Characterizing and Accelerating Indexing Techniques on Distributed Ordered Tables

Shashank Gugnani*, Xiaoyi Lu*, Houliang Qi†‡, Li Zha†‡, and Dhabaleswar K. (DK) Panda*
* Department of Computer Science and Engineering, The Ohio State University
Email: {gugnani.2, lu.932, panda.2}@osu.edu
† Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
Email: qihouliang@software.ict.ac.cn, char@ict.ac.cn
‡ University of the Chinese Academy of Sciences, Beijing, China

***Abstract*—In recent years, most Web 2.0/3.0 applications have been built on top of distributed systems which allow data to be modeled as Distributed Ordered Tables (DOTs) such as Apache HBase. To analyze the stored data, SQL-like range queries over a DOT are fundamental requirements. However, range queries over existing DOT implementations are highly inefficient. Several secondary index techniques have been proposed to alleviate this issue, but they introduce additional overhead while creating and updating the index. Moreover, index techniques introduce several additional challenges for DOTs, particularly, network communication and thread models for concurrent request processing. In this paper, we first characterize the performance of index techniques on DOTs from a networking perspective. We then propose an RDMA-based high-performance communication framework which uses HBase as the underlying DOT implementation to accelerate these techniques. We propose several thread models for our RDMA-based design and compare their performance. We design a parallel insert operation to reduce index creation overhead. We also design several benchmarks to evaluate DOT-based systems. Experimental evaluations with state-of-the-art index techniques (CCIndex and Apache Phoenix) show that our design can reduce the insert overhead for secondary indices to just 23%. Evaluation with TPC-H queries demonstrates an increase in query throughput by up to 2x, while application evaluation with real-world workloads and data (100M records) provided by AdMaster Inc. show up to 35% reduction in execution time.**

***Keywords*-DOT, RDMA, HBase, Indexing, CCIndex, Phoenix**

I. Introduction

Data is being generated at an exponential rate. The problem of storing this ever growing data is a challenge which has motivated researchers to come up with solutions such as Google's Bigtable [12], Yahoo's PNUTS [14], and Apache HBase [2]. These systems can be thought of as distributed NoSQL [24] databases because of similarities in many design and implementation strategies. They allow data to be modeled as a Distributed Ordered Table (DOT), which partitions the table data using continuous keys into regions distributed on several nodes and replicates each region for fault-tolerance.

Web 2.0/3.0 [23] and social-networking applications require a flexible data store and are thus built on top of Bigtable and HBase like systems [11]. The low latency requirement of these applications imposes heavy data analysis (querying) requirements on the underlying DOTs. SQL-like range queries over DOTs is an important requirement to analyze data. However, existing schemes in Bigtable and HBase to evaluate range queries are not efficient. This is because any query requiring data filtering based on values of non-primary key columns can only be resolved by going over all records in the DOT. Numerous solutions [8], [30], [17], [25], [28], [29], [3] have been proposed to solve this challenge by building secondary indices on DOTs to allow fast querying. CCIndex [30] is a state-of-the-art index technique that provides high-performance querying of DOT data. Apache Phoenix [3], a library built on HBase, provides several such index techniques. Although these solutions work extremely well for querying data, they introduce additional overhead in the form of secondary index creation. Each insert operation requires the secondary index tables to be updated as well. Moreover, the network performance characteristics of index techniques on DOTs have not been evaluated in a systematic manner.

Existing DOT systems and their implementations are based on Java TCP [18] sockets communication. The requirement for TCP sockets communication to support several legacy features has crippled its performance and development. The primary problem with TCP sockets is the need to copy data from user buffers into kernel buffers and context-switch to send each message [10], [19]. This drastically increases the network latency and decreases application performance. Modern networking interconnects such as InfiniBand [6] and RoCE [9] offer high-performance and low latency communication protocols using Remote Direct Memory Access (RDMA) which have been widely used by the HPC community. However, the use of RDMA to accelerate index techniques has been relatively unexplored.

Focus	[20]	This paper
RDMA design	√	√
Supported Index Techniques	×	CCIndex, Apache Phoenix
Operations	Get, Put	Get, Put, Scan
Thread Models	Functional partitioning	Functional partitioning, thread-per-request, bounded thread pool
Benchmarks	YCSB	YCSB, TPC-H
Applications	×	AdMaster Inc.
Data Set	Synthetic data	Synthetic and real data

Table I
Comparison with Earlier Work

Web services and applications are becoming more complex and their demand is steadily increasing. This requires the design of new systems that can handle this ever-increasing

*This research is supported in part by National Science Foundation grants #IIS-1447804, #ACI-1664137, #IIS-1636846, and #CNS-1513120 and National Key Research and Development Program of China (Grant No. 2016YFB1000604, 2016YFB0201404).

load and provide a robust and responsive service platform. Much of the previous work [27], [22] highlights the need for a highly decoupled and functionally partitioned (FP) model. Such models are believed to ensure fair response times to clients while ensuring high request processing throughput. These models are designed using processes and threads as the models for concurrent programming. Different functional activities are assigned to different pools of threads, thus utilizing the plethora of cores available in modern processors. However, such designs overlook the overhead of thread synchronization and data transfer between cores, particularly when each request takes only a brief amount of time to be processed. This is particularly true when using RDMA-based communication, where the advanced networking hardware provides extremely low latency communication and offload capabilities. This leads us to the following broad challenges:

- What are the performance characteristics of index techniques on DOTs from a networking perspective?
- How can a high-performance RDMA-based communication framework for index techniques on DOTs be designed to reduce network communication time?
- What is the impact of different thread models for highly concurrent DOT workloads?
- Can an RDMA-based design improve the performance of applications built on DOTs?

In this paper, we propose an RDMA-based communication framework for index techniques on Apache HBase to take advantage of modern networking interconnects to reduce network latency and improve performance. While a previous RDMA-based design [20] exists, they don't focus on index techniques. Indexing techniques bring with them a deluge of challenges, which we focus on solving in this paper. Table I presents a comparison of this paper with the earlier work. Our work proposes various thread models for achieving high request concurrency. We also propose design changes to record inserting implementations to reduce the additional overhead incurred for each insert operation. To analyze the performance of DOT-based queries on our proposed design, we design test queries over TPC-H [16] data and application data provided by AdMaster Inc. To summarize, the main contributions of this paper are as follows:

- Performance characterization of the networking requirements of index techniques on DOTs
- RDMA-based communication framework built on HBase to accelerate index techniques
- Thread models for highly concurrent request processing
- Query benchmarks to evaluate the performance of index techniques on DOTs
- Performance evaluation with CCIndex and Phoenix with TPC-H and AdMaster Inc. application workloads

Our experimental evaluations show that our design can reduce the insert overhead for secondary indices in CCIndex to just 23%. Evaluation with TPC-H queries shows an increase in query throughput for CCIndex and Phoenix by up to 2x while evaluation with application workloads presents up to 35% reduction in execution time.

The rest of this paper is organized as follows. Section II discusses the background for our work and Section III presents a comprehensive performance characterization of index techniques on HBase. Section IV presents our proposed design to accelerate index techniques on HBase, Section V discusses the querying benchmarks we propose, and Section VI demonstrates a performance evaluation of our proposed design. Section VII discusses related work and Section VIII concludes the paper.

II. Background

A. Existing Design of RDMA-HBase

Apache HBase [2] is an open-source distributed NoSQL database modeled after Google's BigTable [12]. RDMA-HBase [20] is a library built on Apache HBase that can be used on RDMA-enabled clusters to exploit the benefits of RDMA. RDMA-HBase implements high-performance designs for data transfer with get and put operations in HBase. Their implementation is based on the popular functional partitioning model (also adopted by Apache HBase), where separate thread pools are used for compute, network, and I/O components of each operation. While their implementation is able to extract optimal performance from RDMA-enabled hardware, they don't consider running index techniques over HBase where several additional challenges need to be addressed. Most importantly, they don't provide a high-performance design of the scan operation, which is the primary requirement when querying data from the tables. Moreover, functional partitioning is assumed to be the best thread model for their implementation, but no evaluations are presented to support this claim. There is no discussion on how to allocate threads for different functional thread pools in their design. Evaluation with real-world applications is also missing in their work. We aim to solve these challenges in this paper.

B. Index Techniques

CCIndex [30] is a complemental clustering index on Distributed Ordered Tables for accelerating multi-dimensional range queries built on Apache HBase. CCIndex builds complemental clustering index tables (CCITs) for each index column. The index tables are regular HBase tables and are split into regions and stored on RegionServers. Each CCIT contains data for all columns. Thus, range queries can be evaluated using a simple **scan** operation and involve no random reads. CCIndex uses the region-to-server mapping information provided by HBase meta-tables to estimate the result size of queries and optimize the query plan.

Apache Phoenix [3] is an open-source library built on top of Apache HBase. Phoenix enables Online Transaction Processing (OLTP) on HBase by providing APIs for SQL queries and ACID transaction guarantees. Phoenix compiles SQL queries to HBase scan operations so that they can be directly run over HBase. Moreover, Phoenix also provides several index techniques on HBase DOTs to accelerate queries, namely global covered index, global uncovered index, and local index. Global covered and uncovered indices are similar to CCIndex. However, global uncovered index does not have the complete column data in each index table, whereas

global covered index contains column data in each index for columns specified during index creation. Local index co-locates index and original table data by keeping index data in shadow column families in the same table to minimize network communication required for index **scan**. However, it incurs additional overhead for locating index data while scanning the index, since the location of index data cannot be pre-determined.

Index related operations, such as scan, are performance critical operations. Thus, a performance characterization of these operations is needed first.

III. Performance Characterization of Index Techniques on HBase

There are several index techniques that have been proposed over HBase. CCIndex is a state-of-the-art index technique that delivers high-throughput and low latency querying on DOTs. As discussed in [17], CCIndex has the best querying performance among several index techniques. Apache Phoenix provides the ability to run SQL queries on HBase directly. Adoption by several enterprise communities has made Phoenix prominent in the context of operational analytics. Based on these observations, we do all evaluations with CCIndex and Phoenix and implement our designs on top of these techniques.

To understand the characteristics of different operations on DOTs, we divide the usage of DOTs into three stages, namely loading data, updating existing data, and querying data. Loading data can be classified as either on-line or off-line loading depending on whether the data already exists while creating the DOT. Most DOT implementations provide a tool for (off-line) bulkloading of existing data into the table. The tool leverages MapReduce to parallely transform the existing data into DOT format and directly copies it into internal DOT files, bypassing the insert code path. On-line inserting and updating of data can be done using the insert and update operations provided by the DOT. DOT supports range queries over primary key and Multi-Dimensional Range Queries (MDRQs). We study the requirements of each of these operations for index techniques on DOTs and how RDMA can be used to benefit each operation.

Bulkloading. (Fig 1(a)) Bulkloading is implemented using the MapReduce framework and HDFS as the underlying filesystem. A MapReduce job is run on the uploaded data to partition data into regions and transform it into HBase format which are directly loaded into each RegionServer. From related works in the field [26], [21], we know that data shuffling in MapReduce and writing files in HDFS are network intensive operations. Thus, the bulkload operation can be considered to be network intensive.

Insert. (Fig 1(b)) Insert is implemented using put operations in HBase. For insert, multiple put operations are involved since all index tables along with the original table have to be updated. The first put operation inserts the record in the appropriate RegionServer for the original table. It is the responsibility of this RegionServer to insert the record into the index tables. This is done with the help of a co-processor in the server side which is invoked after each put operation.

Update. (Fig 1(c)) For update, get, delete, and put HBase operations are required. Get operation is required here to check if any index value has been updated. In case an index value has changed, a delete and subsequent put operation is required to update the index table.

OR-based Range Query. (Fig 1(d)) To evaluate OR-based queries, for each range column the corresponding index table needs to be scanned. After that the results need to be aggregated and returned to the user. Scanning of index tables is done in parallel.

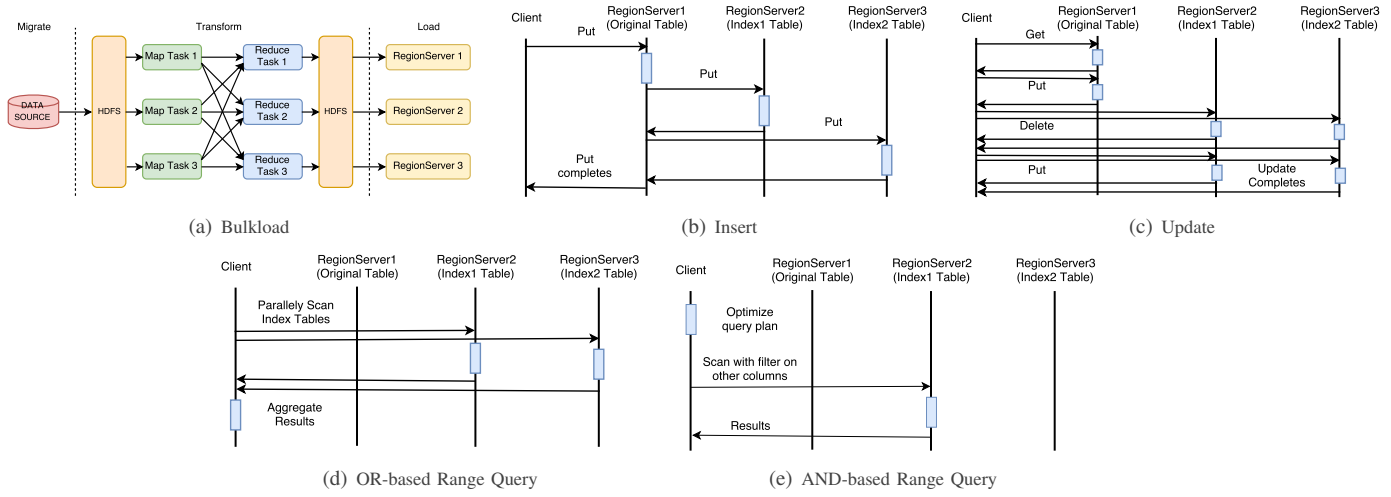
AND-based Range Query. (Fig 1(e)) To evaluate AND-based queries, only one index table is scanned with filters on other columns. Selecting which index table to scan is done by the query optimizer. CCIndex uses region-to-server mapping information in HBase to estimate result size and select the optimal index table to scan. Phoenix also provides a query optimizer, however, its implementation is not well documented.

As discussed above, the main operations of concern for a DOT are bulkload, insert, update, and query. It is clear that the performance of get, put, and scan HBase operations are critical to the overall performance of the application. To understand the performance characteristics and bottlenecks of primitive HBase operations, we conduct a comprehensive profiling analysis of these operations using the Yahoo Cloud Serving Benchmark (YCSB) [15] with one client and 1 KB record size. Fig. 2(a) shows the time breakup of these operations. It is evident that network communication time takes up a big chunk of each operation (27.38% for scan, 25.7% for put, and 36.5% for get). Since scan is heavily used for querying, we also analyze the time-line of the operation on both the server and client sides (Figures 2(b) and 2(c)). The timeline graphs reveal that there is no overlap between the compute and network phases which limits the overall throughput of the system.

Network transfer in HBase and Hadoop is through TCP sockets. The main overhead in the TCP stack is because of the need to copy data from userspace buffers to kernel buffers and context switch to transfer each message. InfiniBand on the other hand processes all communication in userspace and in a 'zero-copy' manner. InfiniBand supports RDMA-based semantics to allow one-sided data transfer. This provides an opportunity to use RDMA-based semantics in InfiniBand to accelerate various HBase operations required for DOT operations. As we have observed, HBase operations are network intensive and have no overlap between the compute and network phases. Thus, the challenge here is to design a high-performance communication framework which provides good overlap between computation and communication and can improve the performance of applications built on DOTs.

IV. Performance Acceleration of Index Techniques on HBase

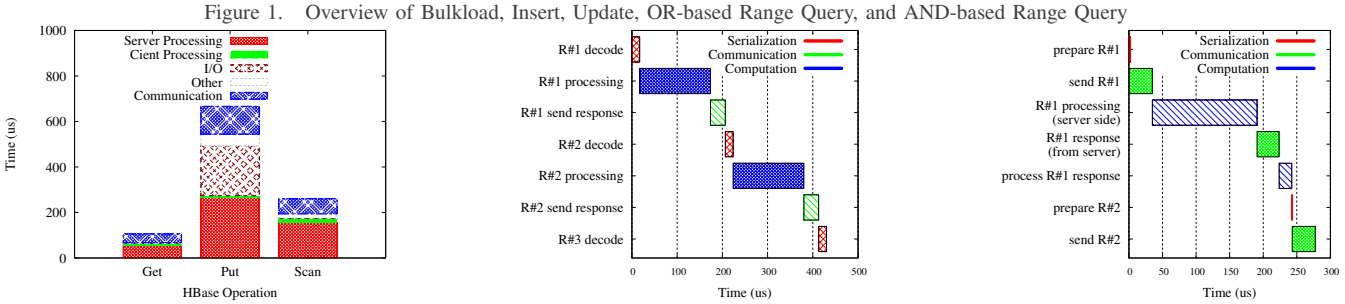
In this paper, we propose designs to accelerate index techniques on HBase using RDMA. In this section, we describe the main components of our design and then illustrate our



(a) Bulkload (b) Insert (c) Update

(d) OR-based Range Query (e) AND-based Range Query

Figure 1. Overview of Bulkload, Insert, Update, OR-based Range Query, and AND-based Range Query



(a) Time Breakup of HBase Operations (b) Server Timeline for Scan Operation (c) Client Timeline for Scan Operation

Figure 2. **Breakup and Time-line of HBase Operations.** Server and Client Processing represent time spent in serializing and de-serializing the message, I/O represents time spent writing data to disk, and communication represents time taken to send the message over the network. R#n represents request number n. It is assumed that the requests are sent one after another.

implementation. While our designs are based on HBase, they can be extended and applied to any DOT system in general.

A. RDMA-enhanced HBase for Indexing Techniques

The main components of our proposed RDMA-enhanced HBase are described below.

RDMA Connection Manager. We introduce a dedicated RDMA connection manager on both the server and client side. The connection manager runs in a separate thread and builds connections with other nodes on demand. It also contains a connection cache which stores the connection information data structures in memory for fast communication. This also ensures that we reuse existing connections to remote nodes and don't create excess or unnecessary connections, thereby reducing network latency.

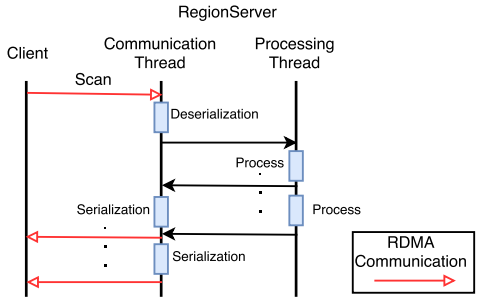
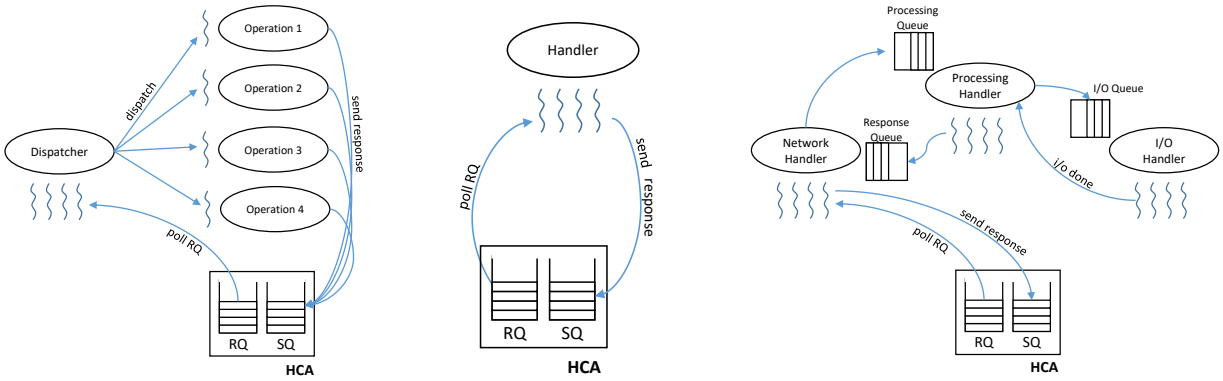


Figure 3. Querying using RDMA-based Scan

RDMA-enhanced Scan. As discussed in Section III, apart from get and put operations, scan is a major requirement for basic operations on DOTs. Thus, we design an RDMA version of the scan operation to accelerate querying performance. Fig. 3 shows our RDMA-based scan design to accelerate queries. With scan operations, the result count varies based on the requested range. Since the result count is not known when a scan operation is issued, HBase divides the scan response into multiple chunks. As soon as the processing thread has scanned enough records, it packages the records into one message and sends it to the communication thread. The communication thread uses RDMA communication to send all response messages to the client. Thus, for large result count, multiple messages will be sent over the network and our RDMA-based design should considerably improve the query performance. We also use non-blocking semantics for network communication on the server side. This is done by offloading the actual message transfer protocol to the network adapter, which allows for maximum overlap between computation and communication, thereby increasing the effective querying throughput. Note that although we have shown two separate threads to perform communication and computation, they can be performed by the same thread as well, depending on the thread model (see Section IV-B).

Buffer Management. Since memory is often a limiting resource, we opt for off-JVM heap buffer allocation for all internal RDMA communication buffers and data structures. This is done by allocating memory for RDMA communication directly in the JNI layer and making it available in the Java



(a) Threaded Server Design (TS)

(b) Shared Threaded Server Design (STS)

(c) Functional Partitioning-based Design (FP)

Figure 4. Thread Models for RDMA-based HBase Design

layer as a DirectByteBuffer. We use ByteBufferOutputStream and ByteBufferInputStream to allow HBase to directly put the communication data into the RDMA communication buffers and preventing the need for additional buffer copies.

B. Thread Models

CCIndex and Apache Phoenix reduce the complexity and computation required for each scan operation when querying table data, resulting in short execution times for each scan operation. While the traditional HBase design, as well as the previous RDMA design were based on the popular functional partitioning design approach [22], we argue that it might not be the best approach for running querying workloads over HBase with index techniques. Functional partitioning is considered to be particularly useful for ensuring fair response times to clients. This argument holds true when each request takes a significant amount of time to be resolved but breaks down if the request only takes a brief amount of time. In this scenario, the overhead of thread synchronization and data transfer between cores can lead to significant performance degradation. This is particularly significant when using RDMA-enabled high-performance interconnects, as the network communication latency is extremely low and can be offloaded to the network adapter. This calls for redesigning the HBase server architecture, particularly for scan operations.

To avoid any performance degradation and scaling to a large number of clients, we believe that a single- threaded approach (i.e. single thread to process entire request) is the best solution. Using a single thread to process the complete request ensures that we don't run into any thread synchronization issues and prevent the need for data transfer between cores. The thread-per-request model and bounded thread pool models are two of the most common single threaded designs. The thread-per-request model is well known for its simplistic design and application in web services [13]. The bounded thread pool model is a variant of the thread-per-request model which imposes a limit on the total number of threads. Our goal here is to show that these models work well for CCIndex and Phoenix over HBase. We implemented three designs based on these single threaded models as well as the default functional partitioning approach for scan operations. These designs are presented in Fig. 4 and the integration of these designs in HBase is presented in detail below.

Threaded Server Design (RDMA-HBase-TS). This design is based on the thread-per-request model, utilized in several web servers. In this design, each HBase operation consumes a thread, with the OS responsible for switching between threads to ensure fair response times for clients. While this approach is relatively easy to program, it can lead to significant performance degradation when the number of threads is large, owing to the overheads associated with threading. In this design, we have a dispatcher thread pool for reading operation requests from the network adapter and launching a thread to process the request in. Each operation thread is responsible for processing the complete operation including posting the network operation for sending the result to the InfiniBand Host Channel Adapter (HCA).

Shared Threaded Server Design (RDMA-HBase-STS). While the threaded server design suffers from the threading overhead for a large number of threads, this design aims to solve this issue by using a bounded thread pool for operation processing. The entire design uses just one thread pool to process the entire operation. Incoming requests wait in the HCA's queues until one of the handler threads pick them up for processing. Limiting the number of concurrent threads leads to a more scalable and robust design as compared to the unconstrained threaded server design. It should be noted that the number of threads can and should be tuned.

Functional Partitioning-based Design (RDMA-HBase-FP). Functional partitioning typically involves partitioning the application's activities into a set of distinct functions which are each processed by separate threads. The interaction between functional units usually takes place through shared queues and variables. In our case, network, computation, and I/O can easily be identified as the main activities. Separate thread pools are assigned for these three tasks with communication between the thread pools using thread-safe queues. The default HBase implementation and the previous RDMA implementation both use a similar version of this design.

In all of these designs, we make use of the advanced networking capabilities of InfiniBand adapters to offload communication to the HCA. This is done by making use of the hardware queues provided by the HCA - Send Queue (SQ)

and Receive Queue (RQ). We maintain the SQ and RQ in a highly optimized manner to ensure optimum parallelism in the network and enable full utilization of the bandwidth offered by the InfiniBand hardware.

Finding the optimal number of threads to use for the STS and FP designs is a challenge, particularly for FP. While solutions like SEDA [27] propose dynamic resource provisioning through resource controllers, and others advocate for static allocation of resources, none of these solutions provides a way to find the optimal number of threads for each functional unit in FP. We claim that allocating threads to functional units in the ratio of the execution time of each function delivers the best performance. Through thorough evaluation, we realized that this approach works extremely well when combined with full subscription allocation (one thread per core) and delivers the best performance.

C. Parallel Insert

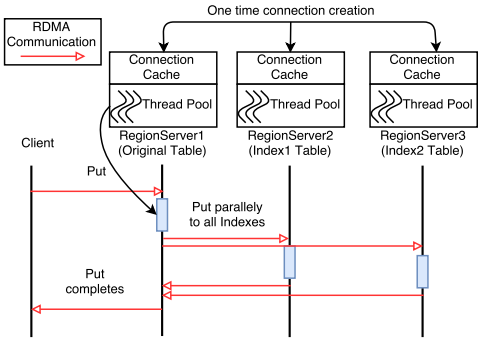


Figure 5. Parallel Insert

As discussed in Section III, inserting records has considerable overhead for any index technique. This is because each insert operation requires multiple put operations to update the index tables. Our RDMA-enhanced communication engine will help in reducing the latency of each put operation, thus reducing the latency of the insert operation. However, as shown in Fig. 1(b), CCIndex updates the index tables sequentially. This is highly inefficient and has a significant impact on insert latency. Moreover, for each insert operation, the RegionServer creates a new connection with other RegionServers to update necessary CCITs. The time taken for connection establishment is significantly higher than the time taken for a put operation. Based on these observations, we design a parallel version of the insert operation as shown in Fig. 5. We add a cached thread pool in each RegionServer which is initialized when the server starts and only creates new threads when required. For each put operation, we fetch a thread from the thread pool and run the put operation in that thread after which the thread is returned to the thread pool. We also introduce a connection cache on each RegionServer to avoid the overhead of connection establishment for each insert operation. Note that this connection cache is different from the connection cache proposed in the previous subsection as it is only meant for caching connections between RegionServers.

D. Implementation

Fig. 6 shows the system architecture of our design. Since our design is implemented on HBase, CCIndex, and Phoenix, we expose their client-side APIs for use by applications. Depending on which underlying index and framework the application requires, it has to use the appropriate client API. However, since Phoenix has a built-in SQL engine, it can directly run the application SQL queries without any modification. Underneath the client APIs, we propose an RDMA-based communication framework built on HBase. For building RDMA connections, we add a dedicated connection manager. The client APIs are mapped to the appropriate underlying RDMA-based RPC calls. On the server side, there is a processing engine which processes requests from the client. For processing index related tasks, co-processors within the processing engine are utilized. For communication with the client, the server also has an RDMA-based communication engine along with a connection manager. The overall design works as follows. The application issues an operation using the client APIs. The client API processes the request and converts it into a series of RPC calls. These are then handed off to the RDMA-based communication engine which communicates with the server RPC engine. The server processes the call and sends a response via the RPC engine back to the client where it is returned to the application.

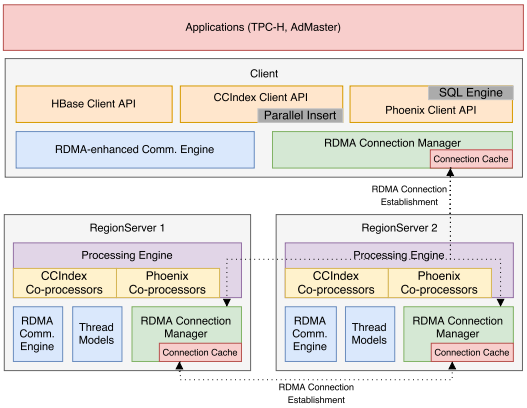


Figure 6. System Architecture

The overall fault-tolerance of the proposed system is unaffected. If we look at the semantics of DOT operations with our proposed design, the network communication semantics have clearly changed. However, it is easy to see that the operational semantics of our design are exactly the same as the original design. Our design changes only impact the network communication code-path, the thread model, and increase the overlap between computation and communication, but preserve the overall order of execution. Any failure in the network communication will be handled by the retransmission logic in the upper layer. This scenario will be rare due to the highly reliable communication offered by InfiniBand.

V. Designing Benchmarks for DOT Systems

To test the performance of our design using DOT-based queries, we design benchmark queries using the TPC-H benchmark suite. TPC-H [16] is a popular benchmark for evaluating the performance of databases. It is commonly used by customers for evaluating and comparing the performance of data warehousing solutions. We analyze the table schema of the TPC-H orders table to come up with a set of queries which can capture the characteristics of most

database application queries. We create four SQL queries for the orders table provided with the TPC-H data. Query 1 is a typical multi-dimensional range query and Query 3 is a typical one-dimensional query. Queries 2 and 4 are fine-grained multi-dimension range queries. Specifically, Query 2 represents the application of multi-dimension OR range query and Query 4 represents the application of multi-dimension AND range query. These queries are the most typical and the most commonly used microbenchmarks which could represent applications based on DOTs.

We also create multi-client benchmarks based on Query 2 and 4, with each client querying a different range of values to achieve load-balancing. Based on the table schema and our sample queries, we use *orderkey* as the primary key with *custkey* and *orderdate* as index columns. To test our design with some real-world application, we use data provided by AdMaster Inc. [1]. AdMaster Inc. is a marketing data technology company that uses Big Data to provide businesses with useful marketing data. The data consists of a set of events for users interacting with ads on smartphones. The overall dataset has more than 50 fields for each record. After discussion with AdMaster data scientists, we have come up with four workloads which represent queries that are daily used by AdMaster to analyze their data. Based on the queries, we use *date* and *os* as the index columns. Table II lists all the benchmark queries that we have proposed.

Phoenix provides a built-in SQL engine to allow applications to directly run SQL queries without any modifications. However, both HBase and CCIndex do not provide such an engine. Thus, we provide an implementation of our synthetic application and AdMaster workload queries based on the CCIndex and HBase client APIs. This allows us to compare the performance of our design with HBase, Phoenix, and CCIndex.

VI. Performance Evaluation

A. Experimental Testbed

Our experimental testbed consists of an InfiniBand cluster with 20 nodes. Each node has two 14 core 2.4 GHz Intel Xeon E5-2680 (broadwell) processors, 512 GB main memory, Mellanox ConnextX-4 EDR (100 Gbps) HCA, and runs CentOS 7.2. Each node also has four 2 TB HDDs and one 400 GB SSD. All experiments are performed with HBase 1.1.2, Apache Phoenix 4.8.1, and CCIndex 1.1.2 (based on HBase-1.1.2). Our RDMA-HBase implementation is based on HBase 1.1.2. For the bulkload experiment, we use an HBase cluster with 4 RegionServers and 1 HMaster. For application-level evaluations, we use a 12 RegionServer and 1 HMaster HBase cluster. For all other experiments, we use a 10 RegionServer and 1 HMaster HBase cluster. Each RegionServer node also runs a DataNode and NodeManager instance, and the HMaster node has a NameNode and ResourceManager instance. The remaining nodes in our testbed are used as client nodes.

B. Microbenchmarks

We evaluate the performance of primitive HBase operations using our design and compare it with default HBase. The performance of the thread models is quite similar for just one client. Thus, we evaluate the RDMA-based design using only the FP model. Fig. 7 shows the result of this analysis. We observe that our design improves the latency of all operations. In addition, the communication time is reduced by a factor of 5.7x, 3.2x, and 5.6x for scan, put, and get, respectively. These results indicate that performance benefits can be obtained by reducing the network communication time. The server processing time is marginally better for our design owing to the efficient buffer management scheme employed in our design. We do not evaluate with [20] because their design is based on a very old version of HBase.

As discussed in Section II, Phoenix provides multiple index techniques. To understand the performance of each technique, we run Queries 2 and 4 with each index. Fig. 8 shows the result of this analysis. It can be observed that the global covered index performs the best for both the queries. This is expected since global covered index keeps data for all columns in each index table, thus not requiring any random reads. Since, global covered index performs the best, for all further evaluations with Phoenix, we use global covered index.

C. Insert

To test insert performance, we create a table with two index columns and one column family. We insert a total of 5 columns for each record with 8 clients running in parallel and varied the record size. Fig. 9 shows the result of this experiment. The default CCIndex insert performance is considerably worse than HBase because of the inefficient sequential insert design and need to create new connections for each insert operation. By leveraging RDMA-based communication, a parallel insert design, and a connection cache, our design reduces the insert overhead from 1541%, 317%, and 346% to 94%, 23%, and 57% for 16k, 32k, and 64k record sizes, respectively.

D. Synthetic Querying Benchmarks

Single Client Evaluation. In this sub-section, we evaluate the performance of our proposed benchmark queries with our design. The performance of the thread models is quite similar for just one client. Thus, we only evaluate the RDMA-based design using the FP model. Figures 10 and 11 show the execution time for the TPC-H queries. For OR-based Queries 1 and 2, our design improves throughput up to 59% and 92% for CCIndex and up to 35% and 70% for Phoenix, receptively. Our design increases query throughput by up to 66.6%, and 51.1% when compared to CCIndex over HBase and by up to 81% and 100% when compared to Phoenix over HBase for AND-based Queries 3 and 4, respectively. We can observe that there is more improvement for Phoenix and CCIndex than with just default HBase. This is because our proposed designs are specifically targeted for index techniques on HBase. We also observe that Phoenix performance is not consistent and depends to a large extent on the type of query; sometimes even performing worse than HBase. To find the reason for this, we further analyze the Phoenix performance runs. We find that the Phoenix bulkload utility does not split HBase tables into regions in an efficient manner. The number of regions in these cases is less than what we see for CCIndex and HBase with the same data. More number of regions implies more parallel querying and load-balancing. Thus, it is possible

Query/Workload	SQL Syntax	Data Source
Query1	SELECT orderkey, orderdate, shippriority FROM orders WHERE custkey = C1 AND orderdate < O1	TPC-H
Query2	SELECT * FROM ORDERS where C1 < custkey < C2 OR O1 < orderdate < O2	TPC-H
Query3	SELECT orderpriority, count(*) as order_count FROM orders WHERE orderdate >= O1 AND orderdate < O2 GROUP BY orderpriority	TPC-H
Query4	SELECT * FROM ORDERS where C1 < custkey < C2 AND O1 < orderdate < O2	TPC-H
Workload1	SELECT count(*) as events, count(distinct distinctID) as users FROM event WHERE date BETWEEN D1 AND D2	AdMaster
Workload2	SELECT hour, count(*) as events, count(distinct distinctID) as users FROM event WHERE date BETWEEN D1 AND D2 GROUP BY hour ORDER by hour	AdMaster
Workload3	SELECT data, event, count(*) as events, count(distinct distinctID) as users FROM event WHERE date BETWEEN D1 AND D2 GROUP BY date, event ORDER by events DESC	AdMaster
Workload4	SELECT event, count(*) as events, count(distinct distinctID) as users FROM event WHERE date BETWEEN D1 AND D2 AND os = O1 GROUP BY date, event ORDER by events DESC	AdMaster

Table II
BENCHMARK QUERIES

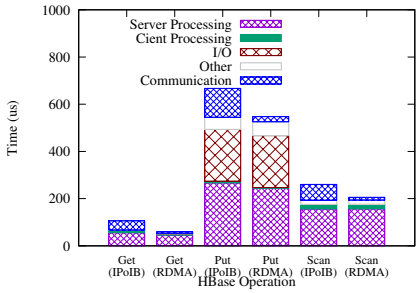


Figure 7. Time Breakup of HBase Operations

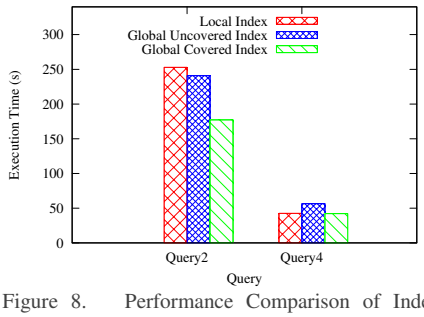


Figure 8. Performance Comparison of Index Techniques in Apache Phoenix

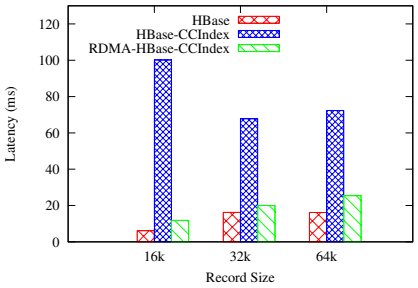
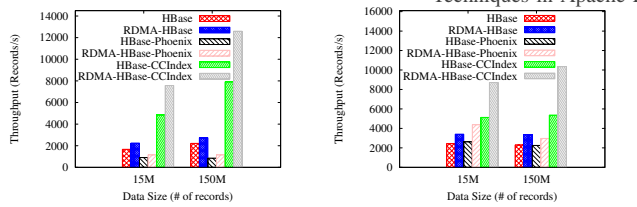
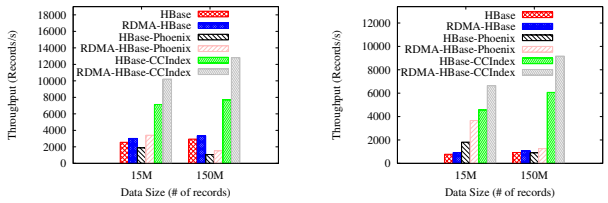


Figure 9. Comparison of Insert Performance



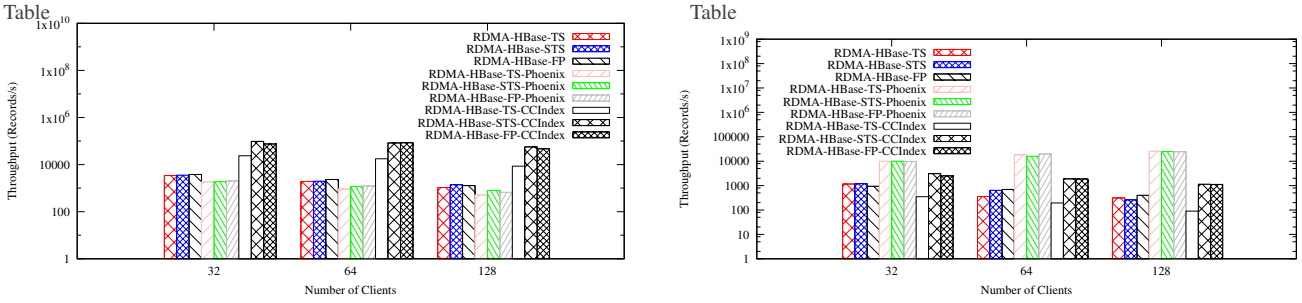
(a) Query1 (b) Query2

Figure 10. Throughput of OR-Based Range Queries on TPC-H Orders Table



(a) Query3 (b) Query4

Figure 11. Throughput of AND-Based Range Queries on TPC-H Orders Table



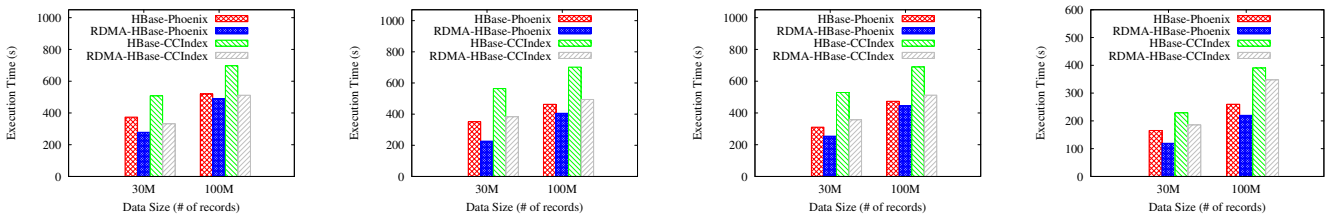
(a) Query2 (b) Query4

Figure 12. Performance Comparison of Thread Models with Queries on TPC-H Orders Table

that through parallel querying and load-balancing, HBase can achieve higher throughput than Phoenix using indices. This depends largely on the type of query, which is why we see inconsistent performance with Phoenix.

Multi-Client Evaluation. Next, we compare the performance of our proposed thread models in HBase using the multi-client versions of Queries 2 and 4. As we have already seen that the RDMA-based design is much better than the default one, we only compare the RDMA-based designs in this evaluation. This evaluation is done with 15M records. Results are presented in Fig. 12. We observe interesting trends with these results. Most importantly, STS performance is better than FP for HBase and Phoenix in most cases (very close in others). At 128 clients, STS is better than FP for both queries for CCIndex and Phoenix. While for HBase, FP is better than STS. This trend validates our claim that single threaded models are good for index techniques, but not for default HBase because of the short processing time of scan operations with CCIndex and Phoenix. While TS performance is comparable to STS and FP for a small number of clients, it rapidly degrades upon increasing the number of clients, as expected.

Interestingly, Phoenix performance is extremely scalable for the AND-based query, while for the OR-based query it does not scale at all. For CCIndex, the trend is completely opposite. This can be attributed to the internal query optimizers in CCIndex and Phoenix. We believe that the CCIndex query optimizer works extremely well for OR-based queries, while the Phoenix one works well for AND-based queries. Our results with other queries also validate this claim. We find that the highly optimized query plan for Query4 with Phoenix is



(a) Workload 1 (b) Workload 2 (c) Workload 3 (d) Workload 4

Figure 13. Latency of AdMaster Workloads

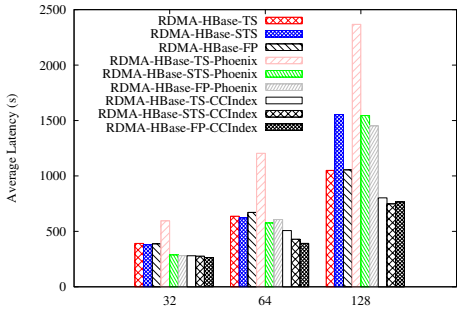


Figure 14. Performance Comparison of Thread Models with Multi-Client AdMaster Workload

the reason why TS performs so well for Phoenix. Surprisingly, with our experimental testbed, we found that using just two threads in the main handler pool delivers the best performance for the STS design. This is a big advantage of using the STS design as it frees up the CPU for performing other tasks while delivering even better performance than FP, which utilizes all the cores in the CPU.

E. Application-level Evaluation

To test our design with some real-world application, we use workloads and data provided by AdMaster. The total data size is 100 Million records with an average record size of 524 bytes. We evaluate the workload performance with the whole dataset and a subset of the data (30M). Fig. 13 shows the performance of these workloads with our FP design. Compared with CCIndex over default HBase, our design improves the workload execution time up to 34%, 32%, 32%, and 19% for Workloads 1, 2, 3, and 4, respectively. For Phoenix over HBase, our design improves execution time up to 25%, 35%, 18%, and 27% for Workloads 1, 2, 3, and 4, respectively. Thus, our design can provide benefits at the application level, proving the efficiency of our approach.

We also design a multi-client workload which incorporates all the application workloads. In this workload, we run multiple clients such that each workload is being run by a quarter of the clients. We feel that this multi-client workload accurately embodies the characteristics of the load and request distribution received by AdMaster databases. This is because, in real world applications, users typically run different types of queries concurrently. We use 15M records from the AdMaster data for this evaluation. Results for this evaluation are presented in Fig. 14. Similar to the TPC-H multi-client benchmark, we observe that STS and FP have similar performance for CCIndex and Phoenix, while for HBase FP has the least average latency for 128 clients. TS performance suffers from the overheads of threading, particularly for 64 and 128 clients. To achieve optimal performance with STS, we observe that we need to increase the number of main handler threads from two (in case of TPC-H) to four. This is primarily because clients are running different workloads, thus more handlers are needed to ensure that a big request does not block other requests. Thus, while STS and FP performance are similar, STS still uses significantly less number of threads.

F. Discussion

Our experimental evaluation provides interesting insights into the performance characteristics of SQL queries on DOTs with different index techniques and thread models. While an RDMA-based design always performs better, the same can not be said about CCIndex or Phoenix. Both perform well in different scenarios. CCIndex performance is scalable for OR-based queries and Phoenix performance is scalable for AND-based queries. The internal query optimizers are responsible for this trend. In addition, inefficient table splitting logic used by the Phoenix bulkload utility leads to inconsistent querying performance. This opens up possible future research avenues like enhancing the query optimizer and bulkload utility. Evaluation of different thread models shows that a functionally partitioned model is sub-optimal when dealing with index techniques. While this model does perform well, the shared threaded model can provide similar performance with significantly fewer cores. This analysis should be kept in mind when designing DOT-based systems, especially when using any form of indexing.

VII. Related Work

A lot of research has been done on building indices on database systems. With distributed NoSQL databases such as HBase becoming popular in recent times, existing research has been ported over to HBase and more index techniques have been proposed on HBase. IHBase [5] is built on HBase and builds indices on tables to reduce the range of rows scanned when querying the table. It builds the index by directly working on the internal HBase files after they are flushed to disk. However, it still requires random reads for querying data. Authors in [17] propose a Local and Clustering Index (LCIndex) on HBase. Their main goal is to improve insert performance for secondary indices while compromising on querying performance. IRIndex [7] or Inside Region Index, builds an index for each file in HBase, instead of each region. For processing queries, IRIndex sorts keys of the original table before scanning the table, thereby drastically reducing random reads. CCIndex [30] and Apache Phoenix [3] also provide index techniques on HBase which have already been discussed at length in this paper.

Improving HBase performance is another area where a lot of research has been done. AsyncHBase [4] provides with an asynchronous version of the HBase client API allowing for more fine-grain control of HBase operations and flexibility when writing applications. Authors in [20] use RDMA to accelerate HBase. They propose RDMA-based get and put operations in HBase and show benefits using YCSB. Their approach uses a functionally partitioned model for achieving high concurrency. However, they consider only get and put operations and disregard other important operations such as scan. In addition, their designs are not specifically optimized for index techniques. Our work is unique in the sense that we thoroughly analyze the requirements and bottlenecks of index techniques on HBase from a networking perspective and propose designs to accelerate them. In addition, our benchmark queries and data are from real-world applications.

VIII. Conclusion and Future Work

In this paper, we first analyzed the requirements of index techniques on DOTs from a networking perspective. Based on our experiences and observations, we proposed an RDMA-based communication framework to reduce the latency of primary operations on DOTs. As proof of concept, we implemented our design using CCIndex and Phoenix over HBase. We found that there is a lack of benchmarks to evaluate index techniques and proposed multiple benchmarks to fill this gap. We explored different thread models for our design and showed that single thread designs are more suited for index techniques. Our evaluations show that using an RDMA-based communication framework can reduce the latency of get, put, and scan operations in HBase. Our insert design reduces the insert latency by up to 8x. Our design can reduce execution time up to 35% for real-world AdMaster workloads and data (100M records). Thus, our design not only allows for faster analysis of data but at the same time reduces the overhead for inserting data. Our designs can significantly benefit companies such as AdMaster Inc. perform faster data analysis leading to better decision making.

In the future, we plan to evaluate with more index techniques and find additional application scenarios for our proposed design. We also plan to make our benchmarks, data, and designs publicly available for community use.

References

[1] AdMaster. http://www.admaster.com.cn/eng/.

[2] Apache HBase. http://www.hbase.apache.org.

[3] Apache Phoenix. http://phoenix.apache.org/.

[4] AsyncHBase. http://opentsdb.github.io/asynchbase/index.html.

[5] IHBase. https://github.com/ykulbak/ihbase.

[6] InfiniBand Trade Association. http://www.infinibandta.com.

[7] IRIndex. https://github.com/wanhao/IRIndex.

[8] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous View Maintenance for VLSD Databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 179–192. ACM, 2009.

[9] I. T. Association et al. Supplement to Infiniband Architecture Specification Volume 1, Release 1.2. 1: Annex A16: RDMA over Converged Ethernet (RoCE), 2010.

[10] M. Baker, H. Ong, and A. Shafi. A Study of Java Networking Performance on a Linux Cluster. *Distributed Systems Group, University of Portsmouth, UK*, 2004.

[11] R. Cattell. Scalable SQL and NoSQL Data Stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[13] D. A. Chappell and T. Jewell. *Java Web Services*. Tecniche Nuove, 2002.

[14] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[16] T. P. P. Council. TPC-H Benchmark Specification. *Published at http://www.tcp.org/hspec.html*, 2008.

[17] C. Feng, X. Yang, F. Liang, X.-H. Sun, and Z. Xu. LCIndex: A Local and Clustering Index on Distributed Ordered Tables for Flexible Multi-Dimensional Range Queries. In *Parallel Processing (ICPP), 2015 44th International Conference on*, pages 719–728. IEEE, 2015.

[18] B. A. Forouzan. *TCP/IP Protocol Suite*. McGraw-Hill, Inc., 2002.

[19] P. W. Frey and G. Alonso. Minimizing the Hidden Cost of RDMA. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 553–560. IEEE, 2009.

[20] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-Performance Design of HBase with RDMA over InfiniBand. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 774–785. IEEE, 2012.

[21] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High Performance RDMA-based Design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 35. IEEE Computer Society Press, 2012.

[22] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. Functional Partitioning to Optimize end-to-end Performance on Many-core Architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE Computer Society, 2010.

[23] T. O'Reilly. What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. *Communications & strategies*, (1):17, 2007.

[24] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012.

[25] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing Multi-dimensional Data in a Cloud System. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 591–602. ACM, 2010.

[26] M. Wasi-ur Rahman, N. S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-Performance RDMA-Based Design of Hadoop MapReduce over InfiniBand. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1908–1917. IEEE, 2013.

[27] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.

[28] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu. Efficient B-tree Based Indexing for Cloud Data Processing. *Proceedings of the VLDB Endowment*, 3(1-2):1207–1218, 2010.

[29] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng. An Efficient Multi-dimensional Index for Cloud Data Management. In *Proceedings of the First International Workshop on Cloud Data Management*, pages 17–24. ACM, 2009.

[30] Y. Zou, J. Liu, S. Wang, L. Zha, and Z. Xu. CCIndex: A Complemental Clustering Index on Distributed Ordered Tables for Multi-dimensional Range Queries. In *IFIP International Conference on Network and Parallel Computing*, pages 247–261. Springer, 2010.