IB device gid index*
8: *localRnicInfo.ip* ⇐ *local IB device bounded IP*
9: **else**
10: *local_support* ⇐ *False*
11: **end if**
12: tcpSyncData(*local_support*, *remote_support*, *localRnicIn-fo*, *remoteRnicInfo*)
13: **if** *local_support* and *remote_support* are *True* **then**
14: *isRdmaSupport* ⇐ *True*
15: **return** *True*
16: **else**
17: *isRdmaSupport* ⇐ *Flase*
18: **return** *False*
19: **end if**

Algorithm 2 Load-Aware Buffer Registration.

Input:
float baseBuffer
float overloadThr
float k
Output:
float bufferSize

1: *memUsage* ⇐ *current memory usage in host*
2: *netUsage* ⇐ *current network throughput in host*
3: calculate *freeMemRate* and *freeNetRate*
4: *loadFactor* ⇐ *freeMemRate* * *freeNetRate*
5: **if** *loadFactor* in range (*0, overloadThr*) **then**
6: *bufferSize* ⇐ *baseBuffer* * *loadFactor* * *k*
7: **else**
8: *bufferSize* ⇐ *baseBuffer*
9: **end if**

baseline value as buffer size. We determine the baseline *buffer size S* based on experience [20]. Formula (1) shows the core idea for the above algorithm, where *constant k* is regular factor.

$$M = k \times S \times \left(1 - \frac{THR}{B}\right) \times \left(1 - \frac{U}{T}\right) \tag{1}$$

Where M refers to calculated memory buffer size for RDMA, regular factor $k \in (0, 1)$, S is the baseline buffer size determined by oplog size RDMA_Mongo, THR is current network throughput, B is the maximum bandwidth limit of RNIC, U is current busy memory size, and T is total DRAM size. Oplog size can be specified in RDMA_Mongo configuration file. The actual values of k, S, B and T used in our experiment are respectively 0.7, 50MB, 100Gbps and 32GB.

4.3. RDMA write with reliable connection

When referring to RDMA one-sided communication primitives, the coordination mechanisms between sender and receiver endpoints are a significant challenge. We take RDMA-based CURD request/response handling between RDMA_Mongo client and mongos component as an example.

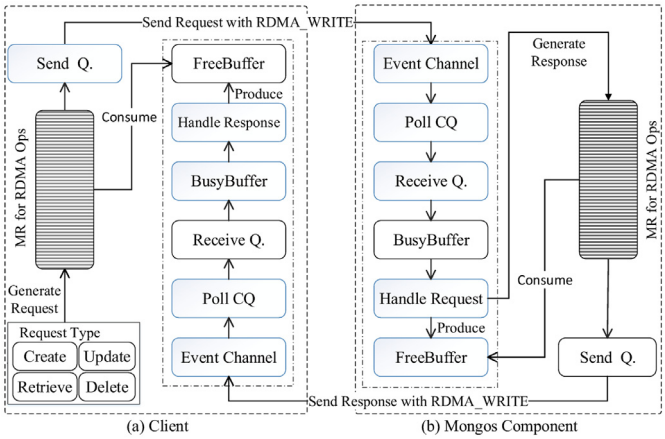

Fig. 6. Coordination mechanism between Client and Mongos Component in RDMA_Mongo.

The message can be divided into two categories: control & data message. Control messages with constant size (less than 4KB) are typically exchanged on small registered buffers, while data messages with highly variable size normally require large registered memory buffer. Different requests/responses correspond to different message types. For CURD requests sent by RDMA_Mongo client as control message, small registered MR (Message Region) on mongos component are required as receive buffers. For Create/Update/Delete response issued by mongos component, small receive buffers on client are required. Whereas, for query responses issued by mongos component as data message, large registered receive buffers on the client are needed and corresponding memory address/rkey(remote key) need to be contained in query request in advance. Each side between clients and mongos contains one small pre-registered send/receive buffer for sending/receiving control message. Considering send/receive buffer contention, RDMA_Mongo should identify when send/receive buffer are idle to be consumed or reused, and when send/receive buffer are occupied.

Fig. 6 shows the detailed description of RDMA-enhanced CURD request/response handling mechanism between client and mongos with one-sided RDMA Write in RC (Reliable Connection) service. The registered buffer is marked as idle or busy by the coordination mechanism, which responses to the circular queues *freeBuffer* or *busyBuffer*. The entries in *freeBuffer*/*busyBuffer* respectively correspond to idle/busy registered buffers maintained by RDMA_Mongo client/mongos components. Before issuing CURD requests, RDMA_Mongo client gains one idle pre-registered buffer from *freeBuffer* queue and encapsulates related *address/rkey* into the CURD request message. The client uses RDMA Write with immediate data to write the CURD request to remote pre-registered control message buffer on mongos. Meanwhile, when the completion event channel in mongos detects a new completed task, an event is triggered to notify mongos to poll completion queue (CQ).

After that, mongos reads CURD request from *busyBuffer*, extract *peer address/rkey*, handles the request, frees busy receive buffer to idle state and generates response. Then the response message is written to peer address with rkey provided by RDMA-_Mongo client. The process of receiving responses in client is similar to that of receiving requests in mongos component.

4.4. RDMA-driven oplogs synchronization

To achieve high availability and redundancy, MongoDB introduces replica set mechanism to maintain the same data set, which contains one primary node and several secondary nodes. Rather than directly pulling data collections from the primary node, the secondary nodes fetch remote operation logs (oplog) and replay these oplogs to achieve data persistence, as shown in Fig. 7. The oplogs synchronization consists of two phases which are initial sync and steady state replication. Initial sync is time-consuming full-scale synchronization while steady state replication is faster and more frequent incremental fetch. We mainly focus on the optimization and redesign of the communication protocols in these two phases with RDMA primitives. There are three scenarios that can trigger initial sync, including when new node has just joined, when the last initial sync failed, and when *resync* command is triggered.

Once the target sync source node is determined, the secondary node will start an *OplogFetcher* thread to continuously poll the remote latest oplog timestamp T_R, compare T_R with local latest oplog timestamp T_L. The Boolean variable *_initialSyncFlag* identifies the initial sync phase. If *_initialSyncFlag* is true, *Oplog-Fetcher* fetches the remote oplogs until consistent point T_L named *minValid*. If T_L is less than T_R, *OplogFetcher* fetches the remote oplogs in timestamp interval (T_L, T_R), which means *minValid* is T_R. Then *OplogApplier* thread is responsible for replaying the fetched oplogs into local durable. The primary node periodically sends heartbeat packets to the secondary nodes to obtain the lasted applied/durable oplogs timestamp and update local topology state.

For offloading the above transmission task into RDMA NIC thoroughly, we leverage kernel-bypassing RDMA Write verbs with immediate data to send messages and oplogs. Fig. 8 specifically depicts RDMA-driven oplogs synchronization. RD-

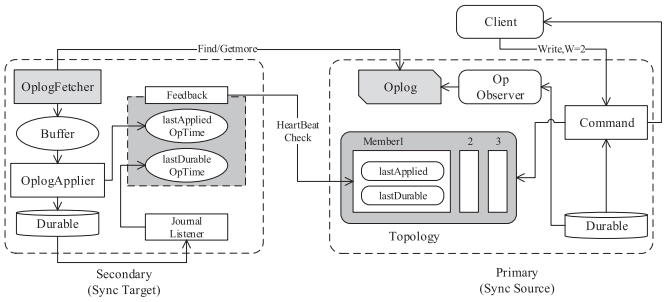

Fig. 7. MongoDB Oplogs Sync Diagram.

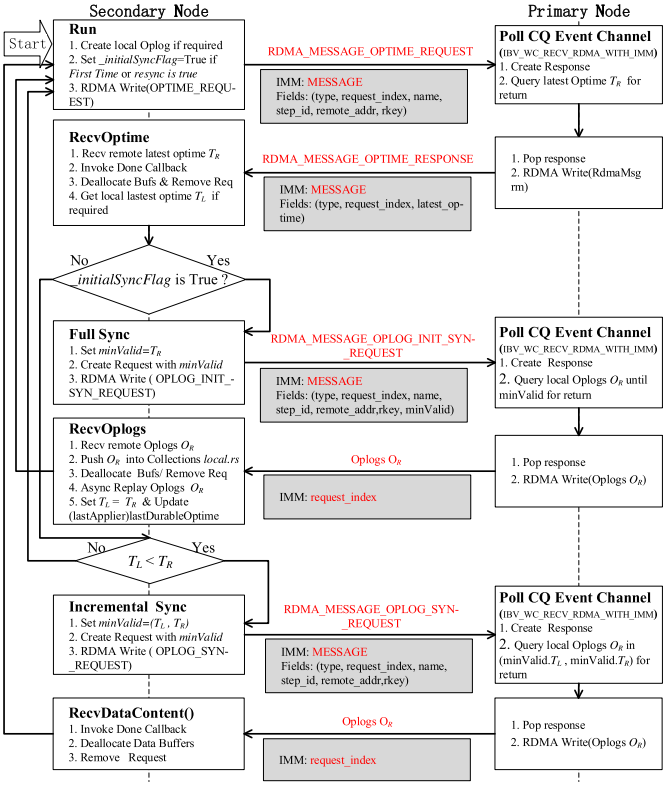

Fig. 8. RDMA-Driven Oplogs Sync in MongoDB.

Table 2
RDMA Message structure.

Field	Size	Field	Size
type	1B	rkey	4B
name_size	2B	data_type	XB
name	512	oplogs_shape	XB
step_id	8B	oplogs_bytes	8B
request_index	8B	min_valid	8B
remote_addr	8B	error_status	size = 4B
/checksum			proto = XB

MA CQ (Completion Queue, CQ) event channel is exploited to listen on incoming RDMA message requests asynchronously. To distinguish different request/response and be better compatible with RDMA-enabled communication, we define one RdmaMessage structure, as shown in Table 2. Algorithm 3 exhibits RDMA-enabled data sync in secondary node and Algorithm 4 shows RDMA-enabled oplog observing in primary node.

Algorithm 3 RDMA-Enabled Secondary Oplogs Sync.

Input: *bool* resync, *int* timeout
resync: bool flag for restarting initial sync
rn: remote target source node for oplogs sync
opq: oplogs blocking queue for caching fetched oplogs
timeout: sleep time for starting the next oplogs sync
wc: pre-allocated work completions array used for polling
MAX_CQ_NUM: max number of CQs
Output: *None*

```
1: _initialSyncFlag ⇐ bool flag for starting initial sync
2: _pause ⇐ bool flag for pause sync process
3: _shutdown ⇐ bool flag for shut down sync process
4: if T_L is Null or resync is True then
5:    _initialSyncFlag ⇐ True
6: end if
7: while _shudown is False do
8:    if _pause is True then
9:       break
10:   end if
11:   T_R ⇐ requestRdmaWriteRemoteOptime(rn)
12:   if _initialSyncFlag is True then
13:      ops ⇐ requestRdmaWriteSyncOplogs(rn, T_R)
14:   else
15:      if T_L < T_R then
16:         ops ⇐ requestRdmaWriteSyncOplogs(rn, T_L, T_R)
17:      else
18:         sleep(timeout)
19:         continue
20:      end if
21:   end if
22:   enqueueOpQueue(opq, ops)
23:   asyncReplayOplogsFromQueue()
24:   T_L ⇐ T_R
25:   updateLastApplierOptime()
26:   updateLastDurableOptime()
27: end while
```

5. Evaluation

In this section, we evaluate RDMA_Mongo's throughput, latency and consuming time over RoCE NICs from two dimensions: single-threaded performance Section 5.2 and muti-thread Section 5.3 performance. In each dimension, we evaluate the performance among four operations: insert, delete, update and query. In Section 5.1, we describe our experiment setup. In Section 5.2, we make a comparative performance analysis of RDMA_Mongo using a single thread. In Section 5.3, we evaluate the consuming time under multi-threaded processing.

5.1. Experiment setup

The performance experiment was conducted on a local cluster with three servers, inter-connected by a Mallanox SN2100 switch, as shown in Fig. 9. Our testbed runs RoCE (RDMA over Converged Ethernet). Each server has a 32GB DRAM and two 12-core Intel Xeon E5-2687W v4 CPU processors with 3.00 GHz of frequency. Each CPU core is equipped with a private 768KB L1 cache and a private 3072KB L2 cache. A 30MB L3 cache is shared by 10 cores on one CPU processor. Each server is equipped with a onnectX-5 MCX516A-CDAT 100Gbps RoCE NIC via PCIe 3.0 x16 connected to one Mallanox SN2100 100Gbps RoCE Switch. CentOS 7.4.1708 with OFED 4.5-1.0.1 stack is run on each server. RDMA_Mongo prototype is based on MongoDB 4.1.1-59-g1dd056d.

For performance comparison between RDMA_Mongo and plain MongoDB, we concentrate on three important performance metrics including *throughput, latency* and *consuming time. Throu-ghput* is defined as how many CURD operations per second RDMA-_Mongo/Mongo processed, as shown in Formula (2). *Latency* is defined as the average delay for each CURD operation, as shown in Formula (3). *Consuming time* indicates how long it takes to process a set of CURD operation. The

Algorithm 4 RDMA-Enabled Oplog Observing.

Input:
cq: RDMA completion queue
ec: RDMA completion event channel
ct: RDMA context entity
Output: *None*

```
1: _isRun ⇐ True
2: _cc ⇐ CQ context
3: _cq ⇐ extracted CQ from CQ event channel
4: while _isRun is True do
5:    ibvGetCqEvent(ec, _cq, _cc)
6:    if cq != _cq then
7:       continue
8:    end if
9:    ibvAckCqEvents(_cq, 1)
10:   ibvReqNotifyCq(_cq)
11:   ne ⇐ ibvPollCq(cq, MAX_CQ_NUM * 2, wc)
12:   if ne == 0 then
13:      continue
14:   end if
15:   for all WorkCompletion, wc_i ∈ wc do
16:      if wc_i.status is not IBV_WC_SUCCESS then
17:         continue
18:      end if
19:      if wc_i.opcode is IBV_WC_RECV_RDMA_WITH_IMM then
20:         imm ⇐ wc_i.imm
21:         doRecvImmCallback(imm, wc_i)
22:         if imm is OPTIME_REQUEST then
23:            T_R ⇐ queryLastOptime()
24:            rdmaWriteResponse(T_R)
25:            continue
26:         else if imm is OPLOG_SYN_REQUEST then
27:            O_R ⇐ queryOplogs(wc_i)
28:            rdmaWriteOplogs(O_R)
29:         end if
30:         continue
31:      else if wc_i.opcode is IBV_WC_RDMA_WRITE then
32:         doRdmaWriteCallback(wc_i)
33:      end if
34:   end for
35: end while
```

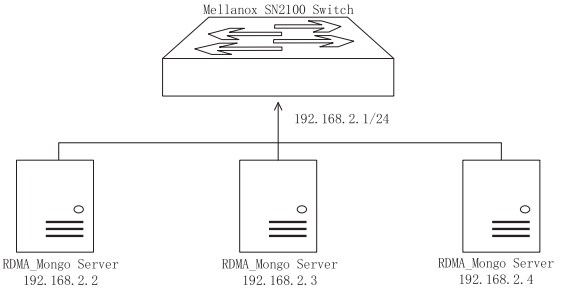

Fig. 9. Network Diagram and Topology.

number of operation records is the independent variable in the experiment. The operation types include insert, delete, update and query for each data record. To reflect large-scale data processing, each run executes CURD operations with varying sizes of data records counts ranging from 0.1K to 1000K. For comprehensize comparison, we measure the consuming time on single-threaded and multi-threaded CURD operations. Each experimental result is the average of 20 runs.

$$P = \frac{op}{T} \tag{2}$$

Where P refers to throughput, op refers to operation count and T refers to total delay.

$$L = \frac{T}{op} \tag{3}$$

Where L refers to average latency per CURD operation, op refers to operation count and T refers to total delay.

5.2. Evaluation with basic performance

In this section, we evaluate the performance of RDMA-_Mongo and plain MongoDB with single-threaded CURD operations. In each run, CURD operations are executed with the increase of data records counts (0.01K–1000K) to compare the performance with RDMA_Mongo and plain MongoDB. After all performance values are collected, the relationship between independent variables (operation and records counts) and performance metrics (throughput /latency /consuming time) are described in Fig. 10.

Based on performance results, we found that RDMA-enhanc-ed paradigm can significantly boost the CURD performance of document-based MongoDB, with improved insert, delete, update and query performance by 30%, 30%, 17% and 15%, respectively.

5.2.1. Insert performance

We measure the overall consuming time of inserting 0.1K, 1K, 10K, 100K, and 1000K data records in one single thread, respectively. Each operation inserts one data record. For each data records size, we measure the consuming time 20 times. After that, we caculate the throughput and latency based on Formula (2) and (3).

Fig. 10 (a) shows the throughput, latency and overall consuming time for insert operations. In terms of throughput, we can see that RDMA_Mongo has higher throughput than native MongoDB with *max throughput gap* of *1194 ops/s* and *min throughput gap* of *804 ops/s*. To be specific, the average throughput of insert operation of RDMA_Mongo is *29.72%* better than native MongoDB. Meanwhile, RDMA_Mongo has significantly lower latency than Mongo with *max latency gap* of *0.213 ms/ops* and *min latency gap* of *0.067 ms/ops*. When it comes to overall consuming time, RDMA_Mongo with one-sided RDMA WRITE achieves much lower consuming time than Mongo with TCP/IP. Moreover, the consuming time of 0.1K operation records is *21.3 ms* while that of 1000K operation records is *67868.7 ms*. We find that the gap of overall consuming time becomes significantly larger once the number of operation records $<=$*10K*. This advantage is from fewer memory copies and CPU context switches in RDMA_Mongo with one-sided RDMA WR-ITE.

5.2.2. Delete performance

Like the experiment setting of the insert operations, we measure the overall consuming time of deleting 0.01K, 0.1K, 1K, 10K, and 100K data records in a single thread, respectively. Each operation deletes one data record with index field. For each data records size, we measure the consuming time 20 times. Afterwards, we caculate the throughput and latency according to Formula (2) and (3).

The throughput, latency and overall consuming time for delete operations are shown by Fig. 10 (b). Compared to plain MongoDB, RDMA_Mongo achieves higher throughput, lower latency and overall consuming time. For throughput, RDMA_Mongo is *37.03%*$\sim$*1.09x* better than plain MongoDB with *max throughput gap* of *1276 ops/s* and *min throughput gap* of *702 ops/s*. For latency, the max gap is *0.265ms* while the min gap is *0.068ms*. As one-sided RDMA WRITE is leveraged by RDMA_Mongo with zore copy and fewer CPU involvement, with the increasing delete records, the gap of overall consuming time is significantly larger.

5.2.3. Update performance

The experiment settings of update operations are like that of insert and delete operations. When updating a data record, new data record needs to be compared to old record. If the data is inconsistent, the update operation is performed. Fig. 10(c) shows the comparison of throughput, latency and overall consuming time under varying size of update operation records between RDMA_Mongo and plain MongoDB. We can see that for *small update records counts* ($\leq$*1000*), RDMA_Mongo has similar performance (throughput /latency /consuming time) to plain MongoDB as the comparison overhead between old and new data records is dominate factor. However, for *large-scale update records counts* ($\geq$*1000*), the overheads of memory copies and CPU context switches for data transfer is dominate factor, and RDMA_Mongo achieves better throughput (10%$\sim$24.95% higher) than plain MongoDB.

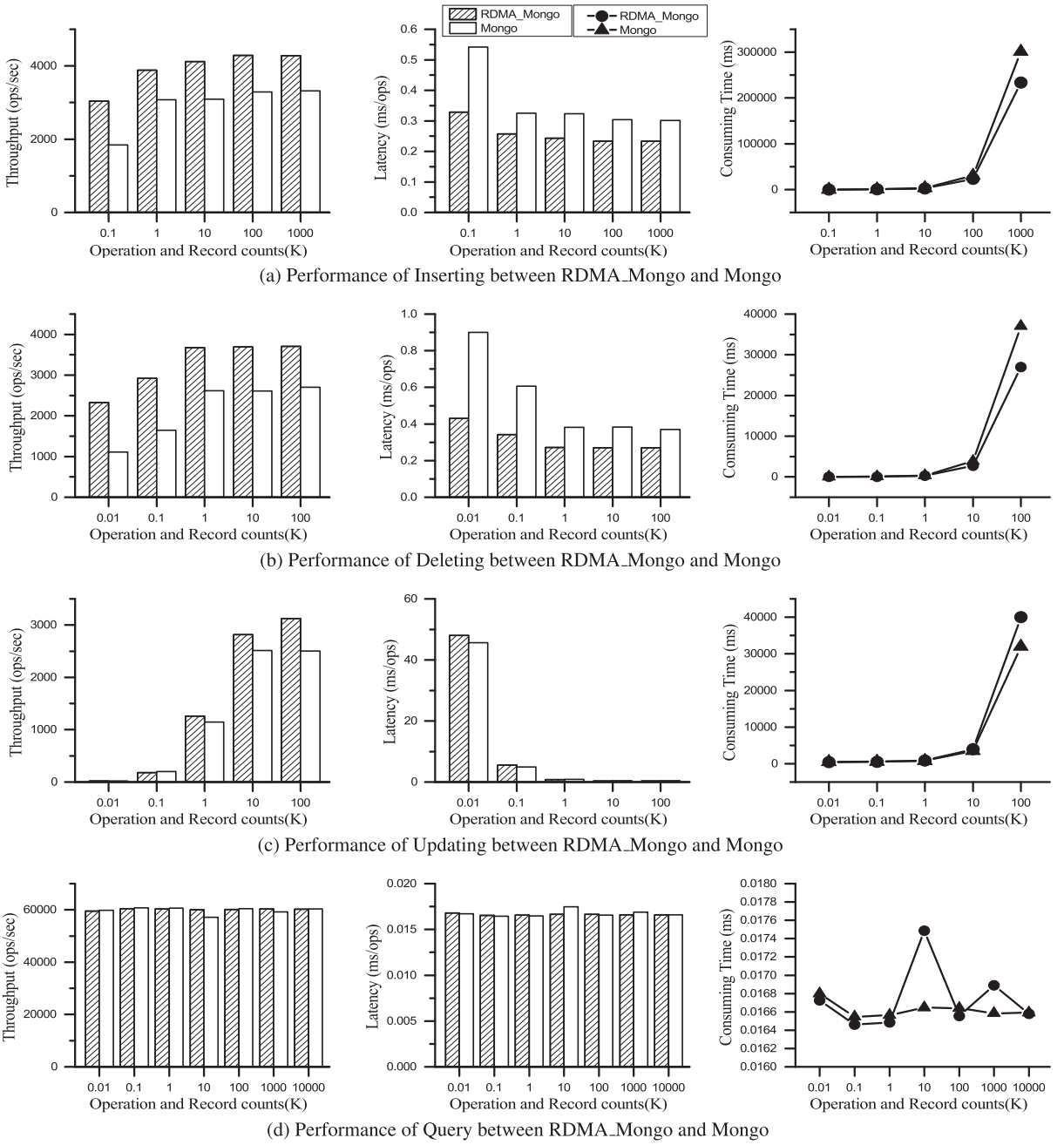

Fig. 10. Throughput, latency, consuming time comparison between RDMA_Mongo and Mongo with different operations.

5.2.4. Query performance

We measure the overall consuming time of querying 10, 100, 1K, 10K, 100K, 1000K and 10000K data records in a single thread, respectively. Each operation queries multiple data record. For each data records size, we measure the consuming time 20 times. After that, the throughput and latency are caculated based on Formula (2) and (3).

Fig. 10 (d) reveals the trends in throughput, latency and consuming time of query operation as data scale grows. We can see that when the queried data size is small *(≤ 1000)*, the performance of both MongoDB and RDMA_Mongo is similar. However, if the amount of data queried becomes large *(> 1000)*, the improved performance is powerful. The larger the amount of query data queried, the better performance is improved. Moreover, with the increasing data size, the query performance of Mongo becomes instable, while RDMA_Mongo is more stable.

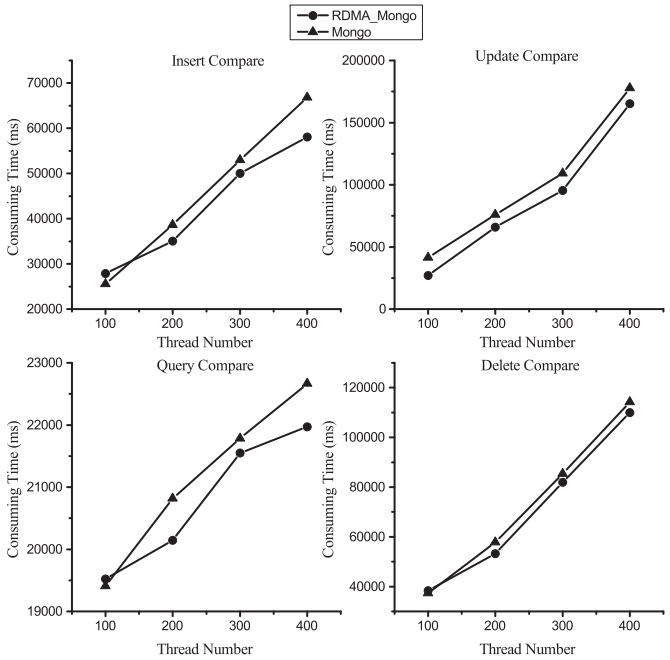

Fig. 11. The consuming time of RDMA_Mongo and Mongo in multi-threaded case, including the operation type of insert, update, query and delete.

5.3. Evaluation with multi-threaded performance

To verify the performance benefit of RDMA_Mongo under high concurrency and large-scale requests, multiple CURD operations are executed simultaneously in varying number of threads ranging from *100* to *400*. Each thread represents an established connection and executes *1000* CURD operation records. Assuming that the number of threads is *N*, each run starts N threads simultaneously and there are *N*1000* operation data records. After the results of overall consuming time of CURD operations are collected, the relationships between thread number and consuming time for different operation types are described in Fig. 11.

According to performance results in Fig. 11, we can find that the trend lines of overall consuming time for highly concurrent CURD operations in RDMA_Mongo are always below that of plain MongoDB. Specifically, as concurrent CURD requests grow, RDMA_Mongo with RDMA channel always achieves lower consuming time compared to plain Mongo with TCP channel. For highly concurrent insert and query performance, the greater the number of threads, the larger the gap of overall consuming time. When the number of threads is 400, the consuming time gap of insert and query achieves the maximum by up to 15%. Moreover, for large-scale data request and processing, frequent memory copying and CPU context switching are the dominate overheads. These advantages are due to the fact that RDMA channels require less memory copies and CPU involvement by bypassing OS kernel than TCP channels, and the saved CPU can be used for resource scheduling and allocation. By comparing the consuming time, we can conclude that the improved performance by RDMA_Mongo in multi-threaded case is similar to that in single-threaded case.

6. Related work

Modern Network Hardware: There are three types of RD-MA networks: Infiniband [34], RoCE, and iWARP [37]. Infiniband is a network designed for RDMA, which guarantees reliable transmission from the hardware level. RoCE and iWARP are both Ethernet-based RDMA technology and they support the corresponding verbs interface [3]. Specially, the RoCE protocol has two versions, RoCEv1 and RoCEv2 [48]. The main difference between the two versions is that RoCEv1 is based on the RDMA protocol implemented by the Ethernet link layer, while RoCEv2 is the UDP layer implementation in the Ethernet TCP/IP protocol. The investigation in [10,30] shows that Infiniband has better performance compared with RoCE and iWARP. However, the network cards and switches for Infiniband are more expensive. By contrast, RoCE and iWARP have relatively low cost due to the only requirement of dedicated NICs. With similar specification, iWARP is more expensive and complex than RoCE [30]. Moreover, iWARP has 4X lower throughput and 3X higher latency than RoCE [30]. Therefore, RoCE enabled network is adopted in this research.

Kernel-Bypass Networking: Kernel-bypass networking eliminates the overhead of OS network stack and data copies between application buffers and OS memory buffers by moving packets processing up to userspace or offloading network stack into dedicated NIC. The mainstream technologies of kernel-bypass networking include DPDK [23] and RDMA. TCP/IP

network stack is moved up to userspace in DPDK. Userspace I/O (UIO) is exploited by DPDK to map NIC device memory into userspace. Huge-pages is leveraged by DPDK to reduce TLB cache miss. Meanwhile, instead of interrupting the CPU, DPDK uses polling to access the packets in the NIC. However, userspace network stack in DPDK still involves CPU intervention. The concurrency of DPDK depends heavily on the number of CPU cores. DPDK causes unnecessary CPU idling in low-load scenarios. For RDMA, network stack is offloaded into RNIC devices without any CPU involvement. High bandwidth and zero-copy mechanism in RDMA promise high throughput and ultra-low latency. The rate of receiving packets in RDMA is completely determined by the forwarding capability of RNICs. So the prototype in this paper is executed in RDMA-enabled network.

RDMA Protocol Stack Optimization: Some research works [9,21,30] focus on the protocol optimization over RDMA recently. Unlike previous end-to-end single-path transport for RDMA, MP-RDMA in [21] presents a multi-path RDMA transport, which takes full advantage of the rich transmission paths in datacenters. IRN in [30] proposes a more efficient loss recovery and end-to-end flow control bounded with the number of in-flight packets, which eliminates the requirement of Priority Flow Control (PFC) and outperforms RoCE with PFC. The research in [9] demonstrates a unified design of unreliable send/recv and RDMA Write over connectionless datagrams, which achieves higher performance and scalability than connection-oriented RDMA transports. The above researches mainly involve the underlying mechanism of RDMA, but lack attention to resource management for RDMA-enabled transactions. RDM-A_Mongo proposes a novel load-aware buffer registration algorithm for efficient memory management. Note that RoCE with PFC is required by RDMA_Mongo.

RDMA-Driven NoSQL Database: Due to stronger demand for the stores and analysis of massive semi-structured/unstructured data, NoSQL databases are expected to provide higher throughput, lower latency and CPU overhead. RDMA-driven NoSQL databases, especially key-value [4,7,14–16,28,38,43,44,49] and graph stores [39], are becoming prevalent in datacenters.

Pilaf in [28] leverages one-sided RDMA Read to process get requests with optimal CPU overhead and proposes self-verifying data structures to eliminate read-write races between clients and server. In FaRM [7], the memory in the cluster is exposed as a transparently shared address space. Lock-free reads over RDMA, function shipping and collocating objects are supported by FaRM to achieve more efficient transactions for key-value store or other applications. HydraDB [44] demonstrates a novel RDMA-based key-value middleware for general purpose and boosts high-performance in-memory key-value store from multiple aspects including RDMA Write enabled message passing, RDMA Read enabled GET operations, remote pointer sharing and a lightweight consistency mechanisms. Moreover, multicore systems are leveraged efficiently by HydraDB to fully take the potential of RDMA. Unlike RDMA Read-enabled key-value systems like FaRM [7] and Pilaf [28], HERD [16] uses a mix of one-sided RDMA Write and two-sided Send/Recv verbs to execute GET/PUT operations, which reduces network round trips and achieves 2X higher throughput than prior FaRM and Pilaf. In Wukong [39], a distributed RDMA-enabled RDF graph stores is proposed to achieve lower latency, higher throughput and concurrency for RDF queries over large RDF datasets. Specifically, RDMA Read is used for small RDF queries while RDMA Write is used for large non-selective queries. However, to the best of our knowledge, there is still no attempt to accelerate document-based NoSQL systems with RDMA. So we focus on one-sided RDMA-enhanced document NoSQL databases with the aims of high throughput, low latency and CPU overhead.

Recently, kernel-bypass networking is an emerging field to accommodate the need of data intensive computing. Despite of some research achievements, a number of open issues and challenges in RDMA-driven system design and optimization are expected to be tackled. The research in this paper revolves around RDMA-driven optimization of document-based NoSQL database.

7. Conclusion

Inspired from RDMA-enhanced paradigm, we propose the first RDMA-enabled document-based NoSQL paradigm named RDMA_Mongo. It can efficiently exploit the feature of OS kernel-bypassing and zero copy in PFC-enabled RDMA network to mitigate the traditional network transmission challenges: frequent CPU involvement and long waiting time for MongoDB client with high concurrency. In this paper, we particularly analyzed the MongoDB network transport layer. Further, we demonstrated a rich design space and the tradeoff among three design choices. RDMA_Mongo employs a novel RDMA context detection algorithm and load-aware buffer registration algorithm to coexist with traditional network stack and register RDMA communication area in a load-ware manner. To accelerate the oplogs synchronization between the primary and secondary node, RDMA write with immediate data is leveraged to send ACK, control message or oplogs while RDMA completion event channel is used to receive messages or oplogs asynchronously. By taking the CURD requests between MongoDB client and mongos as an example, RDMA_Mongo implements a coordination mechanism using more efficient one-sided RDMA Read/Write primitives with Reliable Connection service. Experiment results on both baseline operations and high concurrent number of connections show that the performance of RDMA_Mongo has been greatly improved. Looking ahead, we are committed to making it possible for MongoDB in RDMA enabled network, with better RDMA based sharding strategies, transaction mechanisms and NVMM (Non-Volatile Main Memory) based storage.

Disclosure of conflicts of interest

None.

Acknowledgements

We sincerely thank the anonymous reviewers for their insightful comments. We would like to thank Pengzhi Zhu and Qingchun Song, members of HPC-AI International Advisory Committee, for their technology support. We would like to thank Gaofeng Feng and Gil Blooh, researchers in Mellanox, for their helpful advices. The work of this paper is supported by National Natural Science Foundation of China under Grant (No.61873309, No.61572137, and No. 61728202), and Shanghai 2018 Innovation Action Plan project under Grant No.18510760200-Research on smart city big data processing technology based on cloud-fog mixed mode, and Grant No.18510732000-Development and demonstration application of intelligent logistics management decision system based on big data and artificial intelligence technology, and China National Key R&D Program 2017 Project under Grant No. 2017YFC0910101-Building a safe and reliable guarantee system for biomedical big data, cross-system samples and data sharing, and 2017 Research Projects of Shanghai Science and Technology Commission under Grant No. 17DZ1101000-Research of data circulation market system construction, and Project of Shanghai Municipal Commission of Economy and Informatization under Grant No. 2018-01032-Construction and application of financial regulation sandbox platform.

References

[1] D. Abadi, S. Madden, M. Ferreira, Integrating compression and execution in column-oriented database systems, in: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM, 2006, pp. 671–682.
[2] S. Akiyama, K. Taura, Uni-address threads: scalable thread management for RDMA-based work stealing, in: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, ACM, 2015, pp. 15–26.
[3] M. Beck, M. Kagan, Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure, in: Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching, International Teletraffic Congress, 2011, pp. 9–15.
[4] B. Cassell, T. Szepesi, B. Wong, T. Brecht, J. Ma, X. Liu, Nessie: a decoupled, client-driven key-value store using RDMA, IEEE Trans. Parallel Distrib. Syst. 28 (12) (2017) 3537–3552.
[5] Z. Wu, Z. Lu, P.C.K. Hung, S. Huang, Y. Tong, Z. Wang, QaMeC: A QoS-driven IoVs application optimizing deployment scheme in multimedia edge clouds, Future Gener. Comp. Syst. 92 (2019) 17–28.
[6] Y. Chen, X. Wei, J. Shi, R. Chen, H. Chen, Fast and general distributed transactions using RDMA and HTM, in: Proceedings of the Eleventh European Conference on Computer Systems, ACM, 2016, p. 26.
[7] A. Dragojević, D. Narayanan, M. Castro, O. Hodson, FaRM: fast remote memory, in: 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), 2014, pp. 401–414.
[8] L. George, HBase: The Definitive Guide: Random Access to Your Planet-Size Data, O'Reilly Media, Inc., 2011.
[9] R.E. Grant, M.J. Rashti, P. Balaji, A. Afsahi, Scalable connectionless RDMA over unreliable datagrams, Parallel Comput. 48 (2015) 15–39.
[10] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, M. Lipshteyn, RDMA over commodity ethernet at scale, in: Proceedings of the 2016 ACM SIGCOMM Conference, ACM, 2016, pp. 202–215.
[11] J. Han, E. Haihong, G. Le, J. Du, Survey on NoSQL database, in: 2011 6th international conference on pervasive computing and applications, IEEE, 2011, pp. 363–366.
[12] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, D.K. Panda, High-performance design of HBase with RDMA over Infiniband, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE, 2012, pp. 774–785.
[13] N.S. Islam, M. Wasi-ur Rahman, X. Lu, D.K. Panda, High performance design for HDFS with byte-addressability of NVM and RDMA, in: Proceedings of the 2016 International Conference on Supercomputing, ACM, 2016, p. 8.
[14] N.S. Islam, D. Shankar, X. Lu, M. Wasi-Ur-Rahman, D.K. Panda, Accelerating I/O performance of big data analytics on HPC clusters through RDMA-based key-value store, in: 2015 44th International Conference on Parallel Processing, IEEE, 2015, pp. 280–289.
[15] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N.S. Islam, X. Ouyang, H. Wang, S. Sur, et al., Memcached design on high performance RDMA capable interconnects, in: 2011 International Conference on Parallel Processing, IEEE, 2011, pp. 743–752.
[16] A. Kalia, M. Kaminsky, D.G. Andersen, Using RDMA efficiently for key-value services, ACM SIGCOMM Comput. Commun. Rev. 44 (4) (2015) 295–306.
[17] A. Kalia, M. Kaminsky, D.G. Andersen, FaSST: fast, scalable and simple distributed transactions with two-sided ({RDMA}) datagram RPCs, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016, pp. 185–201.
[18] F. Li, S. Das, M. Syamala, V.R. Narasayya, Accelerating relational databases by leveraging remote memory and RDMA, in: Proceedings of the 2016 International Conference on Management of Data, ACM, 2016, pp. 355–370.
[19] F. Liu, L. Yin, 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, ACM, 2017, pp. 48–63.
[20] X. Lu, M.W.U. Rahman, N. Islam, D. Shankar, D.K. Panda, Accelerating spark with RDMA for big data processing: early experiences, in: 2014 IEEE 22nd Annual Symposium on High-Performance Interconnects, IEEE, 2014, pp. 9–16.
[21] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, T. Moscibroda, Multi-path transport for {RDMA} in datacenters, in: 15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18), 2018, pp. 357–371.
[22] Y. Lu, J. Shu, Y. Chen, T. Li, Octopus: an RDMA-enabled distributed persistent memory file system, in: 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17), 2017, pp. 773–785.
[23] P. MacArthur, Userspace RDMA verbs on commodity hardware using DPDK, in: 2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI), IEEE, 2017, pp. 103–110.
[24] I. MongoDB, Top 5 considerations when evaluating nosql databases, White Paper, 2015, (https://www.ascent.tech/wp-content/uploads/documents/mongodb/10gen-top-5-nosql-considerations-february-2015.pdf).
[25] I. Mellanox Technologies, InfiniBand in the Enterprise Data Center, 2006, (http://www.mellanox.com/pdf/whitepapers/InfiniBand_EDS.pdf).
[26] I. Mellanox Technologies, RDMA Aware Programming User Manual, 2015, (http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf).
[27] Mellanox Technologies I, Mellanox OFED for Linux User Manual, 2018, (http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v4_3.pdf).
[28] C. Mitchell, Y. Geng, J. Li, Using one-sided {RDMA} reads to build a fast, CPU-efficient key-value store, in: Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13), 2013, pp. 103–114.
[29] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, J. Li, Balancing {CPU} and network in the cell distributed b-tree store, in: 2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16), 2016, pp. 451–464.
[30] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, S. Shenker, Revisiting network support for RDMA, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, ACM, 2018, pp. 313–326.
[31] I. MongoDB, Mongodb, 2009, (https://www.mongodb.com/).
[32] I. MongoDB, NoSQL Databases Explained, 2018, (https://www.mongodb.com/nosql-explained).

[33] S. Peng, G. Wang, Y. Zhou, C. Wan, C. Wang, S. Yu, An immunization framework for social networks through big data based influence modeling, IEEE Trans. Dependable Secure Comput. (2017).
[34] G.F. Pfister, An introduction to the infiniband architecture, High Perform. Mass Storage Parallel I/O 42 (2001) 617–632.
[35] M. Poke, T. Hoefler, Dare: high-performance state machine replication on RDMA networks, in: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, ACM, 2015, pp. 107–118.
[36] X. Chen, S. Tang, Z. Lu, J. Wu, Y. Duan, S. Huang, Q. Tang, iDiSC: A New Approach to IoT-Data-Intensive service components deployment in Edge–Cloud-Hybrid system, in: IEEE Access, 7, IEEE, 2019, pp. 59172–59184.
[37] M.J. Rashti, A. Afsahi, 10-Gigabit iWARP ethernet: comparative performance analysis with infiniband and Myrinet-10G, in: 2007 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2007, pp. 1–8.
[38] D. Shankar, X. Lu, N. Islam, M. Wasi-Ur-Rahman, D.K. Panda, High-performance hybrid key-value store on modern clusters with RDMA interconnects and SSDs: non-blocking extensions, designs, and benefits, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 393–402.
[39] J. Shi, Y. Yao, R. Chen, H. Chen, F. Li, Fast and concurrent {RDF} queries with RDMA-based distributed graph exploration, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016, pp. 317–332.
[40] M. Su, M. Zhang, K. Chen, Z. Guo, Y. Wu, RFP: when RPC is faster than server-bypass with RDMA, in: Proceedings of the Twelfth European Conference on Computer Systems, ACM, 2017, pp. 1–15.
[41] S.-Y. Tsai, Y. Zhang, Lite kernel RDMA support for datacenter applications, in: Proceedings of the 26th Symposium on Operating Systems Principles, ACM, 2017, pp. 306–324.
[42] C. Wang, J. Jiang, X. Chen, N. Yi, H. Cui, APUS: fast and scalable Paxos on RDMA, in: Proceedings of the 2017 Symposium on Cloud Computing, ACM, 2017, pp. 94–107.
[43] Y. Wang, X. Meng, L. Zhang, 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, ACM, 2014, pp. 1–13.
[44] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, S. Meng, HydraDB: a resilient RDMA-driven key-value middleware for in-memory cluster computing, in: SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2015, pp. 1–11.
[45] J. Webber, A programmatic introduction to Neo4j, in: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, ACM, 2012, pp. 217–218.
[46] X. Wei, Z. Dong, R. Chen, H. Chen, Deconstructing RDMA-enabled distributed transactions: hybrid is better!, in: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), 2018, pp. 233–251.
[47] X. Wei, J. Shi, Y. Chen, R. Chen, H. Chen, Fast in-memory transaction processing using RDMA and HTM, in: Proceedings of the 25th Symposium on Operating Systems Principles, ACM, 2015, pp. 87–104.
[48] I. Wikimedia Foundation, RDMA over Converged Ethernet, 2018, (https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet).
[49] M. Yang, S. Yu, R. Yu, N. Xiao, F. Liu, W. Chen, InnerCache: a tactful cache mechanism for RDMA-based key-value store, in: 2016 IEEE International Conference on Web Services (ICWS), IEEE, 2016, pp. 646–649.
[50] S. Yu, M. Liu, W. Dou, X. Liu, S. Zhou, Networking for big data: A survey, IEEE Commun. Surv. Tutorials 19 (1) (2017) 531–549.