The End of a Myth: Distributed Transactions Can Scale

Erfan Zamanian[1] Carsten Binnig[1] Tim Kraska[1] Tim Harris[2]
[1] Brown University [2] Oracle Labs
{erfan_zamanian_dolati, carsten_binnig, tim_kraska}@brown.edu timothy.l.harris@oracle.com

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 be no need for developers anymore 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 [44, 23, 43, 12, 34, 40]. As a result, many techniques have been proposed to avoid distributed transactions ranging from locality-aware partitioning [38, 36, 12, 48] and speculative execution [35] to new consistency levels [25] and the relaxation of durability guarantees [26]. 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 a relatively unpredictable performance [9]. For other application, e.g., social apps, a developer might even be unable 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 redesign 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 very 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 (more on that later). 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 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 scalable. One of the most cited reasons is the increased contention likelihood. However, contention is only a side effect. Maybe 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, causing 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 distribution 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 Why we need a System Redesign

Assuming the workload is scalable, the next generation of networks remove the two dominant limiting factors to make distributed transaction scale: 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, the design of some database components along with their data structures have to change. 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 data can be directly accessed via RDMA reads and writes. Similarly, the transaction protocols need to be redesigned to avoid inherent bottlenecks.

While there has been work on leveraging RDMA for distributed transactions, most notably FaRM [13, 14], most works still heavily 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 rather on serializability [14] or do not have transaction support at all [22]. 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 how transactional and analytical database systems have to change and showed the potential of taking advantage of efficiently leveraging high-speed networks and RDMA. In this paper, we follow up on this vision and present and evaluate one of the very first transactional systems for high-speed networks and RDMA. In summary, we make 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 here presented design does no longer restrict the 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 to efficiently read the (latest) consistent snapshot in a distributed SI-based protocol. One of the main limitations of our prototype in [10]. (3) We show that NAM-DB is truly scalable using a full implementation of the TPC-C benchmark including additional variants where we vary factors such as degree of distribution as well as the contention rate. Most notably, for the standard configuration of the 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, that is 2 million more transactions per second than what FARM [14] achieves on 90 machines. Note, that the total TPC-C throughput would be even higher (14.5 million transactions per second) as TPC-C specifies to only report the new-order transactions.

The remainder of the paper is organized as follows: In Section 2 we give an overview of our novel RDMA-based architecture for distributed transaction processing and describe the basic RDMA-based SI protocol as implemented in [10]. Afterwards, we discuss the main contributions of this paper. First, in Section 4 we explain the design of our new timestamp oracle to generate read and commit timestamps in a scalable manner. In Section 5 and Section 6, we explain how data is stored in the memory servers and how transactions are executed and persisted. Afterwards, in Section 7, we present the results of our experimental evaluation using the TPC-C benchmark. Finally, we conclude with related work in Section 8 and an outlook on future avenues in Section 9.

2 System Overview

In our previous work [10], we showed that the distributed database architecture has to radically change to make the most efficient use of fast networks such as InfiniBand.

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. As with Ethernet-based networks, data is copied into OS buffers and the kernel processes them by sending packets over the network, resulting in high CPU overhead and therefore high latencies. While IPoIB provides an easy migration path from Ethernet to InfiniBand, IPoIB cannot fully leverage the network's capabilities [10]. On the other hand, RDMA provides a *verbs* API, which enables remote data transfer using the processing capabilities of an RDMA NIC (RNIC). When using RDMA verbs, most of the processing is executed by the RNIC without OS involvement, which is essential for achieving low latencies. The

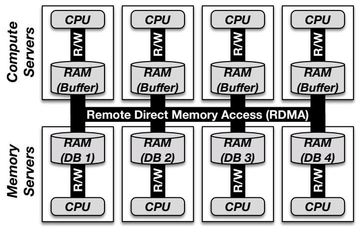


Figure 1: The NAM Architecture

verbs API offers two classes of operations: 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. Readers interested in learning more about RDMA are encouraged to read Section 2 in [10]. Redesigning distributed databases to efficiently make use of RDMA is a key challenge that we tackle in this paper.

In this section, we first give a brief overview of the network-attached-memory (NAM) architecture proposed in [10] that was designed 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. In the following, we give an overview of the tasks of memory servers and compute servers in a NAM architecture.

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 as well 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 on the location of the data.

2.2 Design Principles

In the following, we discuss the central design principles of NAM-DB to enable a truly scalable database system for transactional workloads that builds on the NAM architecture:

Separation of Compute and Memory: Although the separation of compute and storage is not new [**?**, 28, 30, 14], existing database systems that follow this design typically push data access operations into the storage layer. However, when scaling out the compute serves 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 server 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, instead of making data locality the first design consideration in a distributed database architecture, our main goal is to achieve scalability and to avoid bottlenecks in the system design. Therefore, we argue that data locality is just an optimization that can be added on top of our scalable system.

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 decentralized data structure to implement a partitionable read and commit timestamp as shown in Section 4.

3 The Basic SI-Protocol

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

3.1 A Naïve RDMA Implementation

With Snapshot Isolation (SI), a transaction reads the most recent snapshot of a database that was committed before the begin of the transaction and it does not see concurrent updates. In contrast to serializability, a transaction with SI guarantees is only aborted if it wants to update an item which was written since the beginning of the transaction. 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 according Figure 2 show a naïve SI protocol using only one-sided RDMA operations that is based on a global timestamp oracle as implemented in commercial systems [9]. The signature of the RDMA operations used in Listing 1 is given in the following table.

Operation	Signature
RDMA_Read	(remote_addr)
RDMA_Write	(remote_addr, value)
RDMA_Send	(value)
RDMA_FetchAndAdd	(remote_addr, increment)
RDMA_CompAndSwap	(remote_addr, check_value, new_value)

To better focus on the interaction between compute and memory servers, we made the following simplifying assumptions: First, we will not consider the durability guarantee and the recovery of transactions. Second, we assume that there is a catalog service in place, which is always kept up-to-date and helps compute servers find the remote address of a data item in the pool of memory servers; the remote address of a data item in a memory server is simply 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 and thus we do not need to tackle garbage collection of old versions. **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

```
 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

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). 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 give every timestamp a fixed position (i.e., timestamp value - offset) to set a single bit to indicate the success of a transaction. This is possible, as the

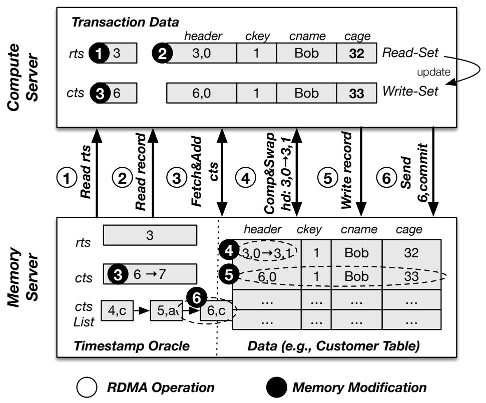


Figure 2: Naïve RDMA-based SI-Protocol

fetch and add operation creates continuous timestamp values.

Finally, the **timestamp oracle** is responsible to advance the read timestamp by scanning the queue of completed transactions. It therefore scans the queue and tries to find the highest commit timestamp (i.e., highest bit) so that every transactions before the 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 adjust 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 transactions, it shows its negative impact when hundreds of thousands 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 $ctsList$ causing that the read timestamp does not advance anymore.

Note that beside all these problems, we have made a few simplifying assumptions regarding durability, recovery, metadata management, version maintenance and garbage collection. In the remaining sections, we will address these open issues and the problems outlined before.

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 to further increase the scalability.

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 to advance 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 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 that we use a data structure called the **timestamp vector** similar to a vector clock, that 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 only the timestamp of the compute server who 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 $t+1$ 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 to the transaction. If not, an older version has to be used so that the condition holds true. 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 that one transaction execution thread in a compute server sees a timestamp vector like $\langle .., t_n, .., t_m + 1, .. \rangle$ whereas another observes $\langle .., t_n + 1, .., t_m, .. \rangle$. As a result the timestamps are still progressing monotonically, like with 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.

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 all transaction threads 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, we realized that due to the reduced network load, the runtime of each transaction is heavily reduced, leading to actually a lower abort rate.

Compression of T_R: The size of T_R currently depends on the number of transaction execution threads, that could rise up to hundreds or even thousands entries when scaling out. Thus, instead of having one slot per transaction execution 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 the two optimizations above are already sufficient to scale to at least 56 nodes. However, storing T_R on one memory server could at some point make the network bandwidth of this server the bottleneck. Fortunately, as a transaction execution thread only needs to update a single slot t_i, it is easy to partition T_R across several memory nodes. 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 decision 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. For the data section, we currently 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

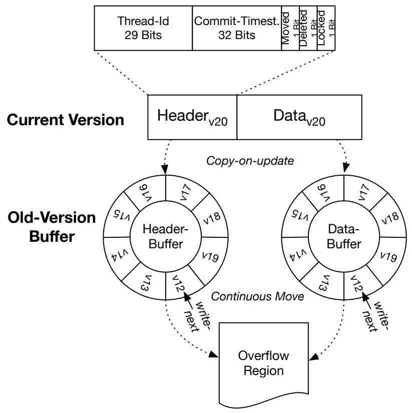


Figure 3: Version Management and Record Layout

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 collected. 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 then 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 store an extra counter which is called the next-write counter.

Another issue is that the circular buffers have only a fixed capacity. The reason that we efficiently want to access them with one-sided RDMA operations and avoid pointer chasing operations. However, in order to support long-running transactions or slow compute servers the number of versions stored in the buffer might not be sufficient since they might need to read older versions. As a solution, a version-mover thread that runs in a memory server continuously moves the header and data section of the old-version buffers to an overflow region and sets the moved-bit in the header-buffer to 1. This does not actually mean that the header and data section are removed from the old-versions buffers. It only means that it can be safely re-used by a transaction that installs 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 [31]. 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 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 as well as three pointers (two pointers to the old-version buffers as well as one pointer to the overflow region).

Different from [31], 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. Retrieving the correct version of a record is implemented by the subsequent lookup on the table structure using the primary key that is returned by the lookup on the secondary index.

In order to implement the hash index in NAM-DB, we use the same hash table design that we have described before that is used for implementing tables. The main difference is that 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 server. The reason is that operations in B^+-trees need to chase multiple pointers from the root to the leave level. When implementing pointer chasing with one-sided RDMA operations, multiple roundtrips between compute and memory servers are required. This is also true for a linked list in a hash table. However, as shown in [31] when clustering keys in a linked list into one memory region one RDMA read / write is often sufficient to execute a get / put an entry if no collision occurs. Moreover, for scaling-out and to avoid that individual memory servers become a bottleneck, we range partition B^+-trees to different memory servers. In the future, we plan to investigate into 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 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 who executed the transaction allocates an extend that allows to store 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: In order to avoid that tables constantly grow and need to allocate more and more space from memory servers, old versions of records need to be garbage collected. The goal of garbage collection is to find out which versions of a record are not required anymore and 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.

In order to implement our garbage collection scheme we capture a snapshot of the timestamp vector T that represents the read timestamp of the timestamp oracle (see Section 4) in regular intervals. We currently create a snapshot of T every minute and store it together with the wall-clock time in a list sorted by the wall-clock time. That way, we can find out which versions can be safely garbage collected based on the maximal transaction execution time. For garbage collecting these versions, a garbage collection thread runs on every memory server 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 first discuss how compute servers execute transactions and then present techniques for logging and recovery as well as 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 read timestamp as well as $(i, T[i])$ as 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 to often, the catalog data is cached by compute servers and refreshed in two cases: The first case is, if a requested database object is not found in the cached catalog, 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 store 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 tolerates failures of 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. Handling these types of failures, could be handled by 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 a 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 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: As benchmark, we used TPC-C [45]. We used the standard schema and configuration of TPC-C without any modifications unless otherwise stated. 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, for showing 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% (where 10% is the standard configuration). As defined by the benchmark we only report the throughput of *new-order* transactions, which can make up to 45% of the benchmark.

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

Cluster A: This is a cluster at Oracle that is composed of 57 machines (two types of machines) in total that are all connected to an InfiniBand FDR 4X network using a Mellanox Connect-IB card. All machines are attached to the same InfiniBand switch. While the first 28 machines of type 1 provide two Intel Xeon E7-4820 processors (each with 8 cores) and 128 GB RAM, the other 29 machines of type 2 provide 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) as operating system and use the Mellanox OFED 2.3.1 driver for the network.

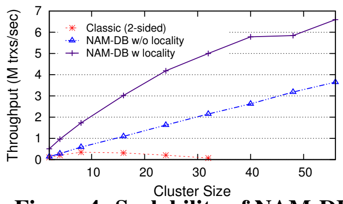


Figure 4: Scalability of NAM-DB

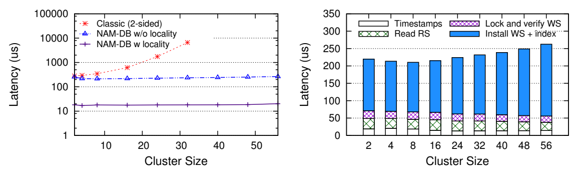
Cluster B: This is our own (smaller) InfiniBand cluster with 8 machines that are all connected to an InfiniBand FDR 4X network using a Mellanox Connect-IB card. All machines are attached to the same InfiniBand switch. Moreover, each machine has two Intel Xeon E5-2660 v2 processors (each with 10 cores) and 256GB RAM. The machines all run Ubuntu 14.01 Server Edition (kernel 3.13.0-35-generic) as operating system and use the Mellanox OFED 2.3.1 driver for the network.

For showing the scalability of NAM-DB, we used Cluster A. However, since we only had restricted access to that cluster, we executed the more detailed analysis (e.g., the effect of data locality) on Cluster B.

7.1 Exp.1: System Scalability

The main goal of this experiment is to show the scalability of NAM-DB for TPC-C on Cluster A. We therefore increased the number of servers starting with 2 servers and scaled-out to 56 servers and executed NAM-DB with and without its locality optimization. For the setup without locality optimization, we deployed 28 compute servers on type 1 machines and 28 memory servers on type 2 machines. Each compute server uses 60 transaction execution threads in this deployment. 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 with an increasing cluster size without exploring locality (blue), with adding locality (purple) and compares them against a more traditional implementation of Snapshot Isolation (red) with two-sided messaged based communication. The results show that **NAM-DB scales nearly linear** with the number of servers to 3.64 million distributed transactions over 56 machines. This is a stunning result as in this configuration all transactions are distributed. However, if we allow the system to take advantage of locality we achieve even 6.5 million TPC-C new-order transactions (recall, in the default setting roughly 10% have to be distributed and the new-order transaction makes only up to 45% of the workload). This is 2 million more TPC-C transactions than the current scale-out record by Microsoft FaRM [14], which achieves 4.5 million TPC-C transactions over 90 machines with very comparable hardware, the same network technology and using as much locality as possible. It should be noted though, that FaRM implements serializability guarantees, whereas NAM-DB supports snapshot isolation. While



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

Figure 5: Latency and Breakdown

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 particular bad as it requires a full read-set validation.

The traditional SI protocol in Figure 4 follows a partitioned shared-nothing design similar to [27] but using two-sided RDMA for the communication. For this variant, we clearly see that it does not scale with the number of servers. Even worse, the throughput even degrades when using more than 10 machines. This degrade results from the high CPU costs of handling messages. This effect can also be seen by the increased latency.

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

Looking more carefully into the latency of NAM-DB w/o locality and breaking the latency down into the different components (Figure 5(b)) reveals that the latency increases slightly mainly because of the overhead to install new version. Most interestingly though the latency for the timestamp oracle does not increase, indicating the efficiency of our technique (note, that we currently do not partition the timestamp vector).

Finally, another interesting question is how NAM-DB compares to the initial prototype in [10]. Since the initial prototype makes many simplifying assumptions on central components (e.g., storage management), we compare NAM-DB when using the old and new timestamp oracle to exclude effects resulting from these simplifications. While NAM-DB with the new oracle scales linearly up to 56 nodes as shown in Figure 4, NAM-DB with the old oracle (not shown in the Figure 4) provides only a linear scalability up to 10 nodes and achieves a maximum throughput of approximately 0.5 million transactions per second. When scaling out further, the throughput however decreases to only 100.000 transactions per second on 56 nodes resulting from the contention on the old timestamp oracle; an effect that we show next.

7.2 Exp.2: Scalability of the Oracle

In this experiment, we analyze the scalability of our new timestamp oracle in isolation, which is one of the main contributions of this paper. For testing the scalability, we vary the number of compute servers that concurrently update the oracle. Different from the prevoius experiment, however, compute servers do not execute any real transaction logic. Instead, each thread in a compute server executes the following actions in a closed loop to put a high load on the oracle: first it reads the current timestamp, second it generates a new commit

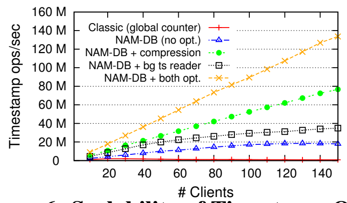


Figure 6: Scalability of Timestamp Oracle

timestamp and then it makes the new commit timestamp visible. This sequence of operations is called a *timestamp transaction* in the sequel, or simply *t-trx* . In this experiment, we use cluster B with 8 servers in total. Each node hosts one compute server that runs 20 transaction execution threads whereas one node runs a memory server that stores the timestamp vector.

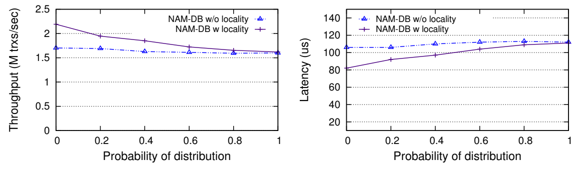
As a baseline, we analyze the original timestamp oracle of our vision paper [10]. This timestamp oracle implements a classical design with one globally-ordered counter, which is accessed concurrently via RDMA operations as described in Section 3 (the red line in Figure 6). This oracle can achieve up to 2 million t-trxs/sec. However, it does not scale at all with an increasing number of clients. In fact, when scaling to more than 20 clients, the throughput even starts to degrade due to high contention on the oracle. However, while not scalable for large deployments, the original oracle design was still sufficient for our prototype in [10]. The reason is that real transactions execute other operations in between reading and updating the global timestamp counter. As a result, the maximal throughput in [10] was at most 1.1 million TPC-W transactions per second, a load that the original oracle could sustain.

In this paper, however, the main goal is to provide a truly scalable oracle for much larger deployments. As presented before, when running the full mix of TPC-C transactions, our system can execute up to 6.5 million new-order transactions on 56 nodes. In other words, it can execute more than 14 million transactions (all TPC-C transactions together); a load that could not be handled by the classic oracle design (red line).

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

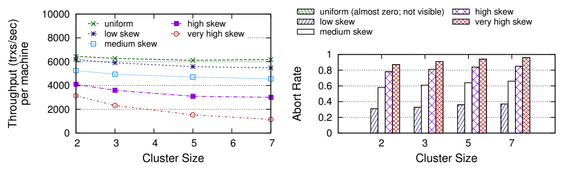
While 20 million t-trxs/sec are already sufficient that the basic new oracle (blue line) does not become a bottleneck in our experiments, we can push the limit even further by applying the optimizations discussed in Section 4.2. One of the optimizations is that one dedicated background fetch thread is used per compute server (instead of per transaction execution thread) to read the timestamp vector periodically in order to reduce 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 using the idea of 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



(a) Throughput (b) Latency

Figure 7: Effect of Locality



(a) Throughput (b) Abort Rate

Figure 8: Effect of Contention

capacity at some point when deployed on clusters with hundreds or thousands of machines. 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 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 requirement to achieve good scale-out properties. This is feasible as with high-speed networks the impact of locality is no longer as severe as 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 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 our cluster B with 8 servers. One server was dedicated for the timestamp oracle. All other 7 machines physically co-locate one memory and one computer server. 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 accessed directly the local memory if possible.

Figure 7 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 no longer are 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 [34]) 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 distributed in-memory transactional database systems. We used the distributed version of H-Store without any modifications over InfiniBand (IB) using IP over IB as communication stack. As a main result, we found out that H-Store has an overall much lower throughput than NAM-DB. On a perfectly partitionable workload (i.e., with no distributed transactions), H-Store reaches only 11K transactions per second (not shown in Figure 7 since the line would not be visible on the given y-scale). These numbers are in line with the ones reported in [36]. However, at 100% distributed transactions the throughput of H-Store drops to only 900 transactions per second, a 90% drop in its performance, while our system still achieves more than $1.5M$ transactions under the same workload; a small performance drop compared to its no-distributed-transaction case (i.e., 0% distributed transactions). This clearly shows the sensitivity of the shared-nothing design to data locality.

7.4 Exp.4: Effect of Contention

In this final experiment, we analyze the effect of contention on the throughput and abort rate with an increasing number of servers and w/o locality (so all transactions are distributed) We vary the skew in the workload; i.e., 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 us 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 about transactions instead of waiting for a lock once a transaction does not acquire a lock. Important is 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.

8 Related Work

In this paper, we have made the case that distributed transactions can indeed scale using the recent generation of RDMA-enabled networks technology.

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.

Another recent work [30] 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 adapted RDMA in existing DBMSs [6, 39, 29]. For example, Oracle RAC [39] 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 [29] 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 recently targeted RDMA for data management, such as *distributed join processing* [7, 42, 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 [33, 31, 22]. 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 [38, 36, 12, 48] and speculative execution [35] to new consistency levels [25, 4, 3] and the relaxation of durability guarantees [26].

Finally, there is also recent work on high-performance OLTP systems for many-core machines [24, 47, 37]. While the main focus of these papers is on scaling-up on a single machine, the focus of this paper is on the aspects of scale-out when leveraging the next generation of high-speed networks and RDMA and thus can be seen as orthogonal to the other work.

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 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] Michael Armbrust et al. PIQL: Success-Tolerant Query Processing in the Cloud. *PVLDB*, 2011.

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

[3] Peter Bailis et al. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 2013.

[4] Peter Bailis et al. Scalable atomic visibility with RAMP transactions. *ACM Trans. Database Syst.*, 2016.

[5] Daniel Barbará et al. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *VLDB J.*, 1994.

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

[7] Claude Barthels et al. Rack-scale in-memory join processing using RDMA. In *ACM SIGMOD*, 2015.

[8] Hal Berenson et al. A Critique of ANSI SQL Isolation Levels. In *ACM SIGMOD*, 1995.

[9] Carsten Binnig et al. Distributed Snapshot Isolation: Global Transactions pay globally, Local Transactions pay locally. *VLDB J.*, 2014.

[10] Carsten Binnig et al. The End of Slow Networks: It's Time for a Redesign. *PVLDB*, 2016.

[11] Matthias Brantner et al. Building a database on S3. In *ACM SIGMOD*, 2008.

[12] Carlo Curino et al. Schism: a Workload-Driven Approach to Database Replication and Partitioning. In *PVLDB*, 2010.

[13] Aleksandar Dragojevic et al. FaRM: Fast Remote Memory. In *NSDI*, 2014.

[14] Aleksandar Dragojevic et al. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*, 2015.

[15] George Eadon et al. Supporting Table Partitioning by Reference in Oracle. In *ACM SIGMOD*, 2008.

[16] Sameh Elnikety et al. Database replication using generalized snapshot isolation. In *SRDS*, 2005.

[17] Jose M. Faleiro et al. Rethinking serializable multiversion concurrency control. *PVLDB*, 2015.

[18] Alan 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 *ICDCS*, 2010.

[20] Stavros Harizopoulos et al. OLTP through the looking glass, and what we found there. In *ACM SIGMOD*, 2008.

[21] Mark C. Jeffrey et al. A scalable architecture for ordered parallelism. In *MICRO*, 2015.

[22] Anuj Kalia et al. Using RDMA efficiently for key-value services. In *ACM SIGCOMM*, 2014.

[23] Robert Kallman et al. H-Store: a high-performance, distributed main memory transaction processing system. In *PVLDB*, 2008.

[24] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *ACM SIGMOD*, 2015.

[25] Tim Kraska et al. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2009.

[26] Vashudha Krishnaswamy et al. Relative serializability: An approach for relaxing the atomicity of transactions. *J. Comput. Syst. Sci.*, 1997.

[27] Juchang Lee et al. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.*, 2014.

[28] Justin J. Levandoski et al. High Performance Transactions in Deuteronomy. In *CIDR*, 2015.

[29] Feng Li et al. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *ACM SIGMOD*, 2016.

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

[31] Christopher Mitchell et al. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX Annual Technical Conference*, 2013.

[32] Patrick E. O'Neil. The Escrow Transactional Method. *ACM Trans. Database Syst.*, 1986.

[33] John K. Ousterhout et al. The case for RAMCloud. *Commun. ACM*, 2011.

[34] Andrew Pavlo. *On Scalable Transaction Execution in Partitioned Main Memory Database Management Systems*. PhD thesis, Brown, 2014.

[35] Andrew Pavlo et al. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB*, 2011.

[36] Andrew Pavlo et al. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *ACM SIGMOD*, 2012.

[37] Danica Porobic et al. Characterization of the impact of hardware islands on OLTP. *VLDB J.*, 2016.

[38] Abdul Quamar et al. SWORD: scalable workload-aware data placement for transactional workloads. In *EDBT*, 2013.

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

[40] Kun Ren et al. Lightweight locking for main memory database systems. In *VLDB*, 2012.

[41] Wolf Rödiger et al. High-speed query processing over high-speed networks. *PVLDB*, 2015.

[42] Wolf Rödiger et al. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *ICDE*, 2016.

[43] Michael Stonebraker et al. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, 2005.

[44] Alexander Thomson et al. The Case for Determinism in Database Systems. In *PVLDB*, 2010.

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

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

[47] Tianzheng Wang et al. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 2016.

[48] Erfan Zamanian et al. Locality-aware Partitioning in Parallel Database Systems. In *ACM SIGMOD*, 2015.