The End of a Myth: Distributed Transactions Can Scale

Erfan Zamanian
Brown University
erfanz@cs.brown.edu

Carsten Binnig
Brown University
carsten_binnig@brown.edu

Tim Harris
Oracle Labs
timothy.l.harris@oracle.com

Tim Kraska
Brown University
tim_kraska@brown.edu

ABSTRACT

The common wisdom is that distributed transactions do not scale. But what if distributed transactions could be made scalable using the next generation of networks and a redesign of distributed databases? There would no longer be a need for developers to worry about co-partitioning schemes to achieve decent performance. Application development would become easier as data placement would no longer determine how scalable an application is. Hardware provisioning would be simplified as the system administrator can expect a linear scale-out when adding more machines rather than some complex sub-linear function, which is highly application specific.

In this paper, we present the design of our novel scalable database system NAM-DB and show that distributed transactions with the very common Snapshot Isolation guarantee can indeed scale using the next generation of RDMA-enabled network technology without any inherent bottlenecks. Our experiments with the TPC-C benchmark show that our system scales linearly to over 6.5 million new-order (14.5 million total) distributed transactions per second on 56 machines.

1 Introduction

The common wisdom is that distributed transactions do not scale [40, 22, 39, 12, 37]. As a result, many techniques have been proposed to avoid distributed transactions ranging from locality-aware partitioning [35, 33, 12, 43] and speculative execution [32] to new consistency levels [24] and the relaxation of durability guarantees [25]. Even worse, most of these techniques are not transparent to the developer. Instead, the developer not only has to understand all the implications of these techniques, but also must carefully design the application to take advantage of them. For example, Oracle requires the user to carefully specify the co-location of data using special SQL constructs [15]. A similar feature was also recently introduced in Azure SQL Server [2]. This works well as long as all queries are able to respect the partitioning scheme. However, transactions crossing partitions usually observe a much higher abort rate and relatively unpredictable performance [9]. For other applications (e.g., social apps), a developer might not even be able to design a proper sharding scheme since those applications are notoriously hard to partition.

But what if distributed transactions could be made scalable using the next generation of networks and we could rethink the distributed database design? What if we would treat every transaction as a distributed transaction? The performance of the system would become more predictable. The developer would no longer need to worry about co-partitioning schemes in order to achieve scalability and decent performance. The system would scale out linearly when adding more machines rather than sub-linearly because of partitioning effects, making it much easier to provision how much hardware is needed.

Would this make co-partitioning obsolete? Probably not, but its importance would significantly change. Instead of being a necessity to achieve a scalable system, it becomes a second-class design consideration in order to improve the performance of a few selected queries, similar to how creating an index can help a selected class of queries.

In this paper, we will show that distributed transactions with the common Snapshot Isolation scheme [8] can indeed scale using the next generation of RDMA-enabled networking technology without an inherent bottleneck other than the workload itself. With Remote-Direct-Memory-Access (RDMA), it is possible to bypass the CPU when transferring data from one machine to another. Moreover, as our previous work [10] showed, the current generation of RDMA-capable networks, such as InfiniBand FDR $4\times$, is already able to provide a bandwidth similar to the aggregated memory bandwidth between a CPU socket and its attached RAM. Both of these aspects are key requirements to make distributed transactions truly scalable. However, as we will show, the next generation of networks does not automatically yield scalability without redesigning distributed databases. In fact, when keeping the "old" architecture, the performance can sometimes even decrease when simply migrating a traditional database from an Ethernet network to a high-bandwidth InfiniBand network using protocols such as IP over InfiniBand [10].

1.1 Why Distributed Transactions are considered not scalable

To value the contribution of this paper, it is important to understand why distributed transactions are considered not scal-

Proceedings of the VLDB Endowment, Vol. 10, No. 6

able. One of the most cited reasons is the increased contention likelihood. However, contention is only a side effect. Perhaps surprisingly, in [10] we showed that the most important factor is the CPU overhead of the TCP/IP stack. It is not uncommon that the CPU spends most of the time processing network messages, leaving little room for the actual work.

Additionally, the network bandwidth also significantly limits the transaction throughput. Even if transaction messages are relatively small, the aggregated bandwidth required to handle thousands to millions of distributed transactions is high [10], causing the network bandwidth to quickly become a bottleneck, even in small clusters. For example, assume a cluster of three servers connected by a 10Gbps Ethernet network. With an average record size of 1KB, and transactions reading and updating three records on all three machines (i.e., one per machine), 6KB has to be shipped over the network per transaction, resulting in a maximal overall throughput of $\sim 29k$ distributed transactions per second.

Furthermore, because of the high CPU-overhead of the TCP/IP stack and a limited network bandwidth of typical 1/10Gbps Ethernet networks, distributed transactions have much higher latency, significantly higher than even the message delay between machines. This causes the commonly observed high abort rates due to time-outs and the increased contention likelihood; a side-effect rather than the root cause.

Needless to say, there are workloads for which the contention is the primary cause of why distributed transactions are inherently not scalable. For example, if every single transaction updates the same item (e.g. incrementing a shared counter), the workload is not scalable simply because of the existence of a single serialization point. In this case, avoiding the additional network latencies for distributed message processing would help to achieve a higher throughput but not to make the system ultimately scalable. Fortunately, in many of these "bottleneck" situations, the application itself can easily be changed to make it truly scalable [1, 5].

1.2 The Need for a System Redesign

Assuming a scalable workload, the next generation of networks remove the two dominant limiting factors for scalable distributed transaction: the network bandwidth and CPU overhead. Yet, it is wrong to assume that the hardware alone solves the problem. In order to avoid the CPU message overhead with RDMA, many data structures have to change. In fact, RDMA-enabled networks change the architecture to a hybrid shared-memory and message-passing architecture: it is neither a distributed shared-memory system (as several address spaces exist and there is no cache-coherence protocol), nor is it a pure message-passing system since memory of a remote machine can be directly accessed via RDMA reads and writes.

While there has been work on leveraging RDMA for distributed transactions, most notably FaRM [13, 14], most works still rely on locality and more traditional message transfers, whereas we believe locality should be a second-class design consideration. Even more importantly, the focus of existing works is not on leveraging fast networks to achieve a truly scalable design for distributed databases, which is our main contribution. Furthermore, our proposed system shows how to leverage RDMA for Snapshot Isolation (SI) guarantees, which is the most common transaction guarantee in practice [18] because it allows for long-running read-only queries without expensive read-set validations. Other RDMA-based systems focus instead on serializability [14] or do not have transaction support at all [20]. At the same time, existing (distributed) SI schemes typically rely on a single global snapshot counter or timestamp; a fundamental issue obstructing scalability.

1.3 Contribution and Outline

In our vision paper [10], we made the case for a shift in the way transactional and analytical database systems must be designed and showed the potential of efficiently leveraging high-speed networks and RDMA. In this paper, we follow up on this vision and present and evaluate one of the first transactional systems for high-speed networks and RDMA. In summary, we make the following main contributions: (1) We present the full design of a truly scalable system called NAM-DB and propose scalable algorithms specifically for Snapshot Isolation (SI) with (mainly one-sided) RDMA operations. In contrast to our initial prototype [10], the presented design has much less restriction on workloads, supports index-based range-request, and efficiently executes long-running read transactions by storing more than one version per record. (2) We present a novel RDMA-based and scalable global counter technique which allows for efficiently reading the latest consistent snapshot in a distributed SI-based protocol. (3) We show that NAM-DB is truly scalable using a full implementation of TPC-C. Most notably, for the standard configuration of TPC-C benchmark, we show that our system scales linearly to over 3.6 million transactions per second on 56 machines, and 6.5 million transactions with locality optimizations, which is 2 million more transactions per second than what FARM [14] achieves on 90 machines. Note, that our total transaction throughput is even higher (14.5 million transactions per second) as TPC-C specifies to only report the new-order transactions.

2 System Overview

InfiniBand offers two network communication stacks: IP over InfiniBand (IPoIB) and remote direct memory access (RDMA). IPoIB implements a classic TCP/IP stack over InfiniBand, allowing existing database systems to run on fast networks without any modifications. While IPoIB provides an easy migration path from Ethernet to InfiniBand, IPoIB cannot fully leverage the network's capabilities and sometimes even degrades the system performance [10]. On the other hand, RDMA provides a *verbs* API, which enables remote data transfer using the processing capabilities of an RDMA NIC (RNIC), but also requires a radical system redesign. When using RDMA verbs, most of the processing is executed by the RNIC without OS involvement, which is essential for achieving low latencies. The verbs API has two operation types: one-sided verbs (read, write and atomic operations) where only the CPU of the initiating node is actively involved in the communication, and two-sided verbs (send and receive) where the CPUs of both nodes are involved (for more details see [10]). Redesigning distributed databases to efficiently make use of RDMA is a key challenge that we tackle in this paper.

In the following, we first give a brief overview of the network-attached-memory (NAM) architecture [10] that was designed

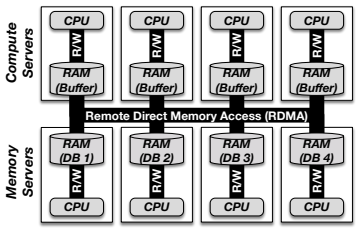


Figure 1: The NAM Architecture

to efficiently make use of RDMA. We then discuss the core design principles of NAM-DB, which builds upon NAM, to enable a scalable transactional system without an inherent bottleneck other than the workload itself.

2.1 The NAM Architecture

The NAM architecture logically decouples compute and storage nodes and uses RDMA for communication between all nodes as shown in Figure 1. The idea is that memory servers provide a shared distributed memory pool that holds all the data, which can be accessed via RDMA from compute servers that execute transactions. This design already highlights that locality is a tuning parameter. In contrast to traditional architectures which physically co-locate the transaction execution with the storage location from the beginning as much as possible, the NAM architecture separates them. As a result all transactions are by default distributed transactions. However, we allow users to add locality as an optimization like an index, as we will explain in Section 6.

Memory Servers: In a NAM architecture memory servers hold all data of a database system such as tables, indexes as well as all other state for transaction execution (e.g., logs and metadata). From the transaction execution perspective, memory servers are "dumb" since they provide only memory capacity to compute servers. However, memory servers still have important tasks such as memory management to handle remote memory allocation calls from compute servers, as well as garbage collection to ensure that enough free space is always available for compute servers, e.g. to insert new records. Durability of the data stored by memory servers is achieved in a similar way as described in [14] by using an uninterruptible power supply (UPS). When a power failure occurs, memory servers use the UPS to persist a consistent snapshot to disks. On the other hand, hardware failures are handled through replication as discussed in Section 6.2.

Compute Servers: The main task of compute servers is to execute transactions over the data items stored in the memory servers. This includes finding the storage location of records on memory servers, inserting/ modifying/ deleting records, as well as committing or aborting transactions. Moreover, compute servers are in charge of performing other tasks, which are required to ensure that transaction execution fulfills all ACID properties such as logging and consistency control. Again, the strict separation of transaction execution in compute servers from managing the transaction state stored in memory servers is what distinguishes our design from traditional distributed database systems. As a result, the performance of the system is independent of the location of the data.

2.2 Design Principles

We now describe challenges in designing NAM-DB to achieve a scalable system design using the NAM architecture.

Separation of Compute and Memory: Although the separation of compute and storage is not new [11, 27, 29, 14], existing database systems that follow this design typically push data access operations into the storage layer. When scaling out the compute servers and pushing data access operations from multiple compute servers into the same memory server, memory servers are likely to become a bottleneck. Even worse, with traditional socket-based network operations, every message consumes additional precious CPU cycles.

In a NAM architecture, we therefore follow a different route. Instead of pushing data access operations into the storage layer, the memory servers provide a fine-grained byte-level data access. In order to avoid any unnecessary CPU cycles for message handling, compute servers exploit one-sided RDMA operations as much as possible. This makes the NAM architecture highly scalable since all computation required to execute transactions can be farmed out to compute servers.

Finally, for the cases where the aggregated main memory bandwidth is the main bottleneck, this architecture also allows us to increase the bandwidth by scaling out the memory servers. It is interesting to note that most modern Infiniband switches, such as Mellanox SX6506 108-Port FDR (56Gb/s) ports InfiniBand Switch, are provisioned in a way that they allow the full duplex transfer rate across all machines at the same time and therefore do not limit the scalability.

Data Location Independence: Another design principle is that compute servers in a NAM architecture are able to access any data item independent of its storage location (i.e., on which memory server this item is stored). As a result, the NAM architecture can easily move data items to new storage locations (as discussed before). Moreover, since every compute server can access any data item, we can also implement work-stealing techniques for distributed load balancing since any compute node can execute a transaction independent of the storage location of the data.

This does not mean that compute servers can not exploit data locality if the compute server and the memory server run on the same physical machine. However, exploring locality becomes just an optimization, not a first-class design decision, that can be added on top of our scalable system, like an index.

Partitionable Data Structures: As discussed before, in the NAM architecture every compute server should be able to execute any functionality by accessing the externalized state on the memory servers. However, this does not prevent a single memory region (e.g., a global read or commit timestamp) from becoming a bottleneck.Therefore, it is important that every data structure is partitionable. For instance, following this design principle, we invented a new data structure to implement a partitionable read/commit timestamp (see Section 4).

3 The Basic SI-Protocol

In this section, we describe first our Snapshot Isolation protocol for the NAM architecture as already introduced in our vision paper [10]. Afterwards, we analyze potential scalability bottlenecks for distributed transactions, which we then address in Sections 4-6 as the main contributions of this paper.

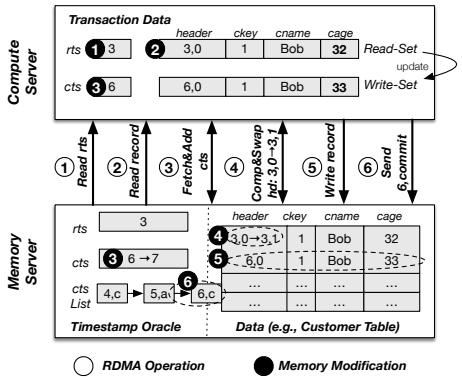


Figure 2: Naïve RDMA-based SI-Protocol

```
1  runTransaction(Transaction t) {
2    // get read timestamp
3    rts = RDMA_Read(&r(rts));
4    // build write-set
5    t.execute(rts);
6    // get commit timestamp
7    cts = RDMA_FetchAndAdd(&r(cts), 1);
8
9    // verify write version and lock write-set
10   commit = true;
11   parfor i in size(t.writeSet) {
12    header = t.readSet[i].header;
13    success[i] = RDMA_CompAndSwap(&r(header), header, setLockBit(header));
14    commit = commit &&  success[i];
15   }
16
17   // install write-set
18   if(commit) {
19     parfor i in size(t.writeSet)
20       RDMA_Write(&r(t.readSet[i]), t.writeSet[i]);
21   }
22   //reset lock bits
23   else {
24     parfor i in size(t.writeSet) {
25       if(success[i])
26         header = t.readSet[i].header;
27         RDMA_Write(&r(header), header);
28     }
29   }
30   RDMA_Send([cts,commit]); //append cts and result to ctsList
31 }
```

Listing 1: Transaction Execution in a Compute Server

3.1 A Naïve RDMA Implementation

With Snapshot Isolation (SI), a transaction reads the most recent (consistent) snapshot of a database that was committed before the beginning of the transaction, and it does not see concurrent updates. Furthermore, transactions only abort if it would overwrite a concurrent change. For distributed systems, Generalized SI (GSI) [16] is more common as it allows any committed snapshot (and not only the most recent one) to be read. While we also use GSI for NAM-DB, our goal is still to read a recent committed snapshot to avoid high abort rates.

Listing 1 and the corresponding Figure 2 show a naïve SI protocol using only one-sided RDMA operations that are based on a global timestamp oracle as implemented in commercial systems [9]. To better focus on the interaction between compute and memory servers, we made the following simplifying assumptions: First, we did not consider the durability guarantee and the recovery of transactions. Second, we assume that there is an up-to-date catalog service, which helps compute servers to find the remote address of a data item in the pool of memory servers; the remote address is returned by the $\&_r$ operator in our pseudocode. Finally, we consider a simple variant of SI where only one version is kept around for each record. Note that these assumptions are only made in this section and we tackle each one later in this paper.

For **executing** a transaction, the compute server first fetches the read-timestamp rts using an RDMA read (step ① in Figure 2, line 3 in Listing 1). The rts defines a valid snapshot for the transaction. Afterwards, the compute server executes the transaction, which means that the required records are read remotely from the memory servers using RDMA read operations (e.g., the record with $ckey = 3$ in the example) and updates are applied locally to these records; i.e., the transaction builds its read- and write-set (step ② in Figure 2, line 5 in Listing 1). Once the transaction has built its read- and write-set, the compute server starts the commit phase.

For **committing**, a compute server fetches a unique commit timestamp (cts) from the memory server (step ③ in Figure 2, line 7 in Listing 1). In the naïve protocol, fetching a unique cts counter is implemented using an atomic RDMA fetch-and-add operation that returns the current counter and increments it in the memory server by 1. Afterwards, the compute server verifies and locks all records in its write-set on the memory servers using one RDMA compare-and-swap operation (line 10-15 in Listing 1). The main idea is that each record stores a header that contains a version number and a lock bit in an 8-Byte memory region. For example, in Figure 2, $(3, 0)$ stands for version 3 and lock-bit 0 (0 means not locked). The idea of the compare-and-swap operation is that the compute server compares the version in its read-set to the version installed on the memory-server for equality and checks that the lock-bit is set to 0. If the compare succeeds, the atomic operation swaps the lock bit to 1 (step ④ in Figure 2, line 13 in Listing 1).

If compare-and-swap succeeds for all records in the write-set, the compute server installs its write-set using RDMA writes (line 19-20 in Listing 1). These RDMA writes update the entire record including updating the header, installing the new version and setting the lock-bit back to 0. For example, $(6, 0)$ is remotely written on the header in our example (step ⑤ in Figure 2). If the transactions fails, the locks are simply reset again using RDMA writes (line 24-28 in Listing 1).

Finally, the compute server appends the outcome of the transaction (commit or abort) as well as the commit timestamp cts to a list ($ctsList$) in the memory server (step ⑥ in Figure 2, line 32 in Listing 1). Appending this information can be implemented in different ways. However, for our naïve implementation we simply use an unsignaled RDMA send operation; i.e., the compute server does not need to wait for the cts to be actually sent to the server, and gives every timestamp a fixed position (i.e., timestamp value - offset) to set a single bit indicating the success of a transaction. This is possible, as the fetch and add operation creates continuous timestamp values.

Finally, the **timestamp oracle** is responsible for advancing the read timestamp by scanning the queue of completed transactions. It therefore scans $ctsList$ and tries to find the highest commit timestamp (i.e., highest bit) so that every transactions before that timestamp are also committed (i.e., all bits are set). Since advancing the read timestamp is not in the critical path, the oracle uses a single thread that continuously scans the memory region to find the highest commit timestamp and also adjusts the offset if the servers run out of space.

3.2 Open Problems and Challenges

While the previously-described protocol achieves some of our goals (e.g., it heavily uses one-sided RDMA to access the memory servers), it is still not scalable. The main reason is that global timestamps have inherit scalability issues [17], which are emphasized further in a distributed setting.

First, for every transaction, each compute server uses an RDMA atomic fetch-and-add operation to the same memory region to get a unique timestamp. Obviously, atomic RDMA operations to the same memory location scale poorly with the number of concurrent operations since the network card uses internal latches for the accessed memory locations. In addition, the oracle is involved in message passing and executes a timestamp management thread that needs to handle the result of each transaction and advance the read timestamp. Although this overhead is negligible for a few thousands transactions, it shows its negative impact when millions of transactions are running per second.

The second problem with the naïve protocol is that it likely results in high abort rates. A transaction's snapshot can be "stale" if the timestamp management thread can not keep up to advance the read timestamp. Thus, there could be many committed snapshots which are not included in the snapshot read by a transaction. The problem is even worse for hot spots, i.e. records which are accessed and modified frequently.

A third problem is that slow workers also contribute to high abort rate by holding back the most recent snapshot from getting updated. In fact, the oracle only moves forward the read timestamp rts as fast as the slowest worker. Note that long transactions have the same effect as slow workers.

Finally, the last problem is that the naïve implementation does not have any support for fault-tolerance. In general, fault-tolerance in a NAM architecture is quite different (and arguably less straight-forward) than in the traditional architecture. The reason is that the transactions' read- and write-sets (including the requested locks) are managed directly by the compute servers. Therefore, a failure of compute servers could potentially result in undetermined transactions and thus abandoned locks. Even worse, in our naïve implementation, a compute server that fails can lead to "holes" in the $ctsList$ that cause the read timestamp to not advance anymore.

4 Timestamp Oracle

In this section, we first describe how to tackle the issues outlined in Section 3.2 that hinder the scalability of the timestamp oracle as described in our naïve SI-protocol implementation. Afterwards, we discuss some optimizations.

4.1 Scalable Timestamp Generation

The main issues with the current implementation of the timestamp oracle are: (1) The timestamp management thread that runs on a memory server does not scale well with the number of transactions in the system. (2) Long running transactions/slow compute servers prevent the oracle from advancing the read timestamp, further contributing to the problem of too many aborts. (3) High synchronization costs of RDMA atomic operations when accessing the commit timestamp cts are stored in one common memory region.

In the following, we explain how we tackle these issues to build a scalable timestamp oracle. The main idea is using a data structure called the **timestamp vector**, similar to a vector clock, which represents the read timestamp as the following:

$$T_R = \langle t_1, t_2, t_3, ..., t_n \rangle$$

Here, each component t_i in T_R is a unique counter that is assigned to one transaction execution thread i in a compute server, where i is a globally unique identifier. This vector can either be stored on one of the memory servers or also be partitioned across several servers as explained later. However, in contrast to vector clocks, we do not store the full vector with every record, but instead store only the timestamp of the compute server that did the latest update:

$$T_C = \langle i, t_i \rangle$$

Here, i is the global transaction execution thread identifier and t_i the corresponding commit timestamp. This helps to mitigate one of the most fundamental drawbacks of vector clocks, the high storage overhead per record.

Commit Timestamps: Each component $t_i = T_R[i]$ represents the latest commit timestamp that was used by an execution thread i. Creating a new commit timestamp can be done without communication since one thread i executes transactions in a closed loop. The reason is that each thread already knows its latest commit timestamp and just needs to increase it by one to create the next commit timestamp. It then uses the previously-described protocol to verify if it is safe to install the new versions in the system with timestamp $T_C = \langle i, t+1 \rangle$ where $t+1$ is the new timestamp.

At the end of the transaction, the compute server makes the updates visible by increasing the commit timestamp in the vector T_R. That is, instead of adding the commit timestamp to a queue (line 32 of Listing 1), it uses an RDMA write to increase its latest timestamp in the vector T_R. No atomic operations are necessary since each transaction thread i only executes one transaction at a time.

Read Timestamps: Each transaction thread i reads the complete timestamp vector T_R and uses it as read timestamp rts. Using T_R, a transaction can read a record, including its header from a memory server, and check if the most recent version is visible to the transaction. The check is simple: as long as the version of the record $\langle i, t \rangle$ is smaller or equal to t_i of the vector T_R, the update is visible. If not, an older version has to be used to meet the condition. We will discuss details of the memory layout of our multi-versioning scheme in Section 5.

It is important to note that this simple timestamp technique has several important characteristics. First, long running transactions, stragglers, or crashed machines do not prevent the read timestamp to advance. The transaction threads are independent of each other. Second, if the timestamp is stored on a single memory server, it is guaranteed to increase **monotonically**. The reason is that all RDMA writes are always materialized in the remote memory of the oracle and not cached on its NIC. Therefore, it is impossible for one transaction execution thread in a compute server to see a timestamp vector like $\langle .., t_n, .., t_m + 1, .. \rangle$ while another observes $\langle .., t_n + 1, .., t_m, .. \rangle$. As a result the timestamps are still progressing monotonically, similar to a single global timestamp counter. However, in the case where the timestamp vector is partitioned, this property might no longer hold true as explained later.

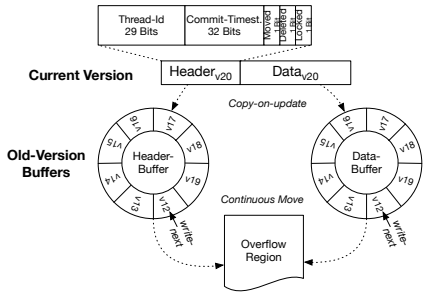


Figure 3: Version Management and Record Layout

4.2 Further Optimizations

In the following, we explain further optimizations to make the timestamp oracle even more scalable:

Dedicated Fetch Thread: Fetching the most recent T_R at the beginning of each transaction can cause a high network load for large transaction execution thread pools on large clusters. In order to reduce the network load, we can have one dedicated thread per compute server that continuously fetches T_R, and allow all transaction threads to simply use the pre-fetched T_R. At a first view, this seems to increase the abort rate since pre-fetching increases the staleness of T_R. However, due to the reduced network load, the runtime of each transaction is heavily reduced, leading instead to a lower abort rate.

Compression of T_R: The size of T_R depends on the number of transaction execution threads, which could rise up to hundreds or even thousands entries when scaling out. Thus, instead of having one slot per thread, we can compress T_R by having only one slot t_i per compute server; i.e., all transaction execution threads on one machine share one timestamp slot t_i. One alternative is that the threads of a compute server use an atomic RDMA fetch-and-add operation to increase the counter value. Since the number of transaction execution threads per compute server is bounded (if we use one dedicated thread per core) the contention will not be too high. As another alternative, we can cache t_c in a compute server's memory. Increasing t_c is then implemented by a local compare-and-swap followed by a subsequent RDMA write.

Partitioning of T_R: In our evaluation, we found that with those two optimizations, a single timestamp server is already able to sustain over 140 million trxs/sec using a single dual-port FDR 4x NIC. In other words, we could scale our cluster to ≈ 500 machines for TPC-C with two FDR 4x ports before the network bandwidth of the server becomes a bottleneck. In case a single server becomes the bottleneck, it is easy to partition T_R across several memory nodes, since a transaction execution thread needs to update only a single slot t_i. This will improve the bandwidth per server as every machine now only stores a fraction of the vector. Unfortunately, partitioning T_R no longer guarantees strict monotonicity. As a result, every transaction execution thread still observes a monotonically increasing order of updates, but the order of transactions between transaction execution threads might be different. While we believe that this does not impose a big problem in real systems, we are currently investigating if we can solve this by leveraging the message ordering guarantees provided by InfiniBand for certain broadcast operations. This direction represents an interesting avenue of future work.

5 Memory Servers

In this section, we first discuss the details of the multi-versioning scheme implemented in the memory serves of NAM-DB, which allows compute servers to install and find a version of a record. Afterwards, we present further details about the design of table and index structures in NAM-DB— as well as memory management including garbage collection. Note that our design decisions are made to make distributed transactions scalable rather than optimize for locality.

5.1 Multi-Versioning Scheme

The scheme to store multiple versions of a database record in a memory server is shown in Figure 3. The main idea is that the most recent version of a record, called the *current version*, is stored in a dedicated memory region. Whenever a record is updated by a transaction (i.e., a new version needs to be installed), the current version is moved to an *old-version buffer* and the new current version is installed in-place. As a result, **the most recent version can always be read with a single RDMA request**. Furthermore, as we use continuous memory regions for the most recent versions, transferring the most recent versions of several records is also only a single RDMA request, which dramatically helps scans. The old-version buffer has a fixed size to be efficiently accessible with one RDMA read. Moreover, the oldest versions in the buffers are continuously copied to an *overflow region*. That way, slots in the old-version buffer can be re-used for new versions while keeping old versions available for long running transactions.

In the following, we first explain the memory layout in more detail and then discuss the version management.

Record Layout: For each record, we store a *header* section that contains additional metadata and a *data* section that contains the payload. The data section is a full copy of the record which represents a particular version. Currently, we only support fixed-length payloads. Variable-length payloads could be supported by storing an additional pointer in the data section of a record that refers to the variable-length part, which is stored in a separate memory region. However, when using RDMA, the latency for messages of up to 2KB remains constant, as shown in our vision paper [10]. Therefore, for many workloads where the record size does not exceed this limit, it makes sense to store the data in a fixed-length field that has the maximal required length inside a record.

The header section describes the metadata of a record. In our current implementation, we use an 8-byte value that can be atomically updated by a remote compare-and-swap operation from compute servers. The header encodes different variables: The first 29 bits are used to store the *thread identifier i* of the transaction execution thread that installed the version (as described in the section before). The next 32 bits are used for the *commit timestamp*. Both these variables represent the version information of a record and are set during the commit phase. Moreover, we also store other data in the header section that is used for version management, each represented by a 1-bit value: a *moved*-bit, a *deleted*-bit, and a *locked*-bit. The moved-bit indicates if a version was already moved from the old-version buffer to the overflow region, and thus its slot can be safely reused. The deleted-bit indicates if the version of a record is marked for deletion and can be safely garbage col-

lected. Finally, the locked-bit is used during the commit phase to avoid concurrent updates after the version check.

Version Management: The *old-version buffer* consists of two circular buffers of a fixed size as shown in Figure 3; one that holds only headers (excluding the current version) and another one that holds data sections. The reason for splitting the header and the data section into two circular old-version buffers is that the size of the header section is typically much smaller than the data section. That way, a transaction that needs to find a particular version of a record only needs to fetch all headers without reading the payload. Once the correct version is found, the payload can be fetched with a separate read operation using the offset information. This effectively minimizes the latency when searching a version of a record.

For installing a new version, we first validate if the current version has not changed since reading it and set the lock-bit using one atomic RDMA compare-and-swap operation (i.e., we combine validation and locking). If locking and validation fails, we abort. Otherwise, the header and the data of the current version is copied to the old version buffer. In order to copy the current version to the buffers, a transaction first needs to determine the slot which stores the oldest version in the circular buffer and find out if that slot can be overwritten (i.e., the moved-bit is set to 1). In order to identify the slot with the oldest version, the circular buffers provide a next-write counter.

Another issue is that the circular buffers have only a fixed capacity, the reason that we want to efficiently access them with one-sided RDMA operations and avoid pointer-chasing operations. However, in order to support long-running transactions a version-mover thread that runs in a memory server which continuously moves the header and data section to an overflow region and sets the moved-bit in the header-buffer to 1. This does not actually mean that the tuples are removed from the old-versions buffers. It only means that it can be safely re-used by a transaction to install a new version. Keeping the moved versions in the buffer maximizes the number of versions that can be retrieved from the old-version buffers.

5.2 Table and Index Structures

In the following, we discuss how table and index structures are implemented in memory servers.

Table Structures: In NAM-DB, we only support one type of table structure that implements a hash table design similar to the one in [30]. In this design, compute servers can execute all operations on the hash table (e.g., put or a get) by using one-sided RDMA operations. In addition to the normal put and get operations to insert and lookup records of a table, we additionally provide an update to install a new record version, as well as a delete operation. The hash tables in NAM-DB stores key-value pairs where the keys represent the primary keys of the table. Moreover, the values store all information required by our multi-versioning scheme: the current record version and three pointers (two pointers to the old-version buffers as well as one pointer to the overflow region).

In contrast to [30], hash tables in NAM-DB are partitioned to multiple memory servers. In order to partition the hash table, we split the bucket array into equally-sized ranges and each memory server stores only one of the resulting sub-ranges as well as the corresponding keys and values. In order to find a memory server which stores the entries for a given key, the compute server only needs to apply the hash function which determines the bucket index (and thus the memory server which holds the bucket and its key-value pairs). Once the memory server is identified, the hash table operation can be executed on the corresponding memory server.

Index Structures: In addition to the table structure described before, NAM-DB supports two types of secondary indexes: a hash-index for single-key lookups and a B^+-tree for range lookups. Both types of secondary indexes map a value of the secondary attribute to a primary key that can then be used to lookup the record using the table structure dicussed before (e.g., a customer name to the customer key). Moreover, secondary indexes do not store any version information. Thus, retrieving the correct version of a record requires a subsequent lookup on the table structure using the primary key.

For NAM-DB's secondary hash indexes, we use the same hash table design that we have described before for the table design. The main difference is that for values in a secondary hash index, we store only primary keys and no pointers (e.g., to old-version buffers etc.) as discussed before. For the B^+-tree index, we follow a different route. Instead of designing a tree structure that can be accessed purely by one-sided RDMA operations, we use two-sided RDMA operations to implement the communication between compute and memory servers. The reason is that operations in B^+-trees need to chase multiple pointers from the root to the leaf level, and we do not want to pay the network overhead for pointer chasing. While pointer chasing is also an issue for hash-tables, which use linked lists, [30] shows that clustering keys in a linked list into one memory region largely mitigates this problem (i.e., one RDMA operation can read the entire linked list). Moreover, for scaling-out and preventing individual memory servers from becoming a bottleneck, we range partition B^+-trees to different memory servers. In the future, we plan to investigate alternative indexing designs for B^+ trees.

5.3 Memory Management

Memory servers store tables as well as index structures in their memory as described before. In order to allow compute servers to access tables and indexes via RDMA, memory servers must pin and register memory regions at the RDMA network interface card (NIC). However, pinning and registering memory at the NIC are both costly operations which should not be executed in a critical path (e.g., whenever a transaction created a new table or an index). Therefore, memory servers allocate a large chunk of memory during initialization and register it to the NIC. After initialization, memory servers handle both allocate and free calls from compute servers.

Allocate and Free Calls: Allocate and free calls from compute servers to memory servers are implemented using two-sided RDMA operations. In order to avoid many small memory allocation calls, compute servers request memory regions from memory servers in extends. The size of an extend can be defined by a compute server as a parameter and depends on different factors (e.g., expected size and update rate). For example, when creating a new table in NAM-DB, a compute server that executed the transaction allocates an extend that allows the storage of an expected number of records and their

different versions. The number of expected records per table can be defined by applications as a hint.

Garbage Collection: To prevent old versions from taking up all the space, the job of the garbage collection is to determine old version records which can be safely evicted. In NAM-DB, garbage collection is implemented by having a timeout on the maximal transaction execution time E that can be defined as a parameter by the application. Transactions that run longer than the maximal execution time might abort since the version they require might already be garbage collected. For garbage collecting these versions, each memory server has a garbage collection thread which continuously scans the overflow regions and sets the deleted-bit of the selected versions of a record 1. These versions are then truncated lazily from the overflow regions once contiguous regions can be freed.

6 Compute Servers

In this section, we discuss how compute servers execute transactions and present techniques for recovery/fault-tolerance.

6.1 Transaction Execution

Compute servers use multiple so called *transaction execution threads* to execute transactions over the data stored in the memory servers. Each transaction execution thread i executes transactions sequentially using the complete timestamp vector T as the read timestamp, as well as $(i, T[i])$ as the commit timestamp, to tag new versions of records as discussed in Section 4. The general flow of executing a single transaction in a transaction execution thread is the same workflow as outlined already in Section 3.1. Indexes are updated within the boundaries of the transaction that also updates the corresponding table using RDMA operations (i.e., we pay additional network roundtrips to update the indexes).

One import aspect that we have not discussed so far is how the database catalog is implemented such that transactions can find the storage location of tables and indexes. The catalog data is hash-partitioned and stored in memory servers. All accesses from compute servers are implemented using two-sided RDMA operations since query compilation does not result in a high load on memory servers when compared to the actual transaction execution. Since the catalog does not change too often, the catalog data is cached by compute servers and refreshed in two cases. In the first case, a requested database object is not found in the cached catalog, and the compute server requests the required meta data from the memory server. The second case is if a database object is altered. We detect this case by storing a catalog version counter within each memory server that is incremented whenever an object is altered on that server. Since transaction execution threads run transactions in a closed loop, this counter is read from the memory server that stores the metadata for the database objects of a given transaction before compiling the queries of that transaction. If the version counter has changed when compared to the cached counter, the catalog entries are refreshed.

6.2 Failures and Recovery

NAM-DB provides a fault-tolerance scheme that handles failures of both compute and memory servers. In the following, we discuss both cases. At the moment, we do not handle failures resulting from network partitioning since the events are extremely rare in InifiniBand networks. These types of failures could be handled using a more complex commit protocol than 2PC (e.g., a version of Paxos based on RDMA), which is an interesting avenue of future work. Moreover, it is also important to note that high-availability is not the design goal of NAM-DB, which could be achieved in NAM-DB by replicating the write-set during the commit phase.

Memory Server Failures: In order to tolerate memory server failures, each transaction execution thread of a compute server writes a private log journal to a memory server using RDMA writes. In order to avoid the loss of a log, each transaction execution thread writes its journal to more than one memory server. The entries of such a log journal are $< T, S >$ where T is the read snapshot used by thread i and S is the executed statement with all its parameters. Commit timestamps that have been used by a transaction execution thread are stored as parameters together with the commit statement and are used during replay. The log entries for all transaction statements are written to the database log before installing the write-set on the memory servers.

Once a memory server fails, we halt the complete system and recover all memory servers to a consistent state from the last persisted checkpoint (discussed below). For replaying the distributed log journal, the private logs of all transaction execution threads need to be partially ordered by their logged read timestamps T. Therefore, the current recovery procedure in NAM-DB is executed by one dedicated compute server that replays the merged log for all memory servers.

In order to avoid long restart phases, an asynchronous thread additionally writes checkpoints to the disks of memory servers using a dedicated read-timestamp. This is possible in snapshot isolation without blocking other transactions. The checkpoints can be used to truncate the log.

Compute Server Failures: Compute servers are stateless and thus do not need any special handling for recovery. However, a failed compute server might result in abandoned locks. Therefore, each compute server is monitored by another compute server called a monitoring compute server. If a monitoring compute server detects that a compute server is down, it unlocks the abandoned locks using the log journals written by the transaction execution threads of this compute server.

7 Evaluation

The goal of our experiments is to show that distributed transactions can indeed scale and locality is just an optimization.

Benchmark: We used TPC-C [41] as our main benchmark without any modifications unless otherwise stated for specific experiments. We generated 50 warehouses per memory server and created all required secondary indexes. All these indexes were implemented using our hash- and B^+-tree index as discussed in Section 5. Moreover, to show the effect of locality, we added a parameter to TPC-C that allows us to change the degree of distribution for new-order transactions from 0% to 100% (10% is the TPC-C specified configuration). As defined by the benchmark we only report the throughput of *new-order* transactions, which roughly make up 45% of all queries.

Setup: For executing the experiments, we used two different clusters, both with an InfiniBand network:

Cluster A has 57 machines, each with Mellanox Connect-IB card, and all connected through a single InfiniBand FDR 4X switch. The cluster contains two types of machines: the first 28 machines (type 1) have two Intel Xeon E7-4820 processors (each with 8 cores) and 128 GB RAM, the other 29 machines (type 2) have two Intel Xeon E5-2660 processors (each with 8 cores) and 256 GB RAM. All machines in this cluster run Oracle Linux Server 6.5 (kernel 2.6.32) and use the Mellanox OFED 2.3.1 driver for the network.

Cluster B has 8 machines connected to a single InfiniBand FDR 4X switch using a Mellanox Connect-IB card. 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 2.3.1 driver for the network.

7.1 Exp.1: System Scalability

To show that NAM-DB scales linearly, the number of servers were increased from 2 to 56 on Cluster A. We used two configurations of NAM-DB, with and without locality. For the setup without locality optimization, we deployed 28 memory servers on type-2 machines and 28 compute servers on type-1 machines, the latter using 60 transaction execution threads per machine. For the setup with the locality optimization, we deployed 56 compute and 56 memory servers (one pair per physical machine). In this deployment, each compute server was running only 30 transaction execution threads to have the same total number in both deployments. Finally, in both deployments we used one additional dedicated memory server on a type-2 machine to store the timestamp vector.

Figure 4 shows the throughput of NAM-DB on an increasing cluster size both without exploring locality (blue) and with adding locality (purple) and compares them against a more traditional implementation of Snapshot Isolation (red) with two-sided message-based communication. The results show that **NAM-DB scales nearly linearly** with the number of servers to 3.64 million distributed transactions over 56 machines. However, if we allow the system to take advantage of locality, we achieve 6.5 million TPC-C new-order transactions. This is 2 million more transactions than the current scale-out record by Microsoft FaRM [14], which achieves 4.5 million TPC-C transactions over 90 machines with comparable hardware and using as much locality as possible. It should be noted though that FaRM was deployed on a cluster with ConnectX-3 NICs, not ConnectIB, which can have an performance impact if the number of queue pairs is large [21]. However, as Section 7.5 will show, for TPC-C this should make almost no difference. Furthermore, FaRM implements serializability guarantees, whereas NAM-DB supports snapshot isolation. While for this benchmark it makes no difference (there is no write-skew), it might be important for other workloads. At the same time, though, FaRM never tested their system for larger read queries, for which it should perform particularly worse as it requires a full read-set validation.

The traditional SI protocol in Figure 4 follows a partitioned shared-nothing design similar to [26] but using 2-sided RDMA for the communication. As the figure shows, this design does not scale with the number of servers. Even worse, the throughput even degrades when using more than 10 machines. The degrade results from the high CPU costs of handling messages.

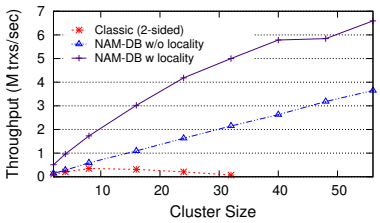


Figure 4: Scalability of NAM-DB

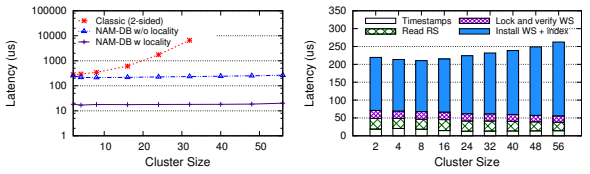
Figure 5(a) shows that the latency of new-order transactions. While NAM-DB almost stays constant regardless of the number of machines, the latency of the classic SI implementation increases. This is not surprising; in the NAM-DB design the work per machine is constant and is not related to the number of machines in the cluster, whereas the classical implementation requires more and more message handling.

When looking more carefully into the latency of NAM-DB w/o locality and its break-down (Figure 5(b)), it reveals that the latency increases slightly mainly because of the overhead to install new versions. In fact, we know from profiling that NAM-DB for TPC-C is **network bandwidth bound**. That is also the main reason why locality improves the overall performance, and we speculate that a system with the next generation of network hardware, such as EDR, would be able to achieve even higher throughputs. Finally, Figure 5(b) shows, that the latency for the timestamp oracle does not increase, indicating the efficiency of our new technique (note that we currently do not partition the timestamp vector).

7.2 Exp.2: Scalability of the Oracle

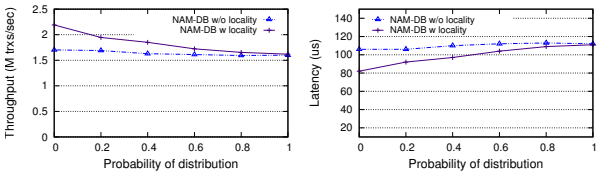
To test the scalability of our novel timestamp oracle, we varied the number of compute servers that concurrently update the oracle. As opposed to the previous experiment, however, compute servers do not execute any real transaction logic. Instead, each compute server thread executes the following three actions in a closed loop: (1) reads the current timestamp, (2) generates a new commit timestamp, and (3) makes the new commit timestamp visible. We call this sequence of operations a *timestamp transaction*, or simply *t-trx*. For this experiment, we used cluster B with eight nodes. The compute servers were deployed on seven machines, where we scaled the number of threads per machine. The remaining one node runs a memory server that stores the timestamp vector.

As a baseline, we analyze the original timestamp oracle of our vision paper [10] (red line in Figure 7), which only achieved up to 2 million t-trxs/sec. As shown in the graph, our old oracle did not scale. In fact, when scaling to more than 20 clients, the throughput starts to degrade due to high contention.However, it should be noted that for smaller clusters, the threshold of 2 million t-trxs/sec might be enough. For example, in our vision paper [10], we executed a variant of TPC-W on a smaller cluster and achieved up to 1.1 million transactions per second; a load that the original oracle could sustain. However, it becomes a bottleneck for larger deployments. As shown before, our system can execute up to 14 million transactions on 56 nodes (6.5 million new-order transactions). This load could not be handled by the classic oracle.



(a) Latency (b) Breakdown for NAM-DB

Figure 5: Latency and Breakdown



(a) Throughput (b) Latency

Figure 6: Effect of Locality

As shown in Figure 6, the new oracle (blue line) can easily sustain the above mentioned load. For example, the basic version with no optimization achieves 20 million t-trxs/sec. However, the basic version still does not scale linearly because the size of the timestamp vector grows with the number of transaction execution threads (i.e., clients) and makes the network bandwidth of the timestamp server the main bottleneck.

While 20 million t-trxs/sec is already sufficient that the basic new oracle (blue line) does not become a bottleneck in our experiments, we can push the limit even further by applying the optimizations discussed in Section 4.2. One of the optimizations is using a dedicated background fetch thread per compute server (instead of per transaction execution thread) to read the timestamp vector periodically. This reduces the load on the network. When applying this optimization (black line, denoted by "bg ts reader"), the oracle scales up to 36 million t-trxs/sec. Furthermore, when applying compression (green line), where there is one entry in the timestamp vector per machine (instead of per transaction thread), the oracle scales even further to 80 million t-trxs/sec. Finally, when enabling both optimizations (yellow line), the oracle scales up to 135 million t-trxs/sec on only 8 nodes.

It is worth noting that even the optimized oracle reaches its capacity at some point when deployed on clusters with hundreds or thousands of machines (we speculate that with these two optimizations and the given hardware we could support a cluster size of 500 machines for TPC-C). At that point, the idea of partitioning the timestamp vector (see Section 4.2) could be applied to remove the bottleneck. Therefore, we believe that our proposed design for the timestamp oracle is truly scalable.

7.3 Exp.3: Effect of Locality

As described earlier, we consider locality an optimization technique, like adding an index, rather than a key requirement to achieve good scale-out properties. This is feasible with high-speed networks since the impact of locality is no longer as severe as it is on slow networks. To test this assumption, we varied the degree of distribution for new-order transactions from 0% up to 100%. The degree of distribution represents the likelihood that a transaction needs to read/write data from/to a warehouse that is stored on a remote server When exploiting locality, transactions are executed at those servers that store the so-called home warehouse. In this experiment, we only executed the new-order transaction and not the complete mix in order to show the direct effect of locality.

For the setup, we again use cluster B with one server acting as the timestamp oracle and the remaining seven machines physically co-locating one memory and one computer server each. The TPC-C database contained 200 warehouses partitioned to all memory servers. When running w/o locality, we executed all memory accesses using RDMA. When running w/ locality, we directly accessed the local memory if possible. Since our HCA's atomic operations are not atomic with respect to the attached processor, all the atomic operations were issued as RDMA atomics, even in locality mode.

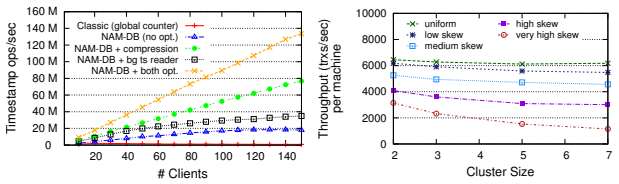
Figure 6 shows that the performance benefit of locality is roughly 30% in regard to throughput and latency. While 30% is not negligible, it still demonstrates that there are no longer orders-of-magnitude differences between them if the system is designed to achieve high distributed transaction throughput.

We also executed the same experiment on a modern in-memory database (H-Store [22]) that implements a classical shared-nothing architecture which is optimized for data-locality. We choose H-Store as it is one of the few freely available transactional in-memory databases. We used the distributed version of H-Store without any modifications using IP over InfiniBand as communication stack. Overall, we observed that H-Store only achieves 11K transactions per second (not shown in Figure 6) on a perfectly partitionable workload. These numbers are in line with the ones reported in [33]. However, at 100% distributed transactions the throughput of H-Store drops to 900 transactions per second (which is approx. a 90% drop), while our system still achieves more than $1.5M$ transactions under the same workload. This clearly shows the sensitivity of the shared-nothing design to data locality.

7.4 Exp.4: Effect of Contention

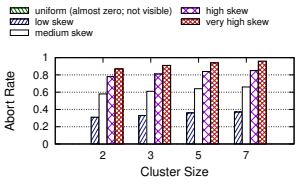
As mentioned earlier, the scalability is influenced by the intrinsic scalability of the workload. In order to analyze the effect of contention on the scalability, we increased the number of machines with different levels of contention. That is, we varied the likelihood that a given product item is selected by a transaction by using a uniform distribution as well as different zipf distributions with *low skew* ($\alpha = 0.8$), *medium skew* ($\alpha = 0.9$), *high skew* ($\alpha = 1.0$) and *very-high skew* ($\alpha = 2.0$).

Figure 8 shows the results in regard to throughput and abort rate. For the uniform and zipf distribution with low skew, we can see that the throughput per machine is stable (i.e., almost linearly as before). However, for an increasing skewness factor the abort rate also increases due to the contention on a single machine. This supports our initial claim that while RDMA can help to achieve a scalable distributed database system, we can not do something against an inherently non-scalable workload that has individual contention points. The high abort rate can be explained by the fact that we immediately abort transactions instead of waiting for a lock once a transaction does not acquire a lock. It is important to note that this does not have a huge impact on the throughput, since in our case the compute server directly triggers a retry after an abort.



(a) Throughput

Figure 7: Scalability of Oracle



(b) Abort Rate

Figure 8: Effect of Contention

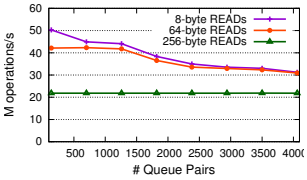


Figure 9: Scalability of QPs

7.5 Exp.5: Scalability of RDMA QPs

Recent work [21] pointed out that a high number of queue pairs (QPs) per NIC limit the scalability. The reason is that the NIC cache is limited. Thus, a high number of QPs may overflow the cache, potentially causing the performance to degrade. To investigate the impact of the number of QPs on our design, we dedicated one machine as server and seven machines as clients on Cluster B and varied the number of queue pairs per client thread while running 20 threads per client. That way, we scaled the number of total queue pairs to approximately 4000. Moreover, at every round, a client thread chooses one of its queue pairs randomly, and issues a fixed-size READ to the server.

Figure 9 shows the total number of performed operations for three different sizes of one-sided RDMA READs. The results show that queue pairs indeed have an impact on the overall performance, but mainly for small messages. For example, we observed a 40% (25%) drop in the throughput of 8-byte (64-byte) READs. However, with 256-byte READs, the number of queue pairs has almost no impact. In this case, the network bandwidth limits the maximum throughput, not the queue pairs. Thus, we argue that for many workloads (as well as benchmarks such as TPC-C) the number of queue pairs does not play an important role.

Also note that queue pairs are not needed to be established between all cores. For example, in the NAM architecture only queue pairs between servers and client-threads are required. This can be further reduced by not using a queue pair for every client-thread (as currently done in NAM-DB) but rather a few dedicated "communicator" threads at the potential cost of additional coordination overhead.

Finally, while in its simplest case, a queue pair is needed per core to enable RDMA and thus the increasing number of cores per CPU might again become a bottleneck, we observe that – at least currently – the cache sizes of NICs increase much faster than the number of cores per CPU. If this trend continues, this might further mitigate the problem in the future.

8 Related Work

Most related to our work is FaRM [14, 13]. However, FaRM uses a more traditional message-based approach and focuses on serializability, whereas we implemented snapshot isolation, which is more common in practice because of its low-overhead consistent reads. More importantly, in this work we made the case that distributed transactions can now scale, whereas FaRMs design is centered around locality.

FaSST [21] is another related project which was published while this paper was under review. Like NAM-DB, FaSST also focuses on scalability but the authors took a different approach by building an efficient RPC abstraction on top of 1-to-many unreliable datagrams using two-sided SEND/RECEIVE verbs. This design minimizes the size of queue pair state stored on NIC cache with the goal of better scalability with the size of cluster. However, as Section 7.5 showed, for many realistic workloads and cluster sizes, the number of queue pairs may not be that influential on performance. Due to their decision of abandoning one-sided RDMA verbs in favor of unreliable datagrams, their system is not able to take full advantage of leveraging the NIC as co-processors to access remote memory, and the design is likely more sensitive to data locality. Finally, and most importantly, FaSST implements serializability guarantees, whereas we show how to scale snapshot isolation, which provides better performance for read-heavy workloads and is more common in practice than serializability (e.g., Oracle does not even support it).

Another recent work [29] is similar to our design since it also separates storage from compute nodes. However, instead of treating RDMA as a first-class citizen, they treat RDMA as an afterthought. Moreover, they use a centralized commit manager to coordinate distributed transactions, which is likely to become a bottleneck when scaling out to larger clusters. Conversely, our NAM-DB architecture is designed to leverage one-sided RDMA primitives to build a scalable shared distributed architecture without a central coordinator to avoid bottlenecks in the design.

Industrial-strength products have also adopted RDMA in existing DBMSs [6, 36, 28]. For example, Oracle RAC [36] has RDMA support, including the use of RDMA atomic primitives. However, RAC does not directly take advantage of the network for transaction processing and is essentially a workaround for a legacy system. Furthermore, IBM pureScale [6] uses RDMA to provide high availability for DB2 but also relies on a centralized manager to coordinate distributed transactions. Finally, 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.

Other projects in academia have also targeted RDMA for data management, such as *distributed join processing* [7, 38, 19]. However, they focus mainly only on leveraging RDMA in a traditional shared-nothing architecture and do not discuss the redesign of the full database stack. SpinningJoins [19] suggest a new architecture for RDMA. Different from our work, this work assumes severely limited network bandwidth (only 1.25GB/s) and therefore streams one relation across all the nodes (similar to a block-nested loop join). Another line of work is on *RDMA-enabled key value stores* RDMA-enabled key/value stores [31, 30, 20]. We leverage some of these results to build our distributed indexes in NAM-DB, but transactions and query processing are not discussed in these papers.

Furthermore, there is a huge body of work on distributed transaction processing over slow networks. In order to reduce the network overhead, many techniques have been proposed ranging from locality-aware partitioning schemes [35, 33, 12, 43] and speculative execution [32] to new consistency levels [24, 4, 3] and the relaxation of durability guarantees [25].

Finally, there is also recent work on high-performance OLTP systems for many-core machines [23, 42, 34]. This line of work is largely orthogonal to ours as it focuses on scale-up rather than scale-out. However, our timestamp oracle could be used in a scale-up solution to achieve better scalability. In addition, it should be noted that our current system is able to achieve a quarter of the performance of the current scale-up record [42]: $\approx 59Ktps/core$ vs $\approx 16.8Ktps/core$ (only counting new-order transactions). This is an impressive result as it is generally easier and more cost-efficient to incrementally scale-out than scale-up a system. Furthermore, future increases in the available network bandwidth is likely to close the gap (recall that the bandwidth of dual-port FDR 4x is roughly a quarter of the memory bandwidth and our system is bandwidth bound).

9 Conclusions

We presented NAM-DB, a novel scalable distributed database system which uses distributed transactions by default and considers locality as an optimization. We further presented techniques to achieve scalable timestamps for snapshot isolation, as well as showed how to implement Snapshot Isolation using one-sided RDMA operations. Our evaluation shows nearly perfect linear scale-out to up to 56 machines and a total TPC-C throughput of 6.5 million transactions per second, significantly more than the state-of-the-art. In the future, we plan to investigate more into avenues such as distributed index design for RDMA and to study the design of other isolation levels in more detail. This work ends the myth that distributed transactions do not scale and shows that NAM-DB is at most limited by the workload itself.

10 References

[1] M. Armbrust et al. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5(3):181–192, 2011.

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

[3] P. Bailis et al. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[4] P. Bailis et al. Scalable atomic visibility with ramp transactions. *ACM Transactions on Database Systems (TODS)*, 41(3):15, 2016.

[5] D. Barbará-Millá et al. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *VLDB Journal*, 3(3):325–353, 1994.

[6] V. Barshai et al. *Delivering Continuity and Extreme Capacity with the IBM DB2 pureScale Feature*. IBM Redbooks, 2012.

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

[8] H. Berenson et al. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[9] C. Binnig et al. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *VLDB Journal*, 23(6):987–1011, 2014.

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

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

[12] C. Curino et al. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1-2):48–57, 2010.

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

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

[15] G. Eadon et al. Supporting table partitioning by reference in oracle. In *Proc. of ACM SIGMOD*, pages 1111–1122, 2008.

[16] S. Elnikety et al. Database replication using generalized snapshot isolation. In *IEEE SRDS 2005*, pages 73–84, 2005.

[17] J. M. Faleiro et al. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015.

[18] A. Fekete. Snapshot isolation. In *Encyclopedia of Database Systems*. Springer US, 2009.

[19] P. W. Frey et al. A spinning join that does not get dizzy. In *Proc. of ICDCS*, pages 283–292, 2010.

[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] R. Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[23] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proc. of ACM SIGMOD*, pages 691–706, 2015.

[24] T. Kraska et al. Consistency rationing in the cloud: pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[25] V. Krishnaswamy et al. Relative serializability: An approach for relaxing the atomicity of transactions. *J. Comput. Syst. Sci.*, 55(2):344–354, 1997.

[26] J. Lee et al. High-performance transaction processing in sap hana. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

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

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

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

[30] 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.

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

[32] A. Pavlo et al. On predictive modeling for optimizing transaction execution in parallel oltp systems. *PVLDB*, 5(2):85–96, 2011.

[33] A. Pavlo et al. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proc. of ACM SIGMOD*, pages 61–72, 2012.

[34] D. Porobic et al. Characterization of the impact of hardware islands on oltp. *VLDB Journal*, 25(5):625–650, 2016.

[35] A. Quamar et al. SWORD: scalable workload-aware data placement for transactional workloads. In *Proc. of EBDT*, pages 430–441, 2013.

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

[37] K. Ren et al. Lightweight locking for main memory database systems. *PVLDB*, 6(2):145–156, 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] M. Stonebraker and U. Cetintemel. "one size fits all": an idea whose time has come and gone. In *Proc. of ICDE*, pages 2–11, 2005.

[40] A. Thomson et al. The case for determinism in database systems. *PVLDB*, 3(1-2):70–80, 2010.

[41] TPC-C. http://www.tpc.org/tpcc/.

[42] T. Wang et al. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, 2016.

[43] E. Zamanian et al. Locality-aware partitioning in parallel database systems. In *Proc. of ACM SIGMOD*, pages 17–30, 2015.