Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks

Tobias Ziegler
TU Darmstadt
tobias.ziegler@cs.tu-darmstadt.de

Sumukha Tumkur Vani
Brown University
sumukha_tumkur_vani@brown.edu

Carsten Binnig
TU Darmstadt
carsten.binnig@cs.tu-darmstadt.de

Rodrigo Fonseca
Brown University
rfonseca@cs.brown.edu

Tim Kraska
MIT
kraska@mit.edu

ABSTRACT

Over the past decade, in-memory database systems have become prevalent in academia and industry. However, large data sets often need to be stored distributed across the memory of several nodes in a cluster, since they often do not fit into the memory of a single machine. A database architecture that has recently been proposed for building distributed in-memory databases for fast RDMA-capable networks is the Network-Attached-Memory (NAM) architecture. The NAM architecture logically separates compute and memory servers and thus provides independent scalability of both resources. One important key challenge in the NAM architecture, is to provide efficient remote access methods for compute nodes to access data residing in memory nodes.

In this paper, we therefore discuss design alternatives for distributed tree-based index structures in the NAM architecture. The two main aspects that we focus on in our paper are: (1) how the index itself should be distributed across several memory servers and (2) which RDMA primitives should be used by compute servers to access the distributed index structure in the most efficient manner. Our experimental evaluation shows the trade-offs for different distributed index design alternatives using a variety of workloads. While the focus of this paper is on the NAM architecture, we believe that the findings can also help to understand the design space on how to build distributed tree-based indexes for other RDMA-based distributed database architectures in general.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300081

ACM Reference Format:
Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3300081

1 INTRODUCTION

Motivation: In the last years, in-memory database systems have become dominant in academia and industry. This is not only demonstrated by the multitude of academic projects including MonetDB, Peloton and HyPer but also by the variety of available commercial in-memory database systems such as SAP HANA, Oracle Exalytics, IBM DB2 BLU, and Microsoft Hekaton. A major challenge of in-memory systems, however, is that large data sets often do not fit into the memory of a single machine anymore. To that end, in-memory databases often need to be stored distributed across the memory of a cluster of machines. For example, Walmart — the world's largest company by revenue — uses a cluster of multiple servers that in total provide 64 terabytes of main memory to process their business data.

An architecture that has recently been proposed for building distributed in-memory database systems is the Network-Attached-Memory (NAM) architecture [5, 39, 44]. The NAM architecture was specifically designed for high-performance RDMA-enabled networks and logically separates compute and memory servers as shown in Figure 1. The memory servers in the NAM architecture provide a shared and distributed memory pool for storing the tables and indexes, which can be accessed from compute servers that execute queries and transactions.

A major advantage of the NAM architecture over the classical shared-nothing architecture is that compute and memory servers can be scaled independently and thus the NAM architecture can efficiently support the resource requirements for various different data-intensive workloads (OLTP, OLAP, and ML [5]). Moreover, as shown in [44], the NAM architecture is less sensitive towards data-locality and can thus support workloads where the database is not trivially partitionable.

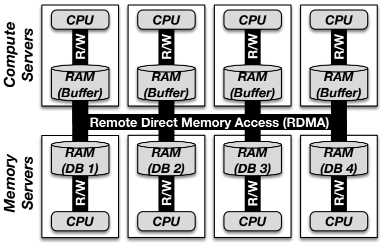


Figure 1: The NAM Architecture

As a result, a recent paper [44] has shown that the NAM architecture can scale out nearly linearly for transactional workloads (OLTP) up to clusters with more than 50 nodes while the classical shared-nothing architecture stops scaling after only a few nodes.

However, what enables the scalability of the NAM architecture is the advent of affordable high-performance networks such as InfiniBand, RoCE, or OmniPath. These networks not only provide high-bandwidth and low-latency, but also allow to bypass the CPU for many of the required data transfer operations using Remote-Direct-Memory-Access (RDMA), minimizing the CPU overhead for every data transfer. Unfortunately, taking full advantage of RDMA especially for smaller data transfers is not easy and as [44] points out requires a careful design for all in-memory data structures.

Previous work [15, 16, 21, 44] therefore made several proposals on how to design data structures mainly to support concurrent updates. However, all these systems assume that secondary indexes are not distributed and do not span more than one server. While it is a reasonable assumption for some workloads, it cannot only severely limit the scalability of the entire system, but also create hot-spots if the index is commonly read/updated, destroying one of the key advantages of the NAM architecture.

In this paper we therefore investigate if it is possible to design a scalable tree-based index structure for RDMA. Our focus is on tree structures in order to handle range queries efficiently, and on the NAM architecture because of its scalability, as well as its capability to separately scale compute and in-memory storage. However, designing a scalable tree-based index structure is not trivial and many design choices exists. For instance, accessing indexes via RDMA leaves the option whether to use one-sided RDMA operations, which do not involve the remote CPU, or two-sided RDMA operations, which are essentially RPC calls. One-sided operations are more scalable as they have less overhead, but unfortunately, they are more complicated to use [5, 17, 20, 44], and might require more than one round-trip. Furthermore, many approaches exists on how the index (inner and leaf nodes) should be distributed across the storage servers. Ideally, the distribution scheme not only leverages the memory resources of all available servers in a fair manner (e.g., the memory requirements are distributed uniformly across all machines), but is also robust towards different access patterns (i.e., uniform vs. skewed, different selectivities, different read/write ratios etc.)

Contributions: In summary we make the following contributions: (1) We discuss the design options of distributed tree-based index structures for the NAM architecture that can be efficiently accessed via RDMA. (2) We present three different possible index implementations that vary in the data distribution scheme as well as the underlying RDMA primitives used to access and update the index. (3) We analyze the performance of the proposed index designs using various workloads ranging from read-only workloads with various access patterns and selectivities to mixed workloads with different write intensities. Furthermore, the workloads used in our evaluation cover uniform and skewed distributions to show the robustness of the suggested index designs. As we will show in our experiments, both design questions, which RDMA primitives to use and how to distribute the index nodes, play a significant role on the resulting scalability and robustness of the index structure. Finally, we believe that the findings of this paper are not only applicable for the NAM architecture, but also represent a more general guideline to build distributed indexes for other architectures (e.g., the shared-nothing architecture) and applications (e.g., ordered key-value stores) over RDMA-capable networks.

Outline: The remainder of this paper is organized as follows: In Section 2 we first give an overview of the capabilities of RDMA-enabled networks and then discuss the design space for tree-based indexes in the NAM architecture. Afterwards, based on the design space we derive three possible tree-based indexing schemes, that we then discuss in detail in Sections 3 to 5. The evaluation, in Section 6, examines these index alternatives with various workloads. As mentioned before, we believe that the findings of this paper generalize to other distributed architectures. Some initial ideas in this direction are discussed in Section 7. Finally, we conclude with an overview of the related work in Section 8 and a summary of the findings and possible avenues of future work in Section 9.

2 OVERVIEW

This section provides an overview of the background of RDMA-capable networks relevant for this paper, discusses the design space of distributed indexes for RDMA and analyzes the scalability of the different alternatives. Readers familiar with RDMA can skip Section 2.1 and continue with Section 2.2.

2.1 RDMA Basics

Remote Direct Memory Access (RDMA) is a networking protocol that provides high bandwidth and low latency accesses to a remote node's main memory. This is achieved by using zero-copy transfer from the application space to bypass the OS kernel. There are several RDMA implementations available — most notably InfiniBand and RDMA over Converged Ethernet (RoCE) [42].

RDMA implementations typically provide different operations (called verbs) that can be categorized into the following two classes: (1) one-sided and (2) two-sided verbs.

One-sided verbs: One-sided verbs (READ/WRITE) provide remote memory access semantics, where the host specifies the memory address of the remote node that should be accessed. When using one-sided verbs, the CPU of the remote node is not actively involved in the data transfer.

Two-sided verbs: Two-sided verbs (SEND/RECEIVE) provide channel semantics. In order to transfer data between the host and the remote node, the remote node first needs to publish a RECEIVE request before the host can transfer the data with a SEND operation. Different from one-sided verbs, the host does not specify the target remote memory address. Instead, the remote host defines the target address in its RECEIVE operation. Another difference is that the remote CPU is actively involved in the data transfer.

A further important category of verbs is *Atomic verbs*. These verbs fall into the category of one-sided verbs and enable multiple host nodes to access the same remote memory address concurrently, while preventing data races at the same time. In RDMA, there are two atomic operations available: remote compare-and-swap (CAS) and remote fetch-and-add (FA). An important difference to READ/WRITE operations is that both atomic operations (CAS and FA) can only modify exactly 8 Bytes on the remote side.

Whether to use one-sided or two-sided verbs strongly depends on the application. While one-sided operations are appealing since they do not involve the remote CPU for being executed, they typically require more complex communication protocols using multiple round-trips between the host and the remote node. On the other hand, two-sided verbs are capable of implementing an RPC-based protocol which requires only two round-trips but involves the remote CPU (potentially heavily) in the execution of the RPC and thus limits the scalability of the application. In this paper, we study these trade-offs for the design of distributed tree-based index structures. A more general analysis of whether to use one-sided or two-sided operations can be found in [17].

2.2 Design Space for RDMA-based Indexes

In this section, we discuss the design space of distributed tree-based index structures for RDMA-based networks. The focus of this paper is on the NAM architecture and thus on the question of how to distribute the index over memory resources of multiple memory servers, as well as how to access the distributed index from compute servers (see Figure 1). However, we believe that the design discussion in this section can also help to understand the design space on how to build distributed tree-based indexes for RDMA in general and thus the findings can be applied to other architectures (e.g., in a shared-nothing architecture). We will discuss some of these other scenarios in more detail in Section 7.

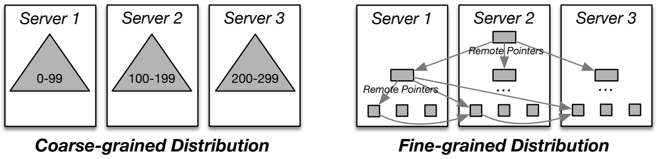


Figure 2: Index Distribution Schemes

In this section, we assume that the tree-based index has a structure similar to a B-$^+$-tree (or more precisely a *B-link* tree [24]); i.e., inner nodes only store separators and leaf nodes store the actual keys of the index. We will describe the details of how we adapted the *B-link* tree in Sections 3 to 5 for our implementation. The index designs discussed in this paper are applicable to primary/secondary as well as clustered/non-clustered tree-based indexes with and without unique keys.

Since tree-based indexes and in particular *B-link* trees are often used as secondary indexes, the following discussions assume a secondary (i.e., non-clustered) index with non-unique keys, where duplicate keys can be stored in the leaf/inner nodes of the index while leaves map secondary keys (called keys further on) to primary keys (called payload further on). The generalization to the other cases (primary index, clustered index) is straightforward.

Index Distribution: In order to distribute the index (inner and leaf nodes) across the memory of multiple machines, two different extreme forms of distribution schemes can be applied — namely a coarse-grained and a fine-grained distribution scheme (see Figure 2):

(1) Coarse-grained Distribution (CG): This scheme applies a classical approach known from shared-nothing architectures. In order to distribute the index over multiple servers, we first apply a partitioning function (either range-based, round-robin, or a hash-based) to the keys that are being indexed and thus decide on which server a key and its payload should be stored. Once all keys and the payload are assigned to servers, we can build a separate tree-based index on each of the memory servers (i.e., we are co-locating inner and leaf nodes on the same server).

(2) Fine-grained Distribution (FG): In the fine-grained distribution scheme, we implement the other extreme case, where we do not partition the index at all but instead build a global index over all keys and distribute the index nodes (i.e., leaf and inner nodes) on a per node-basis over the memory of all machines in a round-robin manner level by level. In order

to connect the index nodes, we use remote memory pointers that encode not only the memory address but also the storage location (i.e., the memory server) which holds the remote node. Details about the implementation of remote memory pointers will be discussed later in Section 4.

While the coarse-grained scheme is the dominant solution to distribute indexes (and data) in slow networks when the network bandwidth is a major limiting factor, the fine-grained scheme is an interesting alternative for fast RDMA-capable networks since it can farm out index requests across all servers and thus lead to better load balancing. This is particularly interesting if local and remote memory bandwidth converges which is already true for the most recent InfiniBand standard HDR 4× that can provide approx. $50GB/s$ for remote memory accesses per dual-port RDMA network card.

At the end of this section, we provide a formal analysis of the scalability of both distribution schemes.

RDMA-based Accesses: As explained in Section 2.1, RDMA provides two different classes of operations to access remote memory, called one-sided and two-sided. In the following, we discuss how one- and two-sided RDMA operations can be used to implement index access methods. A more general discussion about the trade-offs of one- and two-sided can be found in [17].

If we use one-sided operations to access an index from a remote host, each node of a tree-based index typically needs to be accessed independently since RDMA READ/WRITEs can only access one remote memory location at a time. Thus, a lookup of a key in a leaf needs one RDMA READ operation for each index node from the index root down to the leaf level following the remote pointers. Range queries additionally need to traverse the (linked) leaf level and need one additional RDMA READ operation for each scanned leaf.

Implementing insert operations is even more complex since we also need to take care of concurrency control (e.g., using RDMA atomics). In addition, to concurrency control operations, inserts need at least two full index passes from the root to the leaves and back. In the top-down pass, we access every index node from the root to the leaf using RDMA READs and in the bottom-up pass we then need to install at most two pages using RDMA WRITEs for every level (in case splits happen) plus potentially one additional RDMA WRITE for installing a new root node.

In case we use two-sided RDMA operations for index accesses, we can leverage the fact that we can implement an RPC protocol using an approach similar to [21] where the remote CPU is involved to apply the index operations. An RPC call from one server to another can be implemented using a pair of SEND/RECEIVE operations: one pair for sending the request from host to remote server, and one pair for the response (e.g., which contains the result in case of an index lookup). Thus, two-sided operations seem to be more efficient for implementing remote index accesses.

Description	**Symbol**	**Example**
# of Memory Servers	S	4
Bandwidth per Memory Server (GB/s)	BW	50GB/s
Page Size of Index Nodes (in Bytes)	P	1024B
Data Size (# of tuples)	D	100M
Key Size (in Bytes) - same as Value/Pointer Size -	K	8B
Fanout (per index node)	M=P/(3·K)	42
Leaves (# of nodes)	L=D/M	approx. 2.3M
Max. index height (FG, Unif./Skew)	H_{FG}=$\log_M$(L)	4
Max. index height (CG, Unif.)	H^{U}_{CG}=$\log_M$(L/S)	4
Max. index height (CG, Skew)	H^{S}_{CG}=$\log_M$(L)	4

Table 1: Overview of Symbols

	Fine-grained	**Coarse-grained (2-sided)**	
	(1-sided)	**Range**	**Hash**
Step (1): Avail. BW:			
Total BW Uniform	S·BW	S·BW	S·BW
Total BW Skew	S·BW	1·BW	1·BW
Step (2): BW per Q			
Point (Unif., sel=$1/L$)	$\mathrm{H}_{FG}\cdot\mathrm{P}$	$\mathrm{H}^{U}_{CG}\cdot\mathrm{P}$	$\mathrm{H}^{U}_{CG}\cdot\mathrm{P}$
Point (Skew, sel=z/L)	$\mathrm{H}_{FG}\cdot\mathrm{P}+\mathrm{z}\cdot\mathrm{P}$	$\mathrm{H}^{S}_{CG}\cdot\mathrm{P}+\mathrm{z}\cdot\mathrm{P}$	$\mathrm{H}^{S}_{CG}\cdot\mathrm{P}+\mathrm{z}\cdot\mathrm{P}$
Range (Unif., sel=s)	$\mathrm{H}_{FG}\cdot\mathrm{P}+\mathrm{s}\cdot\mathrm{L}\cdot\mathrm{P}$	$\mathrm{H}^{U}_{CG}\cdot\mathrm{P}+\mathrm{s}\cdot\mathrm{L}\cdot\mathrm{P}$	$\mathrm{H}^{U}_{CG}\cdot\mathrm{P}\cdot\mathrm{S}+\mathrm{s}\cdot\mathrm{L}\cdot\mathrm{P}$
Range (Skew, sel= s_z)	$\mathrm{H}_{FG}\cdot\mathrm{P}+s_z\cdot\mathrm{L}\cdot\mathrm{P}$	$\mathrm{H}^{S}_{CG}\cdot\mathrm{P}+s_z\cdot\mathrm{L}\cdot\mathrm{P}$	$\mathrm{H}^{S}_{CG}\cdot\mathrm{P}\cdot\mathrm{S}+s_z\cdot\mathrm{L}\cdot\mathrm{P}$
Step (3): Max. Q/sec			
Max. Throughput	Avail. BW / BW requirement per Q		

Table 2: Scalability Analysis (Theoretical)

However, again when assuming that the local memory bandwidth and network bandwidth are equal, one-sided operations are not worse than two-sided operations w.r.t. the bandwidth and thus throughput. Nevertheless, latency might increase. On the other hand, using one-sided operations do not involve the remote CPU at all. This is especially beneficial for skewed workloads in high-load scenarios leading to higher throughput and lower latency of index accesses, as we will show in our experiments in Section 6.

2.3 Scalability Analysis

In this section, we now formally analyze the theoretical maximal throughput for the different index design variants introduced before. The basic idea of the scalability analysis is to compute the theoretical maximal throughput (i.e., number of index accesses per second) based on the number of available servers that hold index data (i.e., memory servers in the NAM architecture). The theoretical maximal throughput will be computed by the total aggregated (remote) memory bandwidth available that can be provided by all memory servers divided by the bandwidth requirements for each index access (i.e., for each query Q).

In the analysis we did not consider latency which is definitively higher for FG because multiple network round-trips are needed for each query. In our evaluation in Section 6, we discuss the latency of the different index designs and show that under high-load the FG scheme outperforms the CG scheme in terms of latency while being slightly higher for normal load.

Assumptions: For analyzing maximal throughput for S memory servers, we assume that memory bandwidth and network bandwidth are equal and thus we do not differentiate between both. This assumption is realistic as shown in [5]. Since [5] was published, a new generation of InfiniBand hardware (HDR 4×) appeared. With this hardware, the remote memory bandwidth when using two network cards (or one with a dual-port interface) will give us around $50GB/s$, which is close to what we can expect today from the local memory bus of one CPU socket with 4 memory channels.

In our scalability analysis, we further consider different combinations of how the index is distributed across memory servers (fine-grained/FG vs. coarse-grained/CG), as well as different workload characteristics of how index accesses are distributed (uniform vs. skewed). Since, we cannot analyze all possible alternatives we make the following restrictions: (a) We do not differentiate between one-sided and two-sided RDMA operations since we assume memory and network bandwidth to be equal as discussed before. Moreover, we also do not include the CPU load in our analysis. To that end, searching an index page from a compute server using a RPC (based on two-sided operations) or a read using one-sided operations is not different anymore w.r.t. required (remote) memory bandwidth and thus the resulting throughput. (b) We only consider read-only index accesses for the theoretical analysis since read accesses dominate OLTP workloads [22] (and OLAP workloads as well). In our experimental evaluation in Section 6, we also include workloads with write accesses which show similar effects as the theoretical analysis for read-only workloads in terms of throughput. (c) For analyzing skewed workloads, we assume attribute-value-skew on the indexed attribute; i.e., one distinct key dominates the distribution of the secondary index. In case of CG-distribution (hash and range) this means that the majority of index entries (i.e., inner and leaf nodes) will end up being stored on a single memory server.

Analysis and Findings: We now discuss the scalability of throughput for different index designs. For the scalability analysis, we use the symbols that we introduce in Table 1. The findings of the analysis are summarized in Table 2. In the following, we explain the idea behind the analysis in a stepwise manner as indicated in the table.

Step (1) in Table 2: First, for different workload distributions we model the effectively available aggregated bandwidth of all memory servers that hold the index. We assume that we have a cluster with S memory servers, each contributing a bandwidth BW to the aggregated bandwidth. The total available aggregated bandwidth for FG-distribution is always $S \cdot BW$. The reason is that even if workload is skewed (i.e., one secondary key dominates the distribution), index accesses will always be farmed out to all memory servers due to the round-robin distribution of index nodes to servers. This is different for CG-distribution (hash and range). In the skewed case, the very same memory server stores most of the index data, thus effectively limiting the bandwidth to only $1 \cdot BW$ in the worst case.

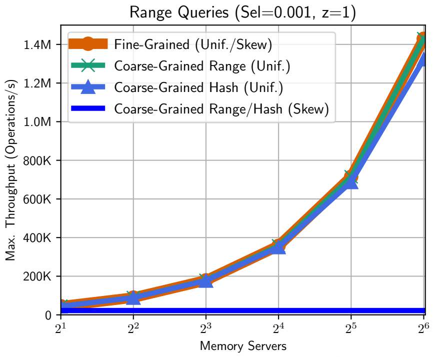


Figure 3: Maximal Throughput (Theoretical)

Step (2), line 2 in Table 2: In a second step, we now consider the (remote) memory bandwidth requirements of an individual index access (i.e., one query) and start with point queries and then continue with range queries.

For uniform distribution, we assume that only one leaf page needs to be read by the point query (i.e., the selectivity is $sel = 1/L$). In both cases (FG and CG), we thus only need to traverse the index height. To that end the memory bandwidth requirement is $H_{FG} \cdot P$ (where P is the size of an index node in bytes). For FG distribution, the index is built over all leaf nodes L and thus the height is $log_M(L)$ where M is the fanout of an index node. The index height for CG (uniform) is only $log_M(L/S)$ since data is first partitioned on S servers. In the CG (skew) case, we assume that the maximal index height under the CG scheme is the same as for the FG scheme. The reason is that most leaf nodes will be stored in one memory server (which also increases the index height). Moreover, we assume a read-amplification of z for skewed workloads; i.e., z leaf pages need to be retrieved instead of 1 only (i.e., the selectivity is $sel = z/L$) resulting in an additional memory bandwidth requirement of $z \cdot P$ for a point query.

For range queries, we assume that under uniform workload a fraction of s leaf pages needs to be retrieved (i.e., the selectivity is $sel = s$). For skewed workloads this will again be amplified by a factor of z (i.e., the selectivity is $sel = z \cdot s = s_z$). Moreover, in the skewed case, we again assume that CG has the same maximal height as FG since most data will be stored in one memory server. Additionally, for the hash-based scheme queries must be send to all S memory servers resulting in S index traversals from root to leaves.

Step (3), line 3 in Table 2: Based on the results in step (1) and (2), we can now derive the theoretical maximal throughput for S memory servers by dividing the available aggregated

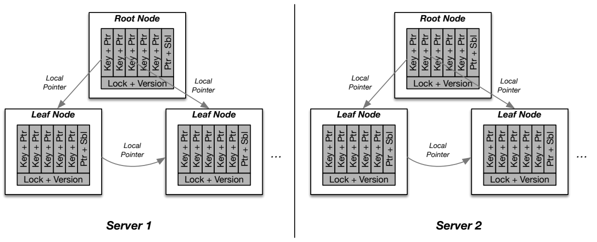


Figure 4: Design 1 - Coarse-Grained Index

memory bandwidth (step 1) by the bandwidth requirements for the different queries (step 2).

Example Results: Figure 3 plots the results of our analysis for range queries (for a uniform and a skewed workload). Point queries show a similar trend; thus we do not show their plot. For Figure 3 plot, we use the example values provided in Table 1 (rightmost column), however, additionally varying the available memory servers between $S = 2 \ldots 64$. We choose a selectivity of $s = 0.001$ and skew amplification of $z = 10$.

We can see that all index designs scale well for uniform workloads. The reason why the CG scheme (uniform) scales slightly worse for hash-partitioning than for range-partitioning is that queries for hash-partitioning need to be sent to all memory servers since all servers might hold relevant index entries for range queries resulting in the fact that the indexes on all S machines need to be traversed. This is different for skewed workloads. Here the FG scheme still shows the same scalability as for a uniform workload while the CG scheme stagnates with increasing memory servers. The main reason is that the total available bandwidth is limited to only BW under skew and is independent from the number of memory servers available. To that end, we can see that in terms of throughput the FG scheme is the only scheme which achieves a throughput which scales with the available memory servers independent of the workload. Next, we present the details of our index implementations using a coarse-grained (CG), a fine-grained (FG), and a hybrid distribution scheme that mixes CG and FG. For each of the index implementations we also discuss which RDMA operations are being used for accessing the index from a remote machine (i.e., from a compute server).

3 DESIGN 1: COARSE-GRAIN/TWO-SIDED

In this section, we discuss our first tree-based index structure design which can be distributed over the memory of multiple servers and accessed by clients via RDMA (e.g., compute servers in the NAM architecture). First, we discuss the details of the distributed index structure itself. Afterwards, we elaborate on how this index structure can be efficiently accessed using RDMA operations.

3.1 Index Structure

The first index structure leverages a coarse-grained distribution scheme as shown in Figure 2. The basic idea of the coarse-grained index distribution scheme is to partition the key space by using a traditional partitioning scheme (either hash- or range-based), between different memory servers. Afterwards, each memory server individually builds a tree-based index for its assigned keys.

```
operation lookup(key, node, parent, parentVersion) {
  version = readLockOrRestart(node);
  if(parent!= null)
    readUnlockOrRestart(parent, versionParent)

  if(isLeaf(node))
     value= getLeafValue(node)
     checkOrRestart(node, version)
     return value
  else
    nextNode = node.findChildInNodeOrSingling(key)
    checkOrRestart(node, version)
    return lookup(key, nextNode, node, version)
}

operation insert(key, value, node, parent, parentVersion){
  version = readLockOrRestart(node);
  if(parent!= null)
    readUnlockOrRestart(parent, versionParent)

  if(isLeaf(node))
    upgradeToWriteLockOrRestart(node, version)
    splitKey = node.insert(key, value)
    writeUnlock(node)
    return splitKey
  else
    nextNode = node.findChild(key)
    splitKey = insert(key, value, nextNode, node,version)
    if(splitKey!=NULL)
      upgradeToWriteLockOrRestart(node, version)
      parentSplitKey = node.insert(key, value)
      writeUnlock(node)
      return parentSplitKey
    return NULL
}
```

Listing 1: Operations of a Coarse-Grained Index

The internal index structure in each node is shown in Figure 4 and follows the basic concepts of a *B-link* tree [24]. However, different from the original *B-link* tree, we use real memory pointers instead of page identifiers. More importantly, we additionally introduce an 8-byte field per index node which stores a pair $(version, lock\text{-}bit)$ where the last bit represents a lock-bit. We use this field to implement a concurrency protocol based on optimistic-lock-coupling [25]. Our adaption of optimistic-lock-coupling for RDMA is explained in the next section.

3.2 RDMA-based Accesses

In order to access the index structure, as shown in Figure 4, from a remote host, we use an RPC-based protocol that uses two-sided RDMA operations. This design thus follows a more traditional paradigm where operations are shipped to the data – similar to how database operations are executed in a shared-nothing architecture. Other index designs which use one-sided operations or a hybrid access protocol that mixes one-/two-sided operations are explained in Sections 4 and 5 respectively.

Our RPC implementation for this index design is using RDMA SEND/RECEIVE similar to the RPC implementation of [21]. In contrast to [21], we are not using unreliable datagrams (UD) to implement the RPC but reliable connections (RC). Typically the overall throughput of index operations is limited by either the CPU or the memory bandwidth as we will show in our experiments; rather than by the number of RDMA operations that the network card can execute (which was the main motivation of using UD in [21]). Furthermore, to better scale-out with the number of clients, we are using shared receive queues (SRQs) to handle the RDMA RECEIVE operations on the memory servers. SRQs allow all incoming clients to be mapped to a fixed number of receive queues, instead of using one receive queue per client [41].

In the following, we mainly focus on how the remote procedures are executed on memory servers which store the part of the index being requested.

Index Lookups: If an incoming RPC represents an index lookup (i.e., a point- or a range-query), a thread which handles the RPC in a memory server traverses the index using a concurrency protocol based on optimistic-lock-coupling [25] but adapted for our tree-based index structure.

Index Updates: Furthermore, we also support index inserts and deletes as on RPCs. In the following, we first discuss inserts and then delete operations. Similar to [25], the thread which handles the insertion RPC, does not acquire any lock in the top-down pass from root to leaf nodes. Instead it acquires the first lock on the leaf level using a local compare-and-swap (CAS). If a leaf needs to be split, due to an insert, the locks are propagated to the parent nodes. Delete operations are implemented by setting a delete bit per index entry instead of removing the key. For removing deleted entries we use an epoch-based garbage collection scheme which runs on each memory server in a NAM architecture and is responsible for removing and re-balancing the index in regular intervals.

The code for the two operations `lookup` (point-query) and `insert` that are executed locally on a memory server is shown in Listing 1. The concurrency scheme of the coarse-grained index relies on the same methods as [25]. Range-queries work similar and only need to traverse the leaf level additionally. The helper methods used in the code of Listing 1 is shown in Appendix A.1. In order to implement the lookup operation of Listing 1, `readLockOrRestart` is used which implements a spinlock on the lock-bit to enter a node. Furthermore, after scanning the content of a node, the lookup operation calls `checkOrRestart` which uses the full version information (including the lock-bit) to check whether a concurrent modification has happened while searching the node. The insert operation of Listing 1 additionally uses a compare-and-swap operation in its `upgradeToWriteLockOrRestart` operation to set the lock-bit before modifying a node and insert a new key. For releasing the lock in `writeUnlock` a fetch-and-add operation is used to atomically reset the lock-bit and carry the bit over to increase the version counter.

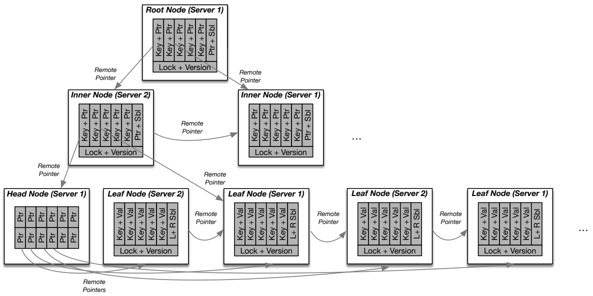


Figure 5: Design 2 - Fine-Grained Index

4 DESIGN 2: FINE-GRAIN/ONE-SIDED

In this section, we discuss our second design of a tree-based index structure.

4.1 Index Structure

The basic idea of the fine-grained index is that the index is distributed on a per-node basis to different memory servers in a round robin fashion as discussed in Section 2. An example index structure is shown in Figure 5.

As in the first design, in addition to the keys and pointers each index node stores an 8-byte field with ($version$, $lock\text{-}bit$) at the beginning of each node. However, different from the first design, pointers are implemented as so called *remote pointers*. More precisely, a remote pointer is a 8-byte field which stores (`nullbit, node-ID, offset`). The `nullbit` indicates whether a remote pointer is a NULL-pointer or not and the `node-ID` encodes the address of the remote memory server (using 7 Bit). The remaining 7 Byte encode an `offset` into the remote memory that can be accessed via RDMA.

Furthermore, we introduce an optimization called head nodes on the leaf level. The optimization will be discussed at the end of this section.

4.2 RDMA-based Accesses

In order to access the index structure shown in Figure 5 from a remote host, we use an RDMA-based protocol that is based on one-sided operations. For the fine-grained distribution scheme we anyway need to access each index node separately (inner and leaf node) and thus one-sided operations are a good fit for the FG distribution scheme. Similar to the index design in Section 3, we use a protocol that is based on optimistic-lock-coupling. Yet, all operations are implemented using one-sided RDMA primitives.

Index Lookups: The intention of the lookup-operation for point-queries is that it can be executed by a compute server to access a remote memory server(s) which store the index node(s). The code for the `remote_lookup` operation which implements a point-query is shown in Listing 2. Range-queries work similar and only need to traverse the leaf level additionally. The code of the helper methods is shown in Listing 4 in Appendix A.1.

The main difference to the lookup protocol of Section 3 is that the `remote_lookup` first copies the accessed node (inner or leaf) with an `RDMA_READ` to the memory of the client. Afterwards, the lookup operation checks on its local copy if the lock is not set (i.e., it is set to 0) and fetches a new copy if the lock is set by implementing a remote spinlock.

An interesting observation is that different from the original protocol in [25], we do not need the version of the node after searching the node. Version checking in the coarse-grained scheme (as shown in line 8 and 12 in Listing 1 for the coarse-grained scheme) is used since a reader may see an intermediate inconsistent state of an index node. In the fine-grained scheme, however, the client holds a local consistent copy which cannot be modified by other clients, hence version checking is not needed. Furthermore, we also do not check the version of the parent again once we traversed down to the next level (as shown in line 4 in Listing 1 for the coarse-grained scheme). Therefore, a concurrent split on the current level might not be detected. However, after a split the first node is written in-place (as discussed below). Thus, we can use the fact that we implement a *B-link* tree and continue the search with the sibling if the search key is not found in the current node.

Finally, a last modification, when using a one-sided protocol, is that compute servers need to know the remote pointer for the root node. This can be implemented as part of a catalog service that is anyway used during query compilation and optimization to access the metadata of the database.

Index Updates: As before, we also support inserts and deletes, which will be discussed in the following.

Inserts are more complex than lookups, since they modify the index structure. Similar to the `remote_lookup` operation the compute server first fetches a local copy and checks if no lock was set. To that end, version checking after traversing from one leaf level to the next is not needed anymore since clients hold a copy of an index node. However, since clients only hold a local copy of the index node, the `remote_upgradeToWriteLockOrRestart` operation uses an `RDMA_CAS` operation for setting the lock-bit on the remote memory server. Moreover, `remote_writeUnlock` resets the lock-bit remotely with `RDMA_FETCH_AND_ADD`. This method additionally installs the modified version of the node on the remote side using an `RDMA_WRITE` as shown in Listing 4 in Appendix A.1. In case a node has to be split (i.e., the *splitkey* is not *NULL*), the `remote_writeUnlock` method additionally writes the second node resulting from the split.

```
operation remoteLookup(key, remNodePtr) {
  node = remote_read(remNodePtr)
  remote_readLockOrRestart(node, remNodePtr)

  if(isLeaf(node))
     value= getLeafValueFromNodeOrSiblings(node)
     return value
  else
    nextNodePtr = node.findChildInNodeOrSiblings(key)
    return remoteLookup(key, nextNodePtr)
}

operation remoteInsert(key, value, remNodePtr){
  node = remote_read(remNodePtr)
  version = remote_readLockOrRestart(node, remNodePtr)

  if(isLeaf(node))
    remote_upgradeToWriteLockOrRestart(node, remNodePtr,version)
    splitKey = node.insert(key, value)
    remote_writeUnlock(node, remNodePtr)
    return splitKey
  else
    nextNodePtr = node.findChildInNodeOrSiblings(key)
    splitKey = remoteInsert(key, value, nextNodePtr)
    if(splitKey!=NULL)
      remote_upgradeToWriteLockOrRestart(node, remNodePtr,version)
      parentSplitKey = node.insert(key, value)
      remote_writeUnlock(node, remNodePtr)
      return parentSplitKey
    return NULL
}
```

Listing 2: Operations of a Fine-Grained Index

Finally, delete operations (no shown in Listing 2) are again implemented by setting a delete bit using a similar protocol as inserts, which modifies a local copy of the page and then writes the node back to the memory server. Moreover, we use an epoch based garbage collection scheme similar to Section 3, although the garbage collection thread is run by a compute server globally for the complete index. The reason is that deletions also need to lock the index nodes (same as writes). In order to implemented these locks, we have to use the same one-sided protocol as for potential concurrent writes which relies on RDMA-based atomics. The reason why we cannot run garbage collection as a local thread on a memory server is that atomicity cannot be guaranteed if remote and local atomic operations would both be used concurrently on the same memory addresses [10].

4.3 Optimization of Index Structure

In addition to the basic index design, we introduce so-called head nodes in the leaf level. Head nodes are additional leaf nodes (with no actual index data) that are installed after every n-th real leaf node. The idea of the head nodes is that they redundantly store remote pointers to all $n-1$ following leaf nodes (i.e., the pointers between leaf nodes are still kept). That way, a compute node which reads a head node during a leaf level scan (which is necessary for a range query) can use the remote pointers to prefetch leaves. This technique is based on selectively signaled RDMA READs as already presented in [39]. Prefetching reduces the network latency by masking network transfer with computation.

One difficulty that arises when using head nodes is that they need to be updated after a leaf node splits. Since head nodes are only an optimization and we keep the actual sibling pointers additionally in each leaf node, we do this similar to garbage collection in an epoch-based manner using an

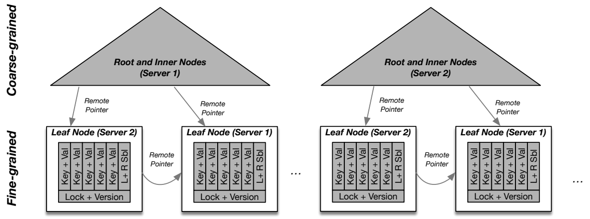


Figure 6: Design 3 - Hybrid Index

additional thread that scans through the leaf nodes of all memory servers and installs new head nodes (resp. removes the old head nodes). A compute server can detect outdated head nodes during traversing the leaf level; i.e., if a sibling pointer in a leaf node is pointing to a leaf node whose remote pointer was not in a head node (i.e., it was not prefetched), then the compute server which traverses the leaf level simply needs to execute an additional remote read for this pointer. This will cause some increase in latency but the scan of the leaf level will still be correct.

5 DESIGN 3: HYBRID SCHEME

This index design is a hybrid scheme combining the two schemes discussed in Section 3 and Section 4.

5.1 Index Structure

For distributing the index, as shown in Figure 6 we use a coarse-grained scheme to partition the upper levels of the index (inner and root node) while we use a fine-grained scheme for the nodes on the leaf level. The intuition is that we combine the best of both designs; i.e., getting low latency by using an RPC-based index traversal and still being able to leverage the aggregated bandwidth of all memory servers by distributing leaves in a fine-grained manner.

That way, even if attribute-value-skew on index key occurs, leaf nodes are still distributed uniformly to all memory servers. Additionally, the leaf level in this index structure can leverage head nodes, similar to the design in Section 4, to enable prefetching for range queries.

5.2 RDMA-based Accesses

For accessing the index, we also use a hybrid scheme of one-sided and two-sided RDMA operations.

Index Lookups: The basic idea is that we (as already discussed before) traverse the upper levels of the index using RPCs that are implemented using two-sided operations. However, instead of returning the actual data, the RPC only returns the remote pointer to the leaf node. Afterwards, in case of a lookup (i.e., point- and range-queries), the compute server fetches the leaf nodes using one-sided RDMA READs.

Index Updates: In case of an insertion, we again use an RPC that traverses the index and returns a remote pointer to a leaf page where the new key should be inserted to. For actually installing the key, the compute server uses the remote pointer and the one-sided protocol from Section 4 to install the new key to the leaf level.

In case a new leaf node has to be inserted (due to a split operation), the compute node will issue an additional RPC over two-sided RDMA to the memory server indicating that a new leaf node has been inserted (using the start key and the new remote pointer as arguments). The memory server will then use the second part of insertion protocol from Section 3 to install the new key into the upper levels of the index.

Workload	Point Queries	Range Queries (sel=s)	Inserts
A	100%		
B		100%	
C	95%		5%
D	50%		50%

Table 3: Workloads of our Evaluation

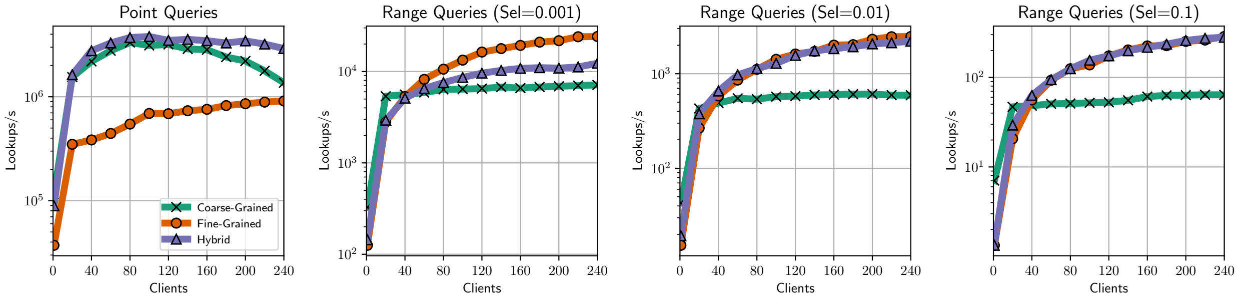
Finally, deletes are handled again by an epoch-based garbage collection. In this scheme, a global garbage collection thread is again executed on a compute server handles deletes for all the leaf nodes while local garbage collection threads on memory servers handle the upper levels. There is no need to synchronize the local garbage collection threads on memory servers with the global garbage collector for leaves since the delete operation already takes care of setting the delete bit in a consistent manner.

6 EXPERIMENTAL EVALUATION

The goal of our experiments is to analyze the different index designs (CG/2-Sided, FG/1-sided, Hybrid) presented in Sections 3 to 5.

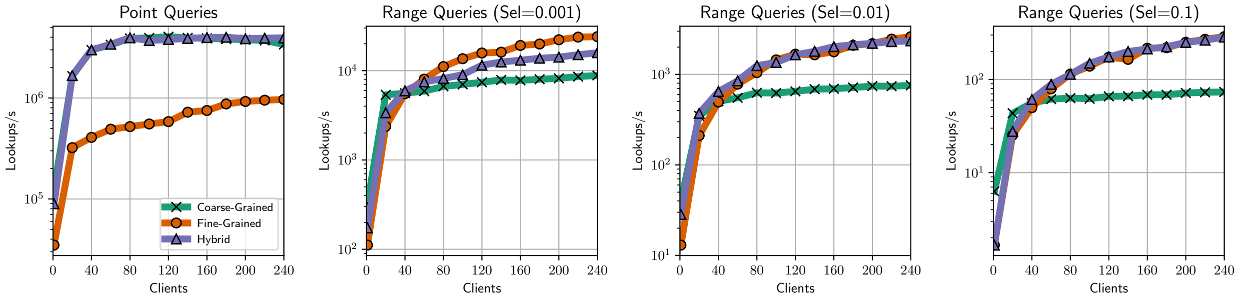
Workloads: We chose the Yahoo! Cloud Serving Benchmark (YCSB) to mimic typical OLTP and OLAP index workloads. Since the original version of the YCSB workload [12] does not cover all relevant cases for tree-based index structures we implemented a modified version. For example, the original version of YCSB only supports queries for short ranges but does not explicitly support different selectivities (low/ high) for query ranges. Table 3 summarizes the workloads of our modified version of the YCSB benchmark: *Workload A* models a read-only workload with 100% point queries while *Workload B* represents read-only workload with range queries where the selectivity can be configured to different values. In our experiment we used $sel = 0.001$ (0.1%), $sel = 0.01$ (1%) and $sel = 0.1$ (10%) to model different cases from low selectivity to high selectivity.

Moreover, the original YCSB only supports a skewed access pattern of queries by using a Zipfian distribution for the requested keys. However, for evaluating a tree-based index structure we also want to modify the skewness of the data itself (to introduce attribute-value skew), as it has a significant impact on the index performance as already discussed in Section 2. Therefore, we generated data sets with monotonically increasing integer keys and values (each 4-Byte) with different sizes: $10M$, $100M$, and $1B$. In order to simulate



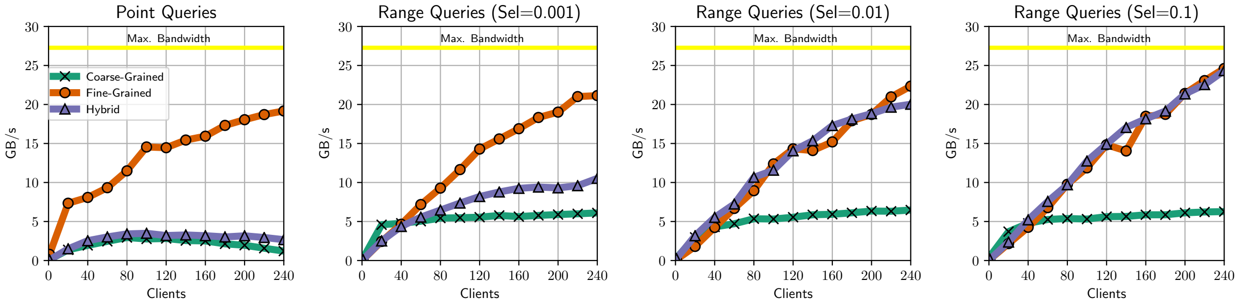
(a) Point Query (b) Range Query (sel=0.001) (c) Range Query (sel=0.01) (d) Range Query (sel=0.1)

Figure 7: Throughput for Workloads A and B (Skewed Data, Size 100M)



(a) Point Query (b) Range Query (sel=0.001) (c) Range Query (sel=0.01) (d) Range Query (sel=0.1)

Figure 8: Throughput for Workloads A and B (Uniform Data, Size 100M)



(a) Point Query (b) Range Query (sel=0.001) (c) Range Query (sel=0.01) (d) Range Query (sel=0.1)

Figure 9: Network Utilization for Workloads A and B (Skewed Data, Size 100M)

non-unique data with attribute-value skew, we use the same data with unique keys/values and assign the data based on key ranges to servers to enforce a skewed distribution; e.g., if we use two servers, we could assign 80% to one server and 20% of the data to the other server.

Setup: For executing all the experiments, we used a cluster with 8 machines featuring a dual-port Mellanox Connect-IB card connected to a single InfiniBand FDR 4× switch. Each machine has two Intel Xeon E5-2660 v2 processors (each with 10 cores) and 256GB RAM. The machines run Ubuntu 14.01 Server Edition (kernel 3.13.0-35-generic) as their operating system and use the Mellanox OFED 3.4.1 driver for the network. All three index designs where implemented using C++ 11 and compiled using GCC 4.8.5.

6.1 Exp.1: Throughput

In our first experiment, we analyze the throughput of the different indexing variants presented in Section 3 to Section 5. The main goal of this experiment is to show the behaviour of the different index designs under a varying load for workloads with and without skew. As discussed in Section 2 a major difference of the individual index designs is how data is distributed and accessed via RDMA. This determines how efficiently an indexing variant can leverage the total aggregated bandwidth of all memory servers in a NAM architecture.

In order to analyze the efficiency of the different indexing strategies and model the throughput behavior, we deployed a NAM cluster with 4 memory servers (on 2 physical machines) and used $1-6$ compute servers (on $1-6$ physical machines) each running 40 compute threads to access the index. We deployed 2 memory servers on each physical machine to exploit the fact that our InfiniBand cards support two ports; i.e., each memory server was thus using its own dedicated port on the networking cards.

In this experiment, we first focus on the read-only workloads A (point queries) and B (range queries). Workloads C and D, which include insertions, will be used in Section 6.3. For workload B, we ran different variants each having a different selectivity ($sel = 0.001$, $sel = 0.01$, and $sel = 0.1$). We use $100M$ key/value pairs throughout this experiment.

For running the workloads under low- and high-load scenarios, we ran each of these workloads with a different number of compute servers starting with one server that hosts 20 compute threads (called clients in this experiment). Each of the client threads executes index lookups (point and range queries) in a closed loop (i.e., it waits for a lookup to finish before executing the next lookup) and spreads lookups uniformly at random over the complete key space.

In order to model attribute-value skew in this experiment, we use range partitioning for the coarse-grained index to assign 80% of the key/value pairs to the first memory server, 12% to the second , 5% to the third, and 3% to the last memory server. Consequently, 80% of the lookups need to be sent to the first server since requests are spread uniformly across the key space. For the hybrid index, we use a similar scheme and only shuffle the leaf nodes in a round robin manner using fine-grained index distribution.

In the following, we first show the throughput results for uniform and skewed data and then discuss the network utilization of the different schemes (coarse-grained, fine-grained, and hybrid).

Discussion of Throughput: The throughput for all schemes with skewed and uniform data are shown in Figure 7 and 8. The x-axis shows the number of clients used and the y-axis the resulting aggregated throughput.

A first interesting observation is that the hybrid approach, combining ideas of coarse- and fine-grained, performs in the most robust manner not only for point and range queries but also for different data distributions (uniform and skewed) as shown in Figure 7 and Figure 8. The reason is that it combines the best of both other schemes: getting low latency by using a RPC-based index traversal as in the coarse-grained scheme, and still being able to leverage the aggregated bandwidth of all memory servers by distributing leaves as in the fine-grained scheme. As a result the hybrid scheme has the highest throughput in (almost) all cases. Only for lower loads (i.e., ≤ 20 clients), the coarse-grained outperforms the hybrid scheme minimally. The reason is that the coarse-grained scheme is slightly more communication efficient since it gets the qualifying data of an index lookup directly in the RPC response. In contrast, the hybrid scheme only gets a remote pointer and then additionally need to read the leaf data using RDMA READ operations.

For scenarios with moderate and higher loads (i.e., > 20 clients), the hybrid scheme clearly outperforms the coarse-grained scheme. The reason for the stagnation of coarse-grained is that the memory servers becomes CPU bound with more than 20 clients, which in the case of coarse-grained becomes the bottleneck. One could argue that four memory servers (executed on two physical machines each having 20 cores) should allow coarse-grained to scale to 40 clients. However, the RDMA network card is attached to one socket only (while each machine has two sockets). To that end, the second memory server on each machine needs to cross the QPI link for every index lookup leading to less throughput for this server (i.e., the experiments show the point when the first memory server on each machine becomes saturated). Furthermore, we can see that the throughput of the coarse-grained scheme for skewed data (Figure 7) is approx. 20% below uniform data (Figure 8) and even declines under a high load for point queries (Figure 7(a)).

Finally, another interesting observation is that the fine-grained approach performs almost as good as the hybrid scheme except for point queries in which the fine-grained scheme achieves a much lower throughput. After all, the fine-grained scheme has a much lower network efficiency (as discussed next) for point queries; i.e., each index lookup needs to transfer multiple index pages over the network to traverse the index. This problem is mitigated in the hybrid scheme since it uses a two-sided RDMA implementation to traverse the index using an RPC to the memory server.

Discussion of Network Utilization: In the following, we discuss the effects that we see in the network utilization as shown in Figure 9. One effect that becomes visible, is that fine-grained scheme, which relies purely on one-sided RDMA operations, is less network efficient for point queries as the hybrid and the coarse-grained scheme. For point queries, the coarse-grained scheme, which uses two-sided RDMA, only needs to transfer one key for the request and value for the response between compute and memory servers to implement the RPC. A similar observation holds for the hybrid scheme, which only needs one additional RDMA operation (i.e., a READ) to fetch the result. In contrast to the coarse-grained and hybrid, the fine-grained scheme needs multiple round-trips to traverse the index (i.e., read n index pages). This results in a higher network load and also translates into lower throughput.

For range queries, instead, the network communication between compute and memory servers is dominated by the data from the leaf level that needs to be transferred. For example, in the case of $s = 0.001$, fine- and coarse-grained need to transfer approx. 1600 pages in our experiments for a data size of $100M$ from the leaf level between compute and memory servers. The index pages that need to be transferred for the fine-grained scheme to traverse the index, thus do not add a noticeable overhead. For example, in our experiments with data sizes of up to $100M$, the index height is only 4; i.e., only 4 pages in addition to the 1600 pages need to be transferred in the fine-grained scheme compared to coarse-grained and hybrid scheme.

Important to note is that for fine-grained as well as for the hybrid scheme, the leaf level is distributed across all memory servers on a per page basis. Therefore, both schemes utilize the remote memory bandwidth of all memory servers when executing range queries (as shown in Figure 9). This allows the fine-grained as well as the hybrid scheme to achieve the same throughput for range queries under skew and uniform data, while the coarse-grained scheme is limited by the bandwidth of one memory server.

6.2 Exp.2: Scalability

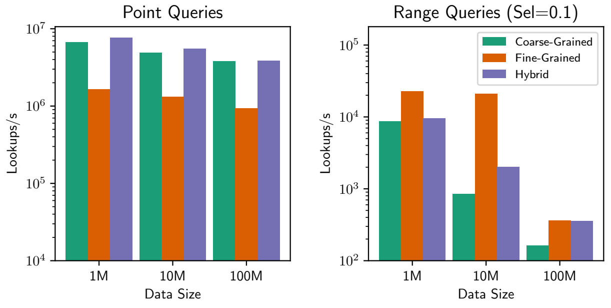
Exp.2a: Varying Data Size. The goal of this experiment is to demonstrate how the operations per second change with different data sizes while maintaining the same number of memory servers. We again analyze all three different index designs using the same setup as before with 4 memory servers on 2 physical machines. Moreover, we used 6 compute servers with a total of 240 clients to show the effect of high load and a uniform data distribution which better leverages all available resources in the different designs.

The results of the experiment can be seen in Figure 10 for point queries and range queries with high selectivities of 10% to show the extreme cases. For point queries, we see that all indexing approaches behave similarly for the different data sizes; i.e., with increasing data size the throughput only drops minimally in all cases. However, for range queries we can see a significant drop for fine-grained and hybrid indexes when data size increases. The reason is that both approaches become network-bound for range queries with a selectivity of $sel = 0.1$.

As we show in our next experiment, adding more memory servers helps to further increase throughput. This underlines the advantage of the NAM architecture of being able to scale the different resources (compute and memory servers) individually if one becomes a bottleneck.

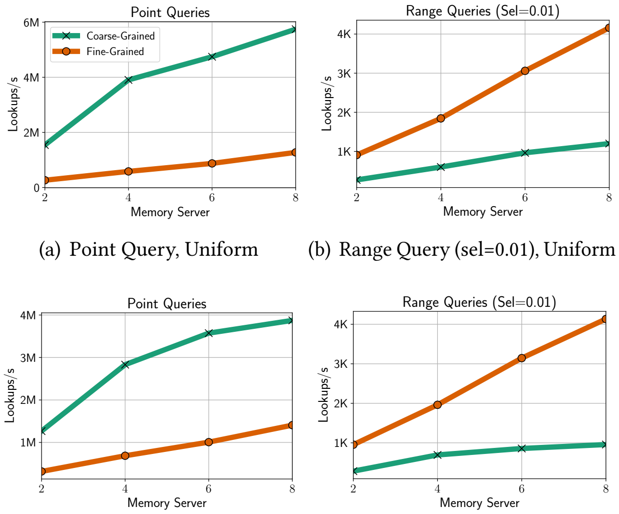
Exp.2b: Varying # of Memory Servers. In this experiment, we analyze the throughput of the coarse-grained and fine-grained indexing scheme when using a different number of memory servers. We do not show the hybrid scheme since the results are, as in the previous experiments, very similar to coarse-grained for point queries and to fine-grained indexes for range queries.

For the setup, we use only 3 machines for compute clients with a total number of 120 clients (i.e. 40 per compute server). Moreover, we used $1 - 8$ memory servers to distribute the index; where two memory servers always shared the same physical machine as before. Furthermore, as workloads we again use all the different queries (point and range queries with $sel = 0.01$) with different data distribution for a data size of $100M$ index entries.



(a) Point Query (b) Range Query (sel=0.1)

Figure 10: Varying Data Size for Workloads A and B (Uniform Data, 240 Clients)



(a) Point Query, Uniform (b) Range Query (sel=0.01), Uniform

(c) Point Query, Skew (d) Range Query (sel=0.01), Skew

Figure 11: Varying # of Memory Servers for Workloads A and B (Size 100M, 120 Clients)

Figure 11 shows the results. The x-axis shows the number of memory servers used for each run and the y-axis the resulting throughput aggregated over all clients. An interesting result of this experiment is that the fine-grained approach can make use of all memory servers for all workloads while the coarse-grained index only can benefit from an increased number of memory servers if data distribution is not skewed. Furthermore, the hybrid scheme is also sensitive to skew in point queries since the index access is dominated by the two-sided RPC-based access (which becomes the bottleneck under skew). For range queries with a high selectivity it behaves similar to the one-sided scheme again and efficiently can make use of an increased number of memory servers.

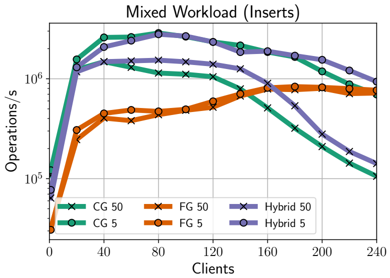


Figure 12: Throughput for Workloads C & D with Inserts (Uniform Data, Size 100M)

6.3 Exp.3: Workloads with Inserts

In the last experiment, we analyze the throughput of the different indexing variants using time workloads C and D which also include insert operations. Workload C is a workload with a low insertion rate (only 5%) whereas workload D has a relatively high insertion rate (50%). The other accesses which are not insertions are comprised only of point queries in those workloads.

In this experiment we use the same experimental setup as in Exp. 1 (Section 6.1) with 4 memory servers and an increasing number of clients. As data set we use the size of $100M$ index entries distributed uniformly across all 4 memory servers. The results of this experiment can be seen in Figure 12. The x-axis shows the number of clients used for each run and the y-axis the resulting throughput of all operations (inserts and lookups) aggregated over all clients.

Again, the hybrid index is the most robust one and clearly outperforms coarse-grained. Furthermore, the hybrid index also dominates the fine-grained index for scenarios with a load with less than 140 clients. For higher loads the fine-grained scheme has a higher throughput while the coarse-gained and hybrid scheme degrade. The main reason is that for the coarse-gained and hybrid scheme, a higher load increases the wait time in the memory servers for spin locks. In consequence, the threads that traverse the index are busy waiting and cannot accept lookups/inserts from other clients. In case of the fine-grained scheme, the clients use remote spin locks, which allow threads in the compute server to progress if they access other nodes of the index.

7 OTHER ARCHITECTURES

In this section, we discuss how the tree-based index alternatives of this paper could be adapted to other architectures than the NAM architecture.

Shared-Nothing Architecture: A classical architecture for distributed in-memory databases is the shared-nothing architecture. In this architecture, data is partitioned across the memory of all nodes, and each node has direct access only to its local memory. Furthermore, indexes are also created locally per partition. The results of this paper can be applied to the shared-nothing architecture in different ways. In the following, we discuss two potential ideas.

First, we could directly use the coarse-grained index design to make indexes that are built locally per partition (i.e., per node) accessible via RDMA also from other nodes. That way indexes could be accessed remotely using RDMA by distributed transactions that not only need to access data on a single node but also need to access data on other nodes. Moreover, transactions that run on the same node where the index resides can leverage locality (i.e., use local memory accesses) and avoid remote memory accesses completely. An additional experiment, which shows the benefits that result from locality in a shared-nothing architecture is shown in an additional experiment in Appendix A.3.

Second, another problem is that indexes often do not fit into the memory of a single node. A recent study [45] shows that the indexes created for typical OLTP workloads can consume up to 55% of the total memory available in a single node in-memory DBMS. This overhead not only limits the amount of space available to store new data but also reduces space for intermediates that can be helpful when processing existing data. Consequently, another idea is to use the hybrid or fine-grained scheme in a shared nothing architecture to leverage the available memory from other nodes (e.g., in a cloud setup). A similar idea has been discussed in [28] where the buffer pool was extended to other machines that have memory available using RDMA.

Shared-Storage Architectures: Shared-storage architectures separating persistent storage from data processing is a preferred architecture for cloud databases since it can provide elasticity and high-availability [1, 7, 40]. Many of these shared-storage based systems aim to push filter operations into the storage layer to reduce data movements. Recent results show that combining Non-volatile memory (NVM) and RDMA facilitate high-performance distributed storage designs [33]. In these designs, our indexing schemes developed for the NAM architecture could also be applied to push filter operations into the RDMA-enabled storage layer.

Many-Core Architectures (Single-Node): Multi-socket, many-core architectures have replaced single-core architectures in the last decade. So far, a typical design has been that the coherency between CPU caches is managed by the hardware. However, these designs have shown to not scale to a large number of cores distributed across multiple sockets. This problem motivated non-cache-coherent (nCC) multi-core architectures where the machines can be partitioned into independent *hardware islands* [11, 19]. One direction to provide software-managed cache coherence is to use RDMA operations to transfer data between the hardware islands [11]. Thus, we believe that our index designs also become relevant when deploying single-node database on future non-cache-coherent architectures.

8 RELATED WORK

Distributed Databases and RDMA:. An important line of work related to this paper are new database designs based on RDMA [3, 15, 16, 21, 30, 44]. Most related to this paper is the work on the NAM architecture [5, 39, 44]. While [44] identified distributed indexes as a challenge, they only discussed them as an afterthought. Furthermore, there exists other database systems that separate storage from compute nodes [6, 9, 26, 40], all of them treated RDMA as an afterthought and none of them discussed index design for RDMA.

Another recent work [30] is similar to the NAM architecture as it also separates storage from compute nodes. The authors discuss indexes for retrieving data from remote storage. However, their assumption is that the index is small enough to be cached completely by a compute node — an assumption which limits the applicability of their proposal. [8] discusses caching of remote memory using RDMA in general. The main insight of the paper is that finding an ideal caching strategy heavily depends on the workload. This is an observation that we also made for tree-based indexes. Our initial results about caching can be found Appendix A.4.

Other systems that focus on RDMA for building distributed database systems are FaRM [15, 16] and FaSST [21]. FaRM exposes the memory of all machines in the cluster as a shared address space. Threads in FaRM can use transactions as an abstraction to allocate, read, write, and free objects in the address space using strict serializability without worrying about the location of objects. In contrast to FaRM, FaSST discusses how remote procedure calls (RPCs) over two-sided RDMA operations can be implemented efficiently. In this paper, we built on these results: Similar to FaRM, we use an abstraction (called remote pointers) to access remote (and local) data in our fine-grained indexing scheme without worrying about the data location. Similar to FaSST, we also leverage RPC calls based on two-sided RDMA operations for implementing our coarse-grained and the hybrid index scheme. However, different from the ideas discussed in FaRM and FaSST, we implemented optimizations targeting tree-based indexes (e.g., using head pages for pre-fetching) as well as different design decisions such as using shared-receive queues to better support scale-out of compute servers connected to a fixed set of memory servers. Furthermore, both — FaRM and FaSST — discuss indexes (typically hash-tables) only as potential applications of their programming model and do not focus on distributed (tree-based) indexes as we do in this paper.

There has also been some work on RDMA-based lock managers [10, 14, 35, 43] which is relevant to this paper to implement concurrency control protocols for distributed databases. However, lock-managers which implement general purpose solutions for coarse-grained concurrency control. In our indexing schemes instead, we developed a concurrency control protocol for fine-grained latching that is based on so called optimistic-lock coupling [25].

Furthermore, many other projects in academia have also targeted RDMA for OLAP workloads, such as distributed join processing [2, 4, 38] or RDMA-based shuffle operations [29]. As opposed to our work these papers discuss RDMA in a traditional shared-nothing architecture only and they also do not consider the redesign of indexes.

Finally, industrial-strength DBMSs have also adopted RDMA. For example, Oracle RAC [37] has RDMA support, including the use of RDMA atomic primitives. Furthermore, SQLServer [28] uses RDMA to extend the buffer pool of a single node instance but does not discuss the effect on distributed databases at all. After all, none of these systems has discussed distributed indexes for RDMA-based architectures.

Distributed Indexes. The design of distributed indexes has not only been discussed in databases [31] but also in the context of information retrieval [13] and web databases [18]. However, none of these directions has particularly focused on the design of distributed indexes for RDMA.

A distributed RDMA-enabled key/value store such as the ones in [20, 23, 27, 32, 34, 36] can also be seen as a distributed index that can be accessed via RDMA. Different from our work, these papers typically focus on put/get for RDMA-based distributed hash tables. [27] additionally leverages programmable NICs to extend the one-sided RDMA primitives with operations that allow clients to add / retrieve new entries from hash-tables in one round-trip instead of multiple ones. However, different from tree-based indexes, distributed hash tables do not support range queries, which are an important class of queries in OLAP and OLTP workloads. To that end, this line of work complements our work and the results can be used as another form of distributed index for point queries. In fact, in [44] the authors used results from this work to build primary clustered indexes.

9 CONCLUSIONS

In this paper we presented distributed tree-based indexes for RDMA. We have discussed different design alternatives regarding the index distribution and the RDMA-based access protocols. While the focus of this paper was on the NAM architecture which separates compute and memory servers, we believe that the discussions and findings can also help to understand the design space also for other distributed architectures in general. Furthermore, there are other important dimensions such as caching to improve the index performance. As mentioned before, we discuss our initial results for caching in Appendix A.4. However, studying caching in detail is beyond the scope of this paper and represents an interesting avenue of future work.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG) under grants BI2011/1 and gifts from Mellanox.

REFERENCES

[1] Scaling out with Azure SQL Database. https://azure.microsoft.com/en-us/documentation/articles/sql-database-elastic-scale-introduction/.

[2] C. Barthels et al. Rack-scale in-memory join processing using RDMA. In *Proc. of ACM SIGMOD*, pages 1463–1475, 2015.

[3] C. Barthels et al. Designing databases for future high-performance networks. *IEEE Data Eng. Bull.*, 40(1):15–26, 2017.

[4] C. Barthels et al. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.

[5] C. Binnig et al. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[6] M. Brantner et al. Building a database on S3. In *Proc. of ACM SIGMOD*, pages 251–264, 2008.

[7] M. Cai et al. Integrated querying of SQL database data and S3 data in amazon redshift. *IEEE Data Eng. Bull.*, 41(2):82–90, 2018.

[8] Q. Cai et al. Efficient distributed memory management with RDMA and caching. *PVLDB*, 11(11):1604–1617, 2018.

[9] D. G. Campbell et al. Extreme scale with full sql language support in microsoft sql azure. In *SIGMOD*, 2010.

[10] H. Chen et al. Fast in-memory transaction processing using RDMA and HTM. *ACM Trans. Comput. Syst.*, 35(1):3:1–3:37, 2017.

[11] S. Christgau and B. Schnor. Software-managed cache coherence for fast one-sided communication. In *PMAM@PPoPP*, pages 69–77. ACM, 2016.

[12] B. F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154, 2010.

[13] P. B. Danzig et al. Distributed indexing: A scalable mechanism for distributed information retrieval. In *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Chicago, Illinois, USA, October 13-16, 1991 (Special Issue of the SIGIR Forum).*, pages 220–229, 1991.

[14] A. Devulapalli et al. Distributed queue-based locking using advanced network features. In *34th International Conference on Parallel Processing (ICPP 2005), 14-17 June 2005, Oslo, Norway*, pages 408–415, 2005.

[15] A. Dragojević et al. FaRM: Fast remote memory. In *Proc. of NSDI*, pages 401–414, 2014.

[16] A. Dragojević et al. No compromises: distributed transactions with consistency, availability, and performance. In *Proc. of OSDI*, pages 54–70, 2015.

[17] A. Dragojevic et al. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.*, 40(1):3–14, 2017.

[18] C. Feng et al. Indexing techniques of distributed ordered tables: A survey and analysis. *J. Comput. Sci. Technol.*, 33(1):169–189, 2018.

[19] M. Gries et al. SCC: A flexible architecture for many-core platform research. *Computing in Science and Engineering*, 13(6):79–83, 2011.

[20] A. Kalia et al. Using rdma efficiently for key-value services. In *Proc. of ACM SIGCOMM*, pages 295–306, 2014.

[21] A. Kalia et al. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proc. of OSDI*, pages 185–201, 2016.

[22] J. Krüger et al. Fast updates on read-optimized databases using multi-core cpus. *PVLDB*, 5(1):61–72, 2011.

[23] C. Kulkarni et al. Beyond simple request processing with ramcloud. *IEEE Data Eng. Bull.*, 40(1):62–69, 2017.

[24] P. L. Lehman et al. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[25] V. Leis et al. The ART of practical synchronization. In *DaMoN*, pages 3:1–3:8, 2016.

[26] J. J. Levandoski et al. High performance transactions in deuteronomy. In *CIDR 2015, Online Proceedings*, 2015.

[27] B. Li et al. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[28] F. Li et al. Accelerating relational databases by leveraging remote memory and RDMA. In *SIGMOD Conference*, pages 355–370. ACM, 2016.

[29] F. Liu et al. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In *EuroSys*, pages 48–63, 2017.

[30] S. Loesing et al. On the Design and Scalability of Distributed Shared-Data Databases. In *ACM SIGMOD*, pages 663–676, 2015.

[31] D. B. Lomet. Replicated indexes for distributed data. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*, pages 108–119, 1996.

[32] X. Lu et al. Scalable and distributed key-value store-based data management using rdma-memcached. *IEEE Data Eng. Bull.*, 40(1):50–61, 2017.

[33] Y. Lu et al. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 773–785, 2017.

[34] C. Mitchell et al. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proc. of USENIX ATC*, pages 103–114, 2013.

[35] S. Narravula et al. High performance distributed lock management services using network-based remote atomic operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), 14-17 May 2007, Rio de Janeiro, Brazil*, pages 583–590, 2007.

[36] J. Ousterhout et al. The case for ramcloud. *Communications of the ACM*, 54(7):121–130, 2011.

[37] Delivering Application Performance with Oracle's InfiniBand Technology, 2012.

[38] W. Rödiger et al. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *Proc. of ICDE*, pages 1194–1205, 2016.

[39] A. Salama et al. Rethinking distributed query execution on high-speed networks. *IEEE Data Eng. Bull.*, 40(1):27–37, 2017.

[40] http://snowflake.net/product/architecture.

[41] S. Sur et al. Shared receive queue based scalable MPI design for infiniband clusters. In *IPDPS*, 2006.

[42] J. Vienne et al. Performance analysis and evaluation of infiniband FDR and 40gige roce on HPC and cloud comp. systems. In *IEEE HOTI*, 2012.

[43] D. Y. Yoon et al. Distributed lock management with RDMA: decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1571–1586, 2018.

[44] E. Zamanian et al. The end of a myth: Distributed transaction can scale. *PVLDB*, 10(6):685–696, 2017.

[45] H. Zhang et al. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD*, pages 1567–1581, 2016.

```
uint64_t readLockOrRestart(Node node){
  uint64_t version = awaitNodeUnlocked(node)
  return version
}

void readUnlockOrRestart(Node node, uint64_t version){
  if(version != node.version)
    restart()
}

void upgradeToWriteLockOrRestart(Node node, uint64_t version){
  if(!CAS(node.version, setLockBit(version)){
    restart()
}

void writeUnlock(Node node){
  fetch_add(node.version, 1)
}

uint64_t awaitNodeUnlocked(Node node){
  uint64_t version = node.version
  while (version & 1) == 1 // spinlock
    pause()

  return node.version;
}
```

Listing 3: Helper Methods for Coarse-Grained Index

```
Node remote_read(NodePtr remotePtr){
  return RDMA_READ(remotePtr)
}

uint64_t remote_readLockOrRestart(Node node, NodePtr remotePtr){
  uint64_t version = remote_awaitNodeUnlocked(node, remotePtr)
  return version
}

void remote_upgradeToWriteLockOrRestart(Node node, NodePtr
     remotePtr, uint64_t version){
  if(!RDMA_CAS(remotePtr, node.version, setLockBit(version)){
    restart()
}

void remote_writeUnlock(NodePtr remotePtr, Node node){
  if(node.right_node != NULL){ //node was split
    remNodePtr2 = RDMA_ALLOC(size(node.right_node))
    RDMA_WRITE(remNodePtr2, node.right_node)
  }
  RDMA_WRITE(remNodePtr, node);
  RDMA_FETCH_AND_ADD(remotePtr, 1)
}

uint64_t remote_awaitNodeUnlocked(Node node, NodePtr remotePtr){
  uint64_t version = node.version
  while (version & 1) == 1 // spinlock
    pause()
    node = RDMA_READ(remotePtr)

  return node.version;
}
```

Listing 4: Helper Methods for Fine-Grained Index

A APPENDIX

A.1 Additional Index Operations

In the following, we show operations used to implement the indexing variants in Sections 3 to Sections 5. The operations in Listing 3 are used by the coarse-grained index (Section 3). These operations are called from compute servers by RPC and are then executed by memory servers. The operations in Listing 4 are used by the fine-grained index (Section 4) where compute servers use one-sided RDMA operations to access the index in the memory servers.

A.2 Latency of Index Designs

In this experiment, we analyze the latency of the different workloads using the same experimental setup and data as in experiment 1 (Section 6.1). Figure 13 and Figure 14 show the results for skewed and uniform data distribution. The x-axis again shows the number of clients used for each run and the y-axis the resulting latency for executing one instance of a query in a client.

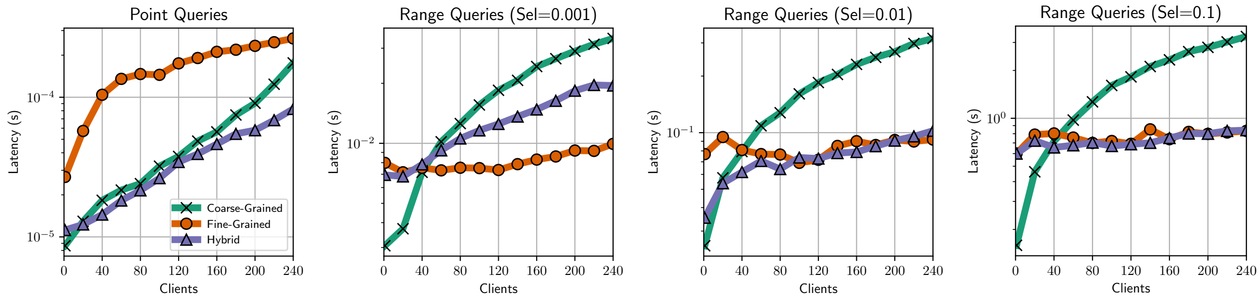
A general pattern that we can see in all workloads is that the latency of the coarse-grained index is best for a low load ($<$ 20 clients). The reason is that this indexing scheme uses RPCs and as long as the memory servers are not becoming CPU bound, the latency in this indexing scheme benefits from using less round-trips between compute and memory servers for index lookups. However, for high load scenarios the fine-grained or the hybrid scheme show smaller latencies than the coarse-grained indexing scheme.

A.3 Effect of Co-location

In this experiment, we analyze the throughput when co-locating compute and memory servers in a NAM architecture. Co-location in the NAM architecture was also discussed in [5, 39, 44] and can be used to mimic a shared-nothing (like) architecture where data and compute is also co-located. The difference is that the memory of all nodes in a co-located NAM architecture is directly accessible not only via RPC but also via one-sided RDMA. The goal of this experiment was to show how the coarse-grained scheme behaves compared to the fine-grained scheme under co-location using typical workload characteristics (i.e., medium load and no skew).

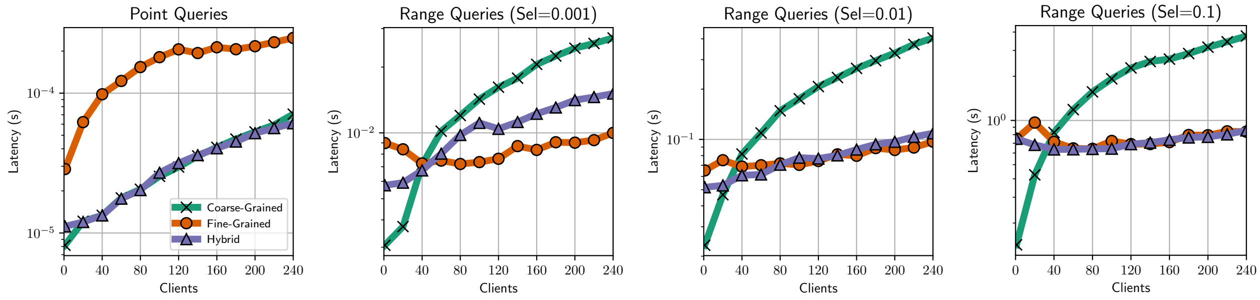
In this experiment, we were running the same workloads as in Exp. 1 (i.e., the read-only workloads A and B) with data of size 100M. For running the workload, we used two NAM variants: one variant with and one without co-location of compute and memory servers. For the variant with co-location, we used 4 physical machines and each of the physical machines hosted a compute and a memory server each running 20 threads pinned to one of the sockets. For the variant without co-location, we used 4 additional dedicated physical machines for compute server, however only one socket of those machine was used for executing the compute server threads. Moreover, the memory servers were deployed on the other 4 machines. To simulate the same resources in the variant without co-location were we used in total 8 machines instead of 4, the compute and memory server used only one socket of each machine (instead of both). Moreover, in both variants (with and without co-location), each memory server used only one port of the RDMA NIC.

For running the workload, every client (i.e., each compute thread) selected the requested key (range) uniformly at random. The throughput results are summarized in Figure 15



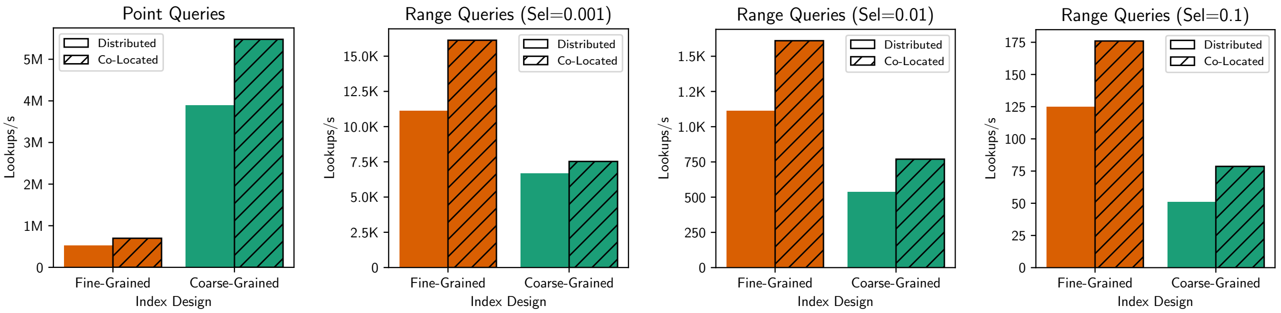
(a) Point Query (b) Range Query (sel=0.001) (c) Range Query (sel=0.01) (d) Range Query (sel=0.1)

Figure 13: Latency for Workloads A and B (Skewed Data, Size 100M)



(a) Point Query (b) Range Query (sel=0.001) (c) Range Query (sel=0.01) (d) Range Query (sel=0.1)

Figure 14: Latency for Workloads A and B (Uniform Data, Size 100M)



(a) Point Query (b) Range Query (sel=0.001) (c) Range Query (sel=0.01) (d) Range Query (sel=0.1)

Figure 15: Effects of Co-location on Throughput (Uniform Data, Size 100M, 80 Clients)

for coarse- and fine-grained. We do not show the results for the hybrid scheme, which again behaved very similar to the coarse-grained for point queries and similar to fine-grained for range queries. As one effect, we can see that all workloads (point and range queries) have a similar relative gain independent when running in the co-located variant. The reason is that in both cases (coarse- and fine-grained), 25% of the memory accesses required by index lookups can be executed locally on the same physical machine where the compute server is running. For coarse-grained, for example, the complete index traversal can be executed locally, if the data requested by a compute server resides in a memory server on the same physical machine. For fine-grained, a compute server cannot execute a complete traversal locally but it can also use local memory accesses for those index pages that reside on the same physical machine with the compute server which requests the data. This also results on average in 25% local accesses since we use 4 dedicated machines for the co-located variant.

This observation is similar to the observations made for co-location in [5, 44] where the authors also report that co-location enables a constant factor of higher throughput depending on the ratio of local/distributed transactions. Furthermore, when looking at the absolute throughput, as expected, under co-location point queries can achieve the highest throughput using the coarse-grained scheme. For range queries, the fine-grained scheme has still the highest throughput similar to what we saw in Exp. 1 (see Section 6.1).

A.4 Opportunities and Challenges of Caching

An interesting dimension in the NAM architecture is caching of hot index nodes in compute servers that are frequently accessed. Caching allows compute servers to avoid remote memory transfers from memory servers. This is similar to the co-location of compute and memory servers as discussed in Appendix A.3, which also allows compute servers to make use of locality. However, different from co-location, for caching, index nodes are replicated from memory servers to compute serves and thus it requires cache invalidation if the index in the memory servers is updated.

For read-only workloads, caching can thus help to avoid expensive remote memory accesses and significantly improve the lookup performance of tree-based indexes in a NAM architecture since no invalidation of cached data is required. We believe that especially the fine-grained scheme benefits from caching since it requires multiple round-trips between compute and memory servers to traverse the index.

Furthermore, the other indexing schemes (coarse-grained and hybrid) can also benefit, especially for range-queries, since (potentially large) results do not need to be transferred fully from memory to compute servers anymore. However, for workloads which include writes (i.e., inserts and deletes of index entries), caching becomes a non-trivial problem since cached index nodes on compute servers need to be invalidated if the index on the memory servers changes.

The problem of cache invalidation has also been discussed in [8], which presents general caching strategies for remote memory accesses using RDMA. For tree-based indexes, where inserts and deletes might propagate up to the index from the leaf level to the root node, we observed that cache invalidation is even a more severe issue since one insert/delete operation might need to invalidate multiple cached index nodes. To that end, we believe that there is a need for developing new caching strategies that take the particularities of tree-based indexes into account to decide whether or not to cache an index node. However, providing an in-depth discussion and analysis of different caching strategies for tree-based indexes that can adapt themselves to the workload is beyond the scope of this paper and presents an interesting avenue of future work.