Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores

Claude Barthels[1*], Ingo Müller[1#], Konstantin Taranov[2*], Gustavo Alonso[1*], Torsten Hoefler[2*]

[1]Systems Group [2] Scalable Parallel Computing Lab

Department of Computer Science, ETH Zurich

[*]{firstname.lastname}@inf.ethz.ch [#]ingo.mueller@inf.ethz.ch

ABSTRACT

Concurrency control is a cornerstone of distributed database engines and storage systems. In pursuit of scalability, a common assumption is that Two-Phase Locking (2PL) and Two-Phase Commit (2PC) are not viable solutions due to their communication overhead. Recent results, however, have hinted that 2PL and 2PC might not have such a bad performance. Nevertheless, there has been no attempt to actually measure how a state-of-the-art implementation of 2PL and 2PC would perform on modern hardware.

The goal of this paper is to establish a baseline for concurrency control mechanisms on thousands of cores connected through a low-latency network. We develop a distributed lock table supporting all the standard locking modes used in database engines. We focus on strong consistency in the form of strict serializability implemented through strict 2PL, but also explore read-committed and repeatable-read, two common isolation levels used in many systems. We do not leverage any known optimizations in the locking or commit parts of the protocols. The surprising result is that, for TPC-C, 2PL and 2PC can be made to scale to thousands of cores and hundreds of machines, reaching a throughput of over 21 million transactions per second with 9.5 million *New Order* operations per second. Since most existing relational database engines use some form of locking for implementing concurrency control, our findings provide a path for such systems to scale without having to significantly redesign transaction management. To achieve these results, our implementation relies on Remote Direct Memory Access (RDMA). Today, this technology is commonly available on both Infiniband as well as Ethernet networks, making the results valid across a wide range of systems and platforms, including database appliances, data centers, and cloud environments.

PVLDB Reference Format:
Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *PVLDB*, 12(13): 2325-2338, 2019.
DOI: https://doi.org/10.14778/3358701.3358702

Proceedings of the VLDB Endowment, Vol. 12, No. 13
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3358701.3358702

1. INTRODUCTION

Concurrency control is an important component in many database systems. Recent publications [13, 17, 24, 36, 41, 42] have shown a renewed interest in distributed concurrency control. Many of these proposals exhibit significant differences in throughput when running on a large number of cores or machines. These systems apply a wide range of optimizations that impose restrictions on the workloads the engine can support. For example, they give up serializability in favor of snapshot isolation [42], impose restrictions on long-running transactions [13, 24, 36], assume partitioned workloads [23], or require to know the read and write set of transactions ahead of time [23, 35]. One common underlying assumption among all these approaches is that Two-Phase Locking (2PL) and Two-Phase Commit (2PC) – the primary components of a textbook implementation of a database lock manager – do not scale to hundreds of machines with thousands of processor cores.

Modern data processing systems and data centers are starting to be equipped with high-bandwidth, low-latency interconnects. This new generation of networks originates from advances in high-performance computing (HPC) systems. Similar to distributed database systems, the performance of scientific applications depends heavily on the ability of the system to efficiently access data on remote compute nodes. The features offered by these modern networks include (i) user-level networking, (ii) an asynchronous network interface that enables interleaving computation and communication, (iii) the ability of the network card to directly access the main memory without going through the processor, i.e., Remote Direct Memory Access (RDMA), and (iv) one-sided remote memory access (RMA) operations. The combination of these features enables new designs for distributed database systems [3, 28, 31, 42] and scalable algorithms such as joins [4, 5, 30].

A recent evaluation of several distributed concurrency control mechanisms suggests that a tight integration of concurrency control and modern networks is needed to scale out distributed transactions [17]. While the costs of synchronization and coordination might be significant on conventional networks, modern networks and communication mechanisms, such as RDMA, have significantly lowered these costs. In this paper, we establish a baseline for running a conventional lock manager over a state-of-the-art network and show that the low latency offered by modern networks makes a concurrency control mechanism based on 2PL and 2PC a viable solution for large-scale database systems. In light of these new network technologies, the design, implementation, and performance of distributed concurrency control mechanisms needs to be re-evaluated. This paper is the first to provide a baseline for a conventional lock manager using 2PL and 2PC and an evaluation of its behavior at large scale on several thousand processor cores.

The lock table used in this experimental evaluation supports all the conventional locking modes used in multi-level granularity locking. The system operates following a traditional design, as explained for instance in the book by Gray and Reuter [16] on transaction management. We introduce no optimizations and no restrictions on transaction structure and operations, nor presume any advance knowledge of the transactions or sequence of submission. We also do not use any pre-ordering mechanism such as an agreement protocol. By using a textbook implementation of a concurrency control mechanism, a database system does not need to compromise on the type of transactions it can execute nor on the isolation levels it can provide. Through the use of Strict 2PL, the system provides strict serializability. To ensure that distributed transactions leave the database in a consistent state, the system uses conventional 2PC [7].

The question we seek to answer is whether modern implementations of 2PL and 2PC can scale and take advantage of large parallel systems. The novelty lies not in the design of the lock table, but in how the system is implemented and evaluated: our prototype uses MPI, a de-facto standard communication layer used by many HPC applications. This enables us to perform our evaluation on a high-end supercomputer, which provides us with a large number of processor cores and a state-of-the-art network.

In the experimental evaluation, we show that, for TPC-C, our implementation can support a throughput of more than 21 million transactions per second in serializability mode with 9.5 million *New Order* transactions per second.

The key insight from the paper is that modern networking reduces the latency for remote memory accesses to values comparable to those of multi-core machines. As a consequence, acquiring a lock residing on a remote machine does not have a significant overhead compared to acquiring a local lock. This is what allows us to build a large-scale distributed lock table that provides the locking throughput necessary to maintain a high transaction throughput.

2. BACKGROUND

This section provides the necessary background on (i) low-latency, RDMA-capable networks, (ii) the Message Passing Interface, and (iii) concurrency control mechanisms used in database systems.

2.1 Low-Latency Networks

Remote Direct Memory Access (RDMA) is an emerging technology that has been shown to provide substantial benefits with respect to lowering the costs of large data transfers and thus gained the attention of several research projects [4, 5, 30, 42]. RDMA is a hardware mechanism through which the network card can directly access parts or all of main memory. Transferring data directly from and to main memory bypasses the CPU and the OS network stack, avoids intermediate copies, and, in turn, enables the network to reach a high throughput while at the same time providing low latency.

In many network implementations, the network card cannot access arbitrary sections of memory. Buffers need to be registered with the network card before they are accessible for RDMA operations. During this registration process, the memory is pinned such that it cannot be swapped out and the necessary address translation information is installed on the network card. This registration process often incurs a significant overhead [14].

Most modern networks provide two types of RDMA operations: one- and two-sided memory accesses [18]. Two-sided operations represent message-passing semantics in which two parties are involved in the communication, i.e., the sender and receiver of a message. One-sided operations represent remote memory access (RMA) semantics in which only the initiator of a request is involved in the communication. The CPU on the target node, on which the memory is accessed, is usually not interrupted and is not aware of the access happening through the network card. There is a small performance difference between both types of communication, with one-sided operations having a lower latency, but requiring more messages for complex interactions.

There are several programming abstractions for using one-sided network operations. The two most popular concepts are Remote Memory Access (RMA) and Partitioned Global Address Space (PGAS) programming. RMA provides access to remote memory regions through read and write operations, while in PGAS programs, these instructions are automatically inserted by the compiler. Read operations fetch data from a remote machine and transfer it to a local buffer, while write operations move data in the opposite direction. In addition, many RMA implementations and networks provide support for additional functionality, most notably remote atomic operations. Examples of such atomic operations are remote fetch-and-add and compare-and-swap instructions.

2.2 Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a widely used communication interface in high-performance computing systems. MPI offers a rich hardware-independent networking interface supporting a variety of networks and supercomputers. At the same time, MPI is capable of achieving good performance by binding to optimized implementations for each target machine and network.

MPI automatically chooses the most appropriate communication method based on the relative distance between two processes. For example, if two processes are located on the same machine, modern MPI implementations use shared memory data structures to communicate. For processes on different nodes, the library automatically uses network-based communication mechanisms.

Most supercomputers and distributed compute platforms ship with a specialized MPI implementation. Today, many of these supercomputers are composed of compute nodes consisting of off-the-shelf components. The compute nodes are connected through a high-throughput, low-latency network, for example, forming a Dragonfly [25] or Slimfly [8] network topology. Traditionally, these networks have been an important aspect distinguishing supercomputers and commodity clusters. With the increasing adoption of modern interconnects such as InfiniBand, we observe many of these advanced network features being introduced in smaller clusters, high-end database appliances, and cloud infrastructure [20, 33].

In our system, we use foMPI-NA [6, 15], a scalable MPI RMA library that, for intra-node communication, uses XPMEM, a portable Linux kernel module that allows to map memory of one process into the virtual address space of another, and, for inter-node communication, DMAPP [34], a low-level networking interface of the Aries network that provides an RDMA interface. foMPI-NA extends MPI's interface with notified accesses such as `MPI_Put_notify`. This call triggers a remote write operation similar to a `MPI_Put` with the addition of an notification on the remote machine. Some network implementations refer to this operation as a *write with immediate*. We found that using one-sided operations with notifications that place messages into manually defined mailbox buffers is faster than using Send/Receive interface of MPI, which has similar semantics.

2.3 Concurrency Control

There are several concurrency control mechanisms that are being used in database systems, e.g., Two-Phase Locking (2PL), optimistic concurrency control (OCC), multi-version concurrency control (MVCC), and timestamp ordering (TO) [7, 16]. These mechanisms have been evaluated and compared against each other in recent publications [17, 41].

Table 1: Lock request mode compatibility matrix

	NL	IS	IX	S	SIX	X
NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	
IX	✓	✓	✓			
S	✓	✓		✓		
SIX	✓	✓				
X	✓					

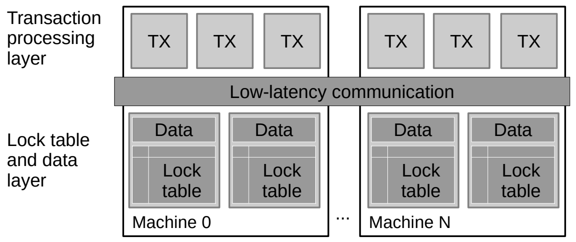


Figure 1: System overview

Furthermore, there has been a significant focus on building reliable, fair, starvation-free locking mechanisms for HPC systems [32] as well as cloud environments [9]. The design of these systems focuses on achieving a high throughput for a small number of highly contended locks and often expects coarse-grained locks to be taken [9]. Many recent RMA locking mechanisms offer support for reader/writer locks [32, 40], but are difficult to extend to more sophisticated locking schemes given the current network technology.

The above locking mechanisms are different from the system we propose: A traditional lock table of a database system offers a large number of locks, most of which are not contended. Furthermore, a database system does not make assumptions about the granularity of the locks. Therefore, it offers several lock modes, including intention locks. The dominant locking method used in database management systems is multi-level granularity locking [16]. It solves the problem that different transactions need to lock and modify resources with a different granularity. Multi-level granularity locking makes use of the hierarchical structure of the data in a database, e.g., a schema contains tables, which in turn contain ranges of tuples. Locks can be acquired at any level in the hierarchy. Before a lock can be acquired on a certain object, all its parent elements (i.e., the elements that contain the object) need to be locked as well. To that end, the locking scheme does not only provide shared (S) and exclusive (X) locks, but also intention locks. The intention shared (IS) and intention exclusive (IX) locks are used to signal that the transaction intents to lock elements further down in the hierarchy in either shared respectively exclusive mode. The shared and intent exclusive mode (SIX) is a combination of the S and IX modes, locking an element in shared mode while stating that one or more child elements will be locked in exclusive mode. Finally, the no lock (NL) mode is used to indicate that the lock is not taken. The compatibility matrix for each lock mode is shown in Table 1.

3. SYSTEM OVERVIEW

Our implementation of a distributed concurrency mechanisms has several components (see Figure 1): (i) the transaction processing layer, (ii) the concurrency control and data layer implementing either a 2-Phase-Locking (2PL) variant or timestamp ordering (TO), and (iii) the communication layer.

3.1 Transaction Processing Layer

The transaction processing agents are responsible for executing the transactions. Each agent runs in its own process, executes one transaction at a time, and is independent of other transaction processing agents. There is no direct communication between the transaction processing agents. Coordination is done exclusively through the concurrency control and data layer.

Upon start-up, the transaction processing agent discovers all available server agents as well as the range of data items for which they are responsible. Each server agent is responsible for a fixed number of items. With this information, the transaction agent can send requests to the appropriate server agent.

When a new transaction starts executing, it has to be assigned a local identifier. System-wide, a transaction gets a global 64-bit identifier consisting of the combination of the local transaction number, which form the upper 40 bits, and the transaction agent identifier, which form the lower 24 bits. If timestamp ordering is used in the concurrency layer, then local transaction numbers are taken from the local system clock, which is synchronized at system startup and at certain intervals. This approach has been used by other authors as well [17]. Apart from assigning an identifier to a transaction, no additional setup is required.

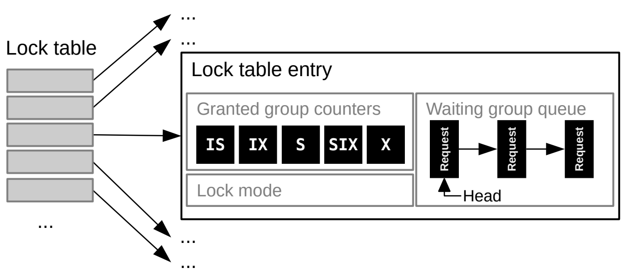
Next, the transaction processing agent carries out the actual logic of the transaction. To read or write from or to the data layer, the transaction generates a request message that is transmitted to the target server agent using a single one-sided RMA write operation. Each request contains a predefined *request* message tag, the identifier of the data item, the identifier of the transaction processing agent, and the requested access mode. If 2PL is used in the concurrency layer, then the access mode corresponds to the lock mode of the lock. In case TO is used, the system just distinguishes between read and write requests. The transaction agent stores the identifier of each server agent it contacts in order to be able to inform it when the transaction is ready to commit or has been aborted.

Corresponding response messages are identified by a specific *Response* message tag. A response messages contains the same information as the request message, with the addition of a flag indicating whether the request was successful or not, as well as the memory address of the requested data item. This allows the transaction processing agent to directly access the data layer through one-sided read and write operations.

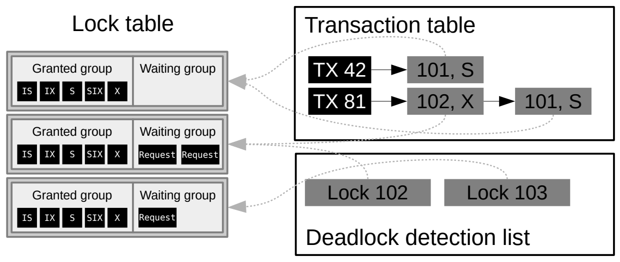
If 2PL is used in the concurrency control layer, the transaction decides at commit time if the Two-Phase Commit (2PC) protocol needs to be executed. This is the case if data has been modified on at least one remote process. If a vote is required, the transaction processing agent starts the 2PC protocol among all involved processes. Processes that did not contribute to a transaction do not participate in the vote. The transaction processing agent takes the role of coordinator, registering how many positive and how many negative votes have been collected. Once every participant has voted, the transaction processing agent informs them about the outcome through the use of a *End of transaction* message. We do not use any optimizations such as Presumed-Abort or Presumed-Commit [7].

3.2 Concurrency Control and Data Layer

The server agents are responsible for receiving and executing access requests from the transaction processing layer. They partition the database into non-overlapping ranges of consecutive data items based on a partitioning scheme all components have agreed on at system startup time. All information needed by the concurrency



(a) Lock table entry



(b) Auxiliary data structures

Figure 2: Data structures used for Two-Phase Locking

control mechanism is colocated with the data items. Therefore, accesses to the server agents do not need to be synchronized. The ranges are chosen such that each server process is responsible for an equal number of data items.

3.2.1 *Two-Phase Locking*

In the case of Two-Phase Locking, the concurrency control mechanism manipulates the following three data structures used to manage the locks: (i) the lock table containing the individual locks, (ii) the transaction table, which contains lists of locks held by each transaction, and (iii) the deadlock detection list that contains all the locks that can be part of a potential deadlock situation. These data structures are shown in Figure 2.

All data structures can be made persistent in order to assist in the recovery protocol. Upon failure of a transaction, the system has a list of all requests, locks, and lock modes for the transaction in question. In case of a failure of one or more server agents, the system can recompute the status of the locks that have either granted or potentially pending requests.

The lock table contains all available locks together with their pending and granted requests. As seen in Figure 2a, the lock data structure is composed of a queue of pending requests (waiting group) and a set of counters (granted group). For each mode, there is exactly one counter indicating how many requests of that mode have been granted. From this information, the lock mode can be computed. This enables the server agent to quickly determine if the head of the queue is compatible with the other requests in the granted group.

The transaction table holds information of each running transaction, which is identified by the global transaction identifier. It implements a multimap, i.e., for each transaction, the table contains a collection of all acquired locks together with their request modes (see Figure 2b). Although individual locks can be released at any point in time, the primary purpose of the transaction table is to implement an efficient Strict 2PL system and to accelerate the recovery in case of system failures. In strict 2PL, there is no shrink phase in which locks are progressively unlocked. Rather, all acquired locks are released by a transaction upon commit or abort. Using this data structure, the lock table server agent can release all the locks held by a transaction without having to receive multiple or variable-sized *Unlock* messages.

For comparison, our prototype implements several deadlock avoidance and detection mechanisms: two textbook mechanisms, namely *No Wait* (NW) and *Wait Die* (WD), as well as a novel time-based mechanism, called *Bounded Wait* (BW).

In *No Wait*, requests that cannot be immediately served are not queued and are immediately rejected, causing the transaction to abort and restart. In this variant, pending requests as mentioned above do not exist. In *Wait Die*, only those requests are queued that come from transactions that are older (i.e., have a smaller transaction identifier) than any of the transactions holding the lock currently. Both of these mechanisms are described in more detail in related work [7].

In the time-based deadlock avoidance mechanism, *Bounded Wait*, the server agent adds the current wall time to all incoming requests before adding them to the waiting group of the requested lock. Furthermore, each server agent keeps a list of local locks that have pending requests (see Figure 2b). The agent iterates over this list to determine how long the head of the queue has been waiting to acquire the lock. A lock must be acquired within a predefined time frame (e.g., 100ms). If a timeout occurs, the transaction is informed about the unsuccessful lock attempt with a negative acknowledgement message and the request is removed from the waiting group. This mechanism enables the system to resolve deadlocks while also avoiding an excessive abortion rate in case of light workload contention. When the last request has been removed from the waiting group, either because it has been granted or because it timed out, the server agent removes the lock entry from the list of locks with pending requests to no longer include it in the computation of deadlock detection mechanism.

3.2.2 *Timestamp Ordering*

To compare the lock-based concurrency control mechanisms with other alternatives, we also implemented textbook timestamp ordering (TO) as explained in related work [7]. In short, timestamp ordering consists in executing conflicting reads and writes to each data item in a predefined order, in our case, in the order of global transaction identifiers. Operations that cannot be carried out in this order lead to the abortion of the corresponding transaction.

In order to enforce this order, TO keeps the timestamps of the last read and the last write of each data item. Any write request of a transaction with a lower timestamp than the read or write timestamp, and any read request with a lower timestamp than the last write request are aborted immediately (as they should have happened *before* the last read or write operation). Write requests with a higher timestamp are kept in a list of pending writes until the corresponding transactions commits, in which case the write is installed, or aborts, in which case it is discarded. Read or write requests with timestamps lower than existing pending writes are also queued, as they need to serve or overwrite the value of those writes, if they are installed. When writes are installed or discarded because a transaction commits or aborts, this may make it possible to serve outstanding read requests, which in turn, may make it possible to install pending writes, and so on. For a detailed explanation, we refer to Bernstein, Hadzilacos, and Goodman [7].

3.2.3 *Data Layer*

The data guarded by the concurrency layer is accessed by the transaction layer through one-sided memory operations. If a transaction needs to read data, it issues a one-sided read (`MPI_Get`) operation that reads out a specific position in the data layer. In order to modify data, the transaction issues a one-sided write (`MPI_Put`) call that instructs the remote network card to overwrite the selected position with the new content that is part of the request. Apart from loading the data and registering the buffers with the network card at start-up, the server agent is not directly involved in data retrieval and manipulation operations. However, it can also persist parts or all of the content of the data layer on disk upon receiving a *Flush data* message from the transaction processing layer in order to be able to recover from failures.

3.3 Low-Latency Communication Layer

In order to support a variety of high-performance networks, the communication between the transaction processing layer and the lock server agents uses the Message Passing Interface (MPI). This architecture has the advantage that the interface is identical for communication between local and remote processes, which hides the complexities arising from large-scale distribution, while still delivering good performance by using the most appropriate communication method based on the relative distance of the processes involved in the communication.

Communication between the transaction processing agents and the server agents is performed exclusively using one-sided RMA operations. In case of conflicts, locks cannot be acquired with a single MPI write operation. Therefore, we do not try to obtain a lock with such an operation, but only request it. Once the lock is granted, the lock server agent confirms this event with another one-sided write operation. We found that this approach is faster than using two-sided communication.

Upon start-up, each process allocates a set of two buffers and registers them with the network card using `MPI_Win_alloc`. This operation is a collective operation, which means that every process involved in the communication needs to execute this operation. During window allocation, the access information to these buffers is exchanged between all processes, such that every component of the system is able to read and write to these regions of memory using RDMA operations. The first of these buffers is used as a mailbox for incoming messages and the second one is used in the voting phase of the 2PC protocol.

Since the server agents can potentially receive requests from any transaction, their mailbox is wide enough that it can accommodate one message from each transaction processing agent. Each process in the transaction processing layer can have at most one pending lock request that needs to be granted before it can continue processing. Therefore, its mailbox size is such that it can hold a single message. *Lock request*, *Response*, and *End of transaction* messages are transmitted by issuing a `MPI_Put_notify` call. In order to avoid synchronization when writing to the mailbox, the i-th transaction processing agent writes its content at the i-th slot in the mailbox.

On the target side, the lock server agent can start listening for incoming notifications by initializing the notified access support of foMPI-NA (`MPI_Notify_init`) and activating a request handle (`MPI_Start`). Using this request handle, a process can either wait for messages (`MPI_Wait`) or perform a non-blocking test to verify if a new notification has been created or not (`MPI_Test`). Once a notification is ready, the target can read out the origin of the request and consume the content at the respective message slot. Using notified access operations, avoids that the target process has to iterate over all message slots, which would limit the scalability of the communication mechanism. Furthermore, using a mailbox is beneficial for small messages as the content of a request can directly be placed in a specific pre-allocated region memory, which avoids any dynamic allocation of RDMA send and receive buffers during execution. When a request is granted, the corresponding notification is placed in the mailbox of the transaction using the same mechanism. The server agents use the non-blocking test to check for incoming messages. If there is no new request to process, it checks for deadlocks. The transaction on the other hand uses the blocking wait operation as it cannot continue processing before the lock has been granted.

The size of the mailbox buffer grows linearly with the number of transaction processing agents. Given that the exchanged messages only contain a few bytes, the system only needs a small amount of main memory to operate the proposed communication layer. The fact that the messages are of fixed size simplifies the design and memory layout of the mailbox buffer in comparison to the generic mailbox-based communication pattern that is used by most MPI implementations. However, as an alternative, for systems that do not support notified access operations, two-sided MPI communication primitives can be used.

The second window is used during 2PC. It is wide enough to accommodate a single 64-bit integer. Before broadcasting a vote request message to the server agents involved in a transaction, the transaction processing agent zeroes out this memory. Upon receiving the vote request, the server agents perform a remote atomic operation on this memory, either incrementing the lower 32 bits to signal a positive vote, or the upper 32 bits to trigger an abort. This is done by issuing an `MPI_Fetch_and_op` operation combined with the `MPI_SUM` argument. Using an atomic operation makes use of the hardware-acceleration available in these network cards and avoids expensive processing in software.

4. EXPERIMENTAL EVALUATION

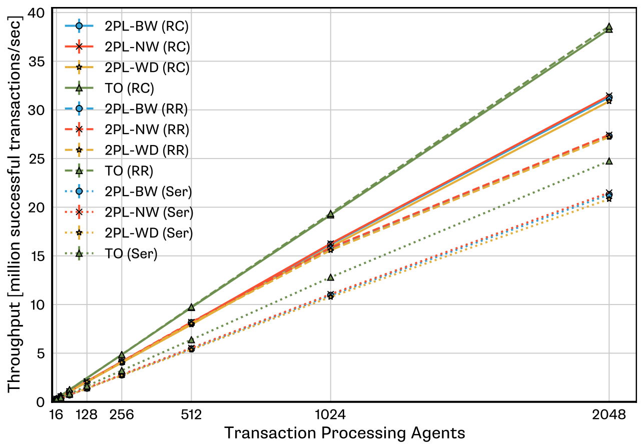
We study the performance of our concurrency control mechanisms on a large-scale compute infrastructure on several thousand processor cores. The source code of our system as well as the scripts needed to run the experiments are available on the web site of our institute.[1]

4.1 Experiment Setup

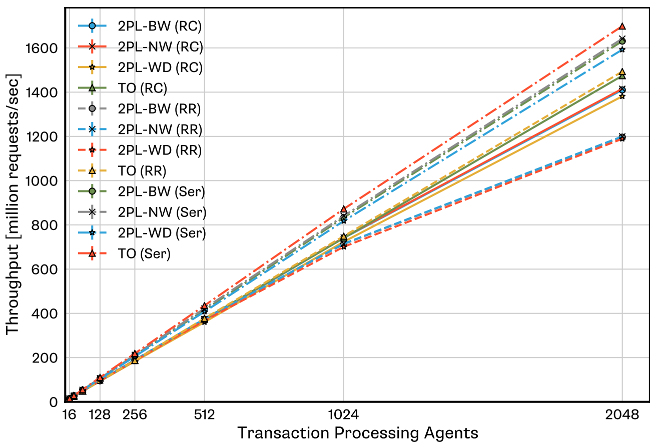
In this section, we describe the experimental platform as well as the workloads used.

Computing Platform: The experiments are conducted on a hybrid Cray XC40/50 computer. The machine is structured in a hierarchical way: it is composed of several compute cabinets, each of which can be fitted with up to three chassis. A chassis can hold sixteen compute blades, which in turn have four compute nodes. The compute nodes of the XC40 partition used in the experiments contain two Intel Xeon E5-2695 v4 processors with up to 128 GB of main memory. The compute nodes do not give the user the ability to write to the local disk and all computations have to be performed in main memory. This setup forces us to disable the flushing of the logs to persistent storage. The compute nodes are connected through a Cray Aries routing and communications ASIC. The Aries ASIC is a system-on-a-chip network device comprising four network cards, one for each of the four nodes of the same blade, and an Aries router. The routers are connected through a Dragonfly [25] topology. The Cray Aries network provides higher throughput and lower latency than most commodity networks. However, in recent years high-speed interconnects have been adopted by many distributed compute infrastructures, in particular database appliances. For example, InfiniBand 4xEDR provides 1-2us latency [37] and both Amazon and Microsoft offer HPC instances for their cloud services

[1] https://www.systems.ethz.ch/projects/rdma



(a) Transaction throughput



(b) Request throughput

Figure 3: Throughput of TPC-C with 2048 warehouses

that achieve 16us and 3us latency, respectively [20, 33]. These machines come at little extra costs compared to other instance types and make the findings in our setup transferable to readily available, off-the-shelf high-speed network infrastructure.

TPC-C Workload: Our concurrency control mechanism implements a conventional lock table conceptually similar to the one used in many existing databases systems. In order to gain insights into scaling out conventional database architectures, we augmented the lock management mechanism of the *MySQL* database server in order to get a detailed trace of all the locks that get acquired. This information includes the transaction number, the identifier of the acquired lock, and the requested lock mode. Using this modified database system, we profiled the TPC-C benchmark using different isolation levels: *serializable*, *repeatable read*, and *read committed*. We run the full benchmark with all transactions including insertions and deletions. In a distributed database system, we envision that different server agents are responsible for managing locks belonging to different TPC-C warehouses. To be able to scale to thousands of cores, we configured the benchmark to simulate 2,048 warehouses. The augmented lock manager provided us with a set of locks and their corresponding lock mode that each transaction requested was either granted or denied. Using the official TPC-C description, we implemented the corresponding queries and access the target data on that warehouse using one-sided read and write operations once all the locks have been acquired.

4.2 Scalability and Isolation Levels

In these experiments, we deploy multiple configurations of the system. Each node in the system uses 16 processor cores that are assigned to the concurrency control and data layer and 16 processor cores assigned to the transaction processing layer. We found that deploying 16 server agents together with 16 transaction processing agents per compute node yields the highest throughput; using either more server agents and fewer transaction processing agents or vice versa is less efficient. Each process is bound to a dedicated core and the processes are distributed equally over both sockets. A server agent is responsible for managing one or more warehouses, while the transaction processing agents execute queries and transactions on behalf of the clients. In TPC-C, each client has a home warehouse, which is accessed most frequently. Therefore, it is reasonable to assume that clients connect to a transaction processing agent that is located on the same physical machine as the data belonging to its home warehouse. Requests targeting a specific warehouse originate from a single source in the transaction processing layer. This setup also reduces the number of conflicts and aborts as transactions targeting the same home warehouse are serialized within the transaction processing layer.

We scale our implementation from a single machine up to 128 physical compute nodes, which corresponds to a total of 4,096 processor cores. In the execution of the TPC-C benchmark used to collect the traces that contains the history of acquired locks by the transaction, we used a total of 2,048 warehouses. Although our concurrency control system is agnostic to the workload and can support an arbitrary number of warehouses, using the traces we collected, the transaction processing agents are limited to replaying transactions that target up to the maximum number of available warehouses.

In Figure 3a, we see executions of the TPC-C trace, using concurrency control mechanisms at different isolation levels. We can observe that all configurations are able to take advantage of the increased core count and are able to scale to thousands of cores. We observe a linear performance increase as we scale out both layers of the system simultaneously. At full scale, all variants of 2PL can support around 21 million transactions per second in serializable mode (Ser) while timestamp ordering (TO) can support around 24.5 million (taking all transactions of the workload mix into account). Relaxing the isolation level to read committed (RC) or repeatable read (RR) increases the throughput to around 32 million transactions per second for the 2PL variants and to around 38 million transactions per second for TO. The corresponding throughput in terms of successful *New Order* transactions (SNOT), which is the official metric for TPC-C, is about 9.5 and 11 million SNOT/s for 2PL and TO in serializable mode, respectively, and about 14.5 and 17 million SNOT/s in RC and RR.

As shown in Figure 3, the deadlock avoidance mechanism chosen for 2PL does not have a significant impact on performance in this experiment. The reason is simple: (i) the time of an individual requests is roughly constant and (ii) the number of requests does not depend on the mechanism. Statement (i) is confirmed by Figure 3b. The throughput in terms of requests (lock requests for 2PL variants, timestamped read and write requests for TO) is largely independent of concurrency control mechanism and isolation level and all configurations can sustain a throughput of over 1,350 million lock requests per second. This is because, in this experiments, all configurations

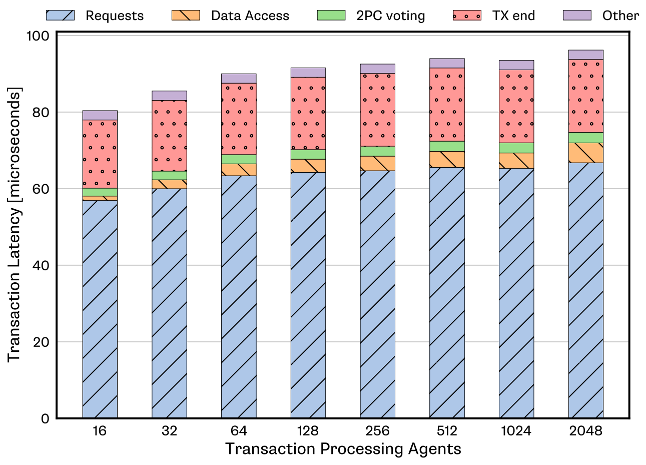


Figure 4: Latency breakdown for TPC-C with 2,048 warehouses and 2PL-BW in serializable mode

can immediately serve virtually all of the requests. The number of warehouses in this experiments is always larger than or equal to the number of transaction processing agents and server agents. Thus, most requests to a particular lock server agent originate from the one transaction processing agent whose home warehouse is placed on the same machine and who has serialized its requests. The few other requests coming from accesses to remote warehouses by other transaction processing agents are spread over many different rows, so there is no contention. The transaction throughput is thus mainly a function of the *number* of requests, which is given by the workload trace as mentioned in Statement (ii). In serializable mode, MySQL takes on average 52.6 locks per transaction, while running a transaction in the *read committed* or *repeatable read* mode requires only 27.6 locks on average. Without intention locks, the number of locks decreases to about 46.8 and 22.5, respectively, explaining the small advantage of timestamp ordering over the 2PL variants.

When adding compute nodes, both layers can be scaled out in the right proportions, thus ensuring that no component is becoming the bottleneck. As resources are added, the lock table can either be distributed with a finer granularity such that each server agent is responsible for fewer locks. Alternatively the higher core count can be used to serve more requests overall. Furthermore, we can observe that a stricter isolation level requires taking significantly more locks, which results in a lower transaction throughput.

4.3 Execution Time Break-Down

Figure 4 shows a breakdown of the latency of each transaction of 2PL-BW on the TPC-C workload with 2,048 warehouses in serializable mode. The majority of the execution time of the TPC-C workload is dedicated to acquiring locks. Around 25% of the time is needed for accessing the data, executing the 2PC protocol, and informing the server agents that a transaction has ended There are multiple reasons for this behavior. First, transactions request multiple locks, while there is at most one vote operation per transaction. Second, locks are acquired one after the other as they are needed. For systems that require a deterministic behavior of the workload, this time could be lowered by either requesting multiple locks in quick succession or by issuing requests that target multiple locks, thus amortizing the round-trip latency. Vote requests can always be issued and collected in parallel. The time required to execute a vote is dependent on the slowest participant, not the number of participants. Third, the majority of transactions target the home warehouse of the client. Since transactions are executed by a processing agent colocated with the locks and the data, most transactions modify items in local memory and acquire only local locks. Transactions that do not modify data on more than one server agent do not execute a 2PC protocol. Transactions that need to execute the commit protocol often have a small number of participants in the voting phase. For the TPC-C workload, a transaction needs to contact on average 1.1 server agents.

The vote of the 2PC protocol requires 8.5 microseconds on average, which is higher than the time required to acquire a lock. The reason for this behavior is that multiple servers need to be contacted, which does not happen completely in parallel. The more servers are involved, the higher the chance that a single straggler delays the outcome. Finally, in our system, atomic operations are cached in fast memory on the network card and updates to these values only become visible after an expensive synchronization call by the initiator of the vote. In general, we observed that remote updates to atomic values become visible to the local processor faster in networks that do not rely heavily on caching intermediate values.

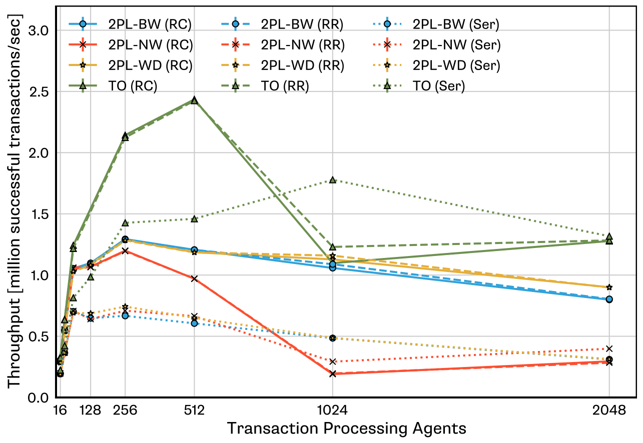
Given that the TPC-C workload with its concept of a warehouse can be partitioned across many physical nodes, we observe that the latency of both operations does not change significantly as we add more cores. This shows that our system exhibits predictable and scalable performance for partitionable workloads.

In additional experiments that we conducted, we observed that the discussion in this section is largely representative for all concurrency control mechanisms. This is expected for the reasons discussed above: in this experiment without contention, each request has roughly constant cost and the number of requests is given by the workload. However, since the number of requests is slightly lower for TO than for 2PL, the time spent on accessing the lock table layer is reduced.

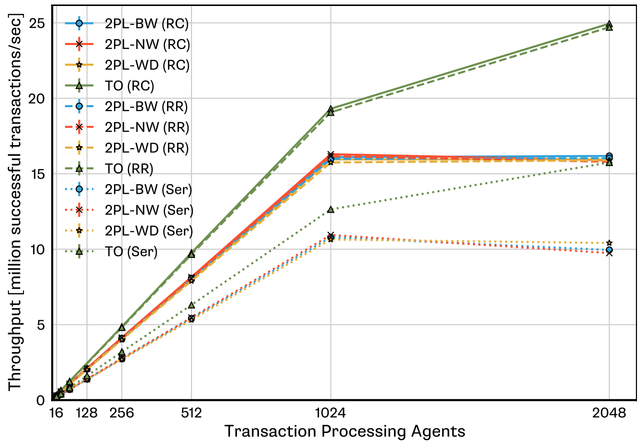
4.4 Contention

In order to study how the different concurrency control mechanisms handle contention, we reduce the number of warehouses in TPC-C. We produce a workload trace as described in Section 4.1. However, this time, we run the benchmark on MySQL with the desired number of warehouses. As mentioned in previous experiments, we assign the locks of at most one warehouse to each server agent. If there are more server agents than warehouses, we split each warehouse across several agents. As long as possible (i.e., until we have 16 times more warehouses than agents), we split the locks of one warehouse among agents of the same node and only afterwards across nodes. The transaction processing agents remain colocated with the lock server agents. When we start splitting the locks across server agents, we also run the same number of transaction processing agents per warehouse, taking transactions from the workload trace in a round-robin fashion.

Figures 5 shows how the different concurrency control mechanisms behave. As long as the number of warehouses is equal or larger than the number of server agents (i.e., less than 64 and less than 1,024 in Figures 5a and 5b, respectively), the throughput scales linearly with the number of cores as before. As soon as warehouses are split across several agents, contention limits scaling: The variants of 2PL maintain their performance for a limited amount of contention, but start decreasing as contention gets higher. Under heavy contention, *Bounded Wait* and *Wait Die* have a significant advantage over *No Wait* – the cost of queuing requests is thus lower than that of repeatably restarting transactions, an effect we study in more detail below. TO seems to be able to exploit more of the parallelism in the workload as it is capable of increasing its performance with more cores even though this setup introduces light contention.

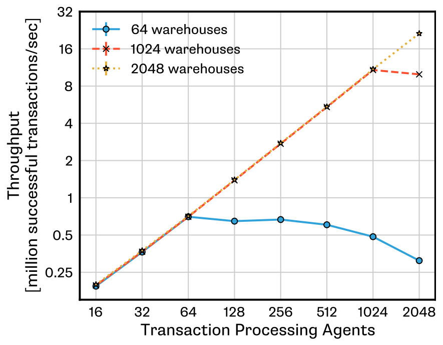


(a) 64 warehouses

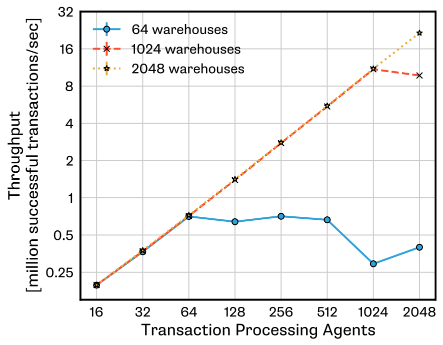


(b) 1,024 warehouses

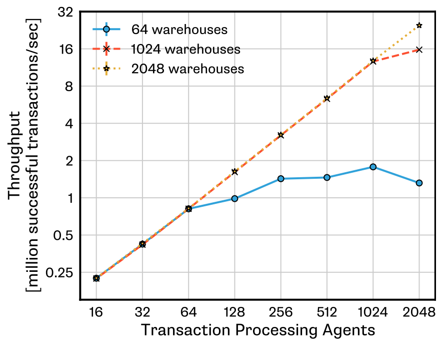
Figure 5: Throughput of TPC-C under contention for different system configurations



(a) 2PL-BW (Ser)



(b) 2PL-NW (Ser)



(c) TO (Ser)

Figure 6: Throughput of TPC-C under contention for several deadlock detection and avoidance mechanisms

Without contention, the isolation level only changes the throughput by a constant factor, which roughly corresponds to the number of locks per transactions in the workload trace.

Figure 6 shows the same data for TO in serializable mode from a different angle: Throughput scales linearly with more cores until contention starts, after which point throughput flattens off and eventually decreases. The corresponding plots of the 2PL variants and other isolation levels looks similar, so we omit them here.

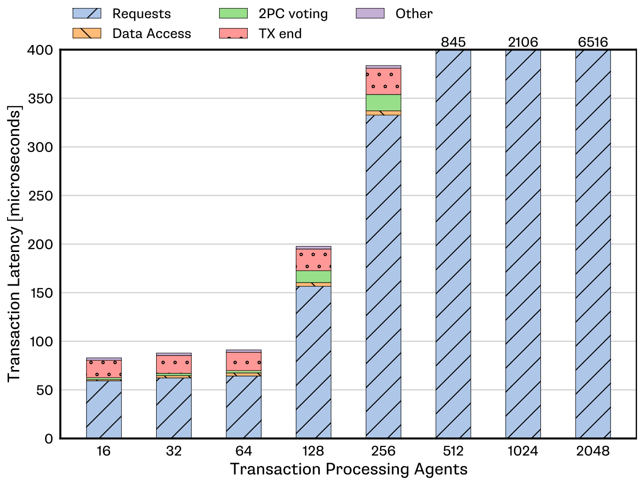
Figure 7a shows the transaction latency breakdown for serializable 2PL-BW running TPC-C with 64 warehouses. From the figure, we can observe that the time is mainly spent in the lock request phase, which increases with contention. This is because requests for contended locks cannot be served immediately anymore and induce waiting time. To be precise, as the number of transaction processing agents that share a warehouse doubles, the time an invidual transaction spends waiting for locks also doubles – a trend that continues for the bars that are cut off the plot (for 512, 1,024, and 2,048 agents). This means that transactions on the same warehouse are essentially executed serially, which explains the plateau observed in Figure 6. Also, the time spent in 2PC now takes slightly longer, but not to the point to make this phase costly compared to the other phases. The remaining phases, in particular releasing locks, are largely unaffected by contention. The breakdown of the other concurrency control mechanisms and isolation levels are qualitatively the same, so we omit them here.

To understand better the difference in performance between the various deadlock avoidance mechanisms, we have a look at how many transactions are aborted. Figure 7b shows their abort rate for TPC-C with 64 warehouses, i.e., with high contention. We can observe that *No Wait* (2PL-NW) has a much higher abort rate than the other mechanisms. This is expected given that this mechanism aborts any transaction as soon as there is a conflict. This strategy does not seem to pay off as 2PL-NW has the lowest throughput among the mechanisms we study. On the other hand, *Wait Die* (2PL-WD) has a considerably higher abort rate than our variant *Bounded Wait* (2PL-BW), but in the end, both strategies lead to roughly the same performance. In practice, a higher abort rate also results in wasted work for the application – hence the preference for 2PL-BW.

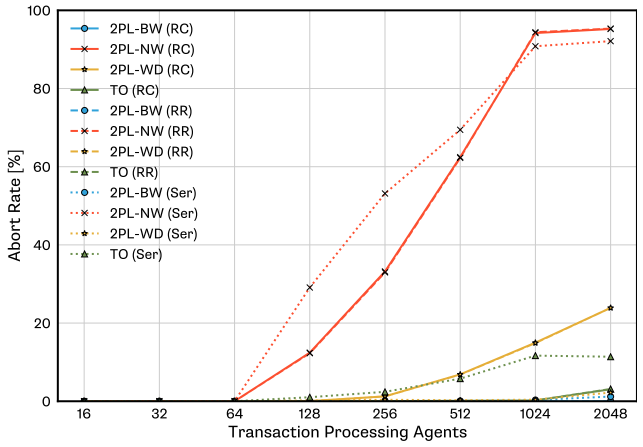
4.5 Network Latency

In order to show the importance of a low-latency network, we introduce an artificial delay for any request between different nodes. Figure 8 shows the throughput for 128 transaction processing agents and 128 warehouses for a varying delay. This configuration corresponds to the one shown in Figure 3b for 128 transaction processing agents and achieves, as expected, the same throughput if the added delay is non-existent.

However, as we increase the delay on networked requests, performance drops significantly: -8% for a delay of 1us, and around a 44 times lower throughput for a delay of 1ms. This experiment clearly



(a) Latency breakdown for TPC-C and 2PL-BW in serializable mode



(b) Abort rate for TPC-C

Figure 7: System configuration with 64 warehouses

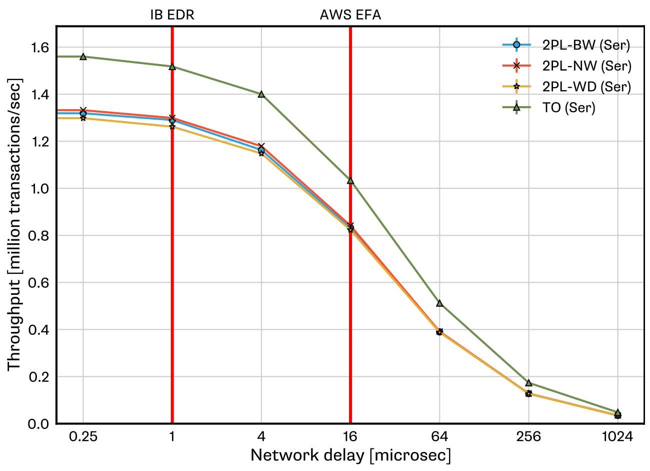


Figure 8: Impact of network latency for 128 transaction processing agents and 128 warehouses

shows how crucial the network latency is for our system. Furthermore, we can conclude that conventional networks, exposed by the operating system through standard sockets, cannot fulfill the latency requirements to make transactions scale to thousands of cores and hundreds of machines.

On the other hand, our numbers also show that a latency in the order of a few microseconds is enough to achieve competitive performance. This means that the single-microsecond latency of InfiniBand EDR (IB EDR) networks as well as the 16us latency offered in modern cloud infrastructure such as the Elastic Fabric Adapter of Amazon Web Services (AWS EFA) should achieve only slightly (up to 2x) lower throughput than the ones in our study. This makes our findings transferable to readily available, off-the-shelf network infrastructure used in most datacenters and cloud environments. For instance, our approach with 2PL in full serializabe mode would reach around 360k successful *New Order* transactions per second on low-latency EFA networking using 128 cores. This result is roughly on par with the throughput obtained by various approaches in related work [17] on conventional networking using 4x more cores. Details are shown in Table 2.

5. DISCUSSION

In this section, we discuss (i) the key insights gained from the experimental evaluation, (ii) the implications on large-scale, distributed lock management, and (iii) directions for future systems and the designs of concurrency control mechanisms.

5.1 Key Insights

As our experiments show, the use of RDMA to implement remote locking allows sharding of the lock table across a large number of distributed nodes. The fast network enables remote locking with an overhead that is only slightly larger than that of obtaining local locks. The sharding of the lock table is the feature that allows the proposed approach to reach such high throughput levels as each node is only dealing with a fraction of the locking traffic. As our results show, the same mechanism of fast RDMA enables a very low-overhead implementation of Two-Phase commit.

The network latency plays an important role in that one can obtain a lock the faster the lower the latency is. This also has an effect on overall throughput (see Figure 8). For the typical range of latencies that can be observed in the cloud, the performance advantage from sharding the lock table remains. In data centers, data appliances, or racks where a distributed database system is often deployed, the performance numbers we obtain should also be reachable – especially when considering how networks are evolving and that InfiniBand HDR (High Data Rate) is already available for some systems and InfiniBand NDR (Next Data Rate) is under development. These new generations of networks will be providing much higher bandwidth and potentially lower latency.

The experiments also show that exploiting the underlying network benefits not only 2PL but also other approaches such as TO. In fact, TO often performs better than 2PC with strict serializability. However, 2PC easily supports a wide range of consistency levels (serializable, repeatable read, read committed) without restrictions on transaction structure and, when combined with multi-version concurrency control or snapshot isolation (i.e., removing the need for locks for read operations), can offer not only more versatility but also better performance than the alternatives. Adopting our approach in existing engines should be much easier than changing the entire transaction management stack to operate under TO or optimistic mechanisms.

5.2 Locking at Large Scale

In the experimental evaluation, we used TPC-C to evaluate our lock table implementation. We observe that the workloads used in this evaluation provide very little contention (including TPC-C). This can be seen by the short amount of time that lock requests spend in the waiting group.

Note that the baseline we provide in this paper intends to test the scalability of concurrency control mechanisms, not the scalability of the workload, a problem already pointed out in related work [42]. It is important to distinguish between the scalability of the underlying mechanism that is offered by the database system and the characteristics of the workload: In the presence of high-speed networks, using a lock-based concurrency control mechanism is a scalable approach for enforcing high transaction isolation levels. To translate this performance to a high throughput in terms of transactions, one requires a scalable workload. This is not the same as having a partitioned workload, but rather depends on the amount of contention present in the workload.

Most database workloads do not have a single highly-contented item and thus not a single lock that every transaction seeks to lock in exclusive mode. However, if a workload exhibited such contention, for most concurrency control mechanisms, a lower overall throughput would be observed than what the mechanism could support. In such a scenario, we would not observe a degradation of the remote access latency, but rather an increase in the waiting time of requests in the queue or a high abort rate if the deadlock detection timeout is too short.

Using a weaker isolation level translates to fewer locks being taken. This means that the load on the lock table decreases and the freed resources could be added to the transaction processing layer to process more transactions in parallel. The overall throughput can be further increased as the isolation level requirements are lowered. Locking mechanisms are not only useful to implement pessimistic concurrency control. *Snapshot isolation* and *optimistic concurrency control* mechanisms can be implemented on top of a locking system, not to prevent concurrent access, but to detect conflicts. In such systems, even fewer locks are needed. For example, in snapshot isolation, transactions do not take locks for reading data, but only need to detect write-write conflicts on the same data element.

The TPC-C workload is partitionable to a large extent. Most accesses require only local locks. If that is not the case, the latency offered by modern networks is small enough that the concurrency control mechanism can still accommodate several million lock operations per second and locks can still be acquired within a couple of microseconds.

The deadlock detection mechanism proposed in this paper is based on timeouts (DL-BW). A request can only wait for a specific predefined period of time in the waiting group before it is canceled. The idea is to detect deadlocks while also not aborting too many transactions in case of light contention on one of the locks. In an alternative design, the server agent could also construct a wait-for graph in order to detect deadlock situations. Since two transactions can be conflicting on two locks managed by two different server agents such a mechanism would require an additional communication protocol between the processes managing the lock table.

In order to make our results relevant for scaling out existing database systems, we avoided any design decision that would restrict the isolation level, consistency guarantees, or the types of transactions that the system would support. Even without any special architectures and optimizations, our distributed lock table can support well over 1,350 million lock operations per second on 4,096 cores. Thus, unlike it is commonly assumed, 2PL and 2PC are viable options to implement distributed, transactional concurrency control.

5.3 Future Designs

In the future, we expect that high-performance networks will offer more functionality to offload compute to the network card [11, 19], which will enable alternative designs for implementing a lock table. Having a richer programming abstraction can potentially save communication round-trips. To that end, new network interfaces are being proposed [2].

Remote append: Following a conventional lock table design, new requests first need to be added to a queue in the waiting group. With the current network technologies, this logic needs to be executed by a process on the CPU. Using one-sided operations for adding an element to a remote queue would require multiple round-trip times: First the end of the queue needs to be identified. Next, a slot for writing the data needs to be reserved (e.g., using a remote atomic operation). Afterwards, the actual content can be added to the queue. Similar steps are needed for removing an element from the queue. While such a system would potentially remove the need for any server agent processes, most network implementations would require all operations to be routed through the network card, even local accesses. Although modern networks are fast, the latency introduced by multiple round trips would still significantly impact performance, in particular the message rate is bound to become a significant bottleneck. Future networks need to provide support for more sophisticated atomic operations in order to avoid that multiple round trips are needed to accomplish simple tasks. For example, atomic append and remove operations would be useful not only for server agents, but for any system which manipulating remote queue-like data structures.

Conditional operations: Conditional operations enable the developer to create simple *if-then-else* operations. For an operation with condition check, the remote network card will first evaluate if the remote data is in a specific state before applying the operation, thus eliminating several round trips and reducing the need for running expensive synchronization or agreement protocols. Conditional operations can be used to significantly accelerate the locking system proposed in this paper: A conditional operation is an efficient way to first check the status of the lock and the queue. If the request can be granted (*if*-branch) the lock counter is being incremented (using a fetch-and-add operation), otherwise (*else*-branch), the request is added to the queue using the previously proposed *append* operation. All these operations would require a single round-trip without involving the remote processor.

Given the arguments above, we expect that future hardware makes it possible to further improve performance of distributed concurrency control by providing a richer set of instructions that can be executed over the network. A more sophisticated network instruction set architecture would (i) eliminate the need to notify the remote processor and thus further reduce latency of individual requests, and (ii) free up a significant amount of processor resources that currently dedicated to running the server logic and offload these operations partially or entirely to the network card [19].

6. RELATED WORK

In this section, we present a detailed analysis of related work in the context of (i) data processing over modern networks and (ii) distributed concurrency control, as well as (iii) a quantitative comparison of several distributed concurrency control mechanisms.

6.1 Data Processing over Fast Networks

High-performance networks have been used in related work to accelerate database systems. Using RDMA requires careful design of systems and algorithms [14].

Analytical queries often involve complex join operations that need to move large amounts of data between the compute nodes. Liu et al. [27] evaluate different aspects of data shuffling over RDMA networks in order to improve the performance of distributed analytical database engines. Furthermore, the authors compare their data exchange operator to MPI-based communication over InfiniBand. Barthels et al. [4] use modern network technologies to accelerate such queries. Their implementation of a distributed join makes use of asynchronous networks to interleave compute and communication and uses RMA operations to directly place data at specific locations in main memory to avoid intermediate copies of the data. The authors scale their implementation to thousands of cores using a supercomputer and use MPI as their communication layer [5]. Rödiger et al. [30] propose a join algorithm that can take advantage of modern networks while also mitigating the negative performance impacts caused by data skew. Several database systems use high-performance networks to accelerate specific parts of the system. BatchDB [28] uses RDMA as a low-latency, high-throughput communication mechanism to replicate data from the primary copy to workload specific replicas. HyPer can distinguish between local and distributed parallelism and uses a tailored communication multiplexer for RDMA networks [31]. Key-value stores profit from the low-latency access to accelerate key lookups [12, 22].

NAM-DB [42] is a database that has been designed from the ground up with one-sided network operations in mind. In NAM-DB, a node can be a memory node, that exposes its main memory to the network, or a compute node. In addition to exposing the main memory to other machines, memory servers perform memory management tasks such as memory allocation and registration, and garbage collection. This system differs from our approach as it avoids synchronization, requires a scalable timestamp mechanism, and forces data to reside in specialized data structures that can be accessed by one-sided RMA operations while also supporting multiple versions. This approach does not enable NAM-DB to offer serializability and is restricted to snapshot isolation.

Other early work include Swissbox [1] and TellStore [29]. Further, more recent work explores the design of replication schemes [43] and index data structures [44] with low-latency RDMA network technology similar to ours.

6.2 Distributed Concurrency Control

Recent years have seen a renewed interest in large-scale concurrency control due to the increasing amount of parallelism and the benefits that it entails. Many of the proposed approaches achieve impressive performance but they often do so by making compromises on the isolation level, types of locks supported, or support for long-running transactions.

Schmid et al. [32] propose a distributed topology-aware implementation of MCS locks optimized for high contention using one-sided network instructions. Yoon et al. [40] design a locking protocol based on fetch-and-add operations that is fault-tolerant and starvation-free. Both approaches have in common that they only support two locking modes (shared and exclusive) and cannot easily be extended to the six modes we support as this would require wider machine words than those supported by atomic RDMA operations available on current hardware.

Spanner [10] is a large-scale distributed database system that focuses on geographic distribution. The system not only uses a lock table to implement concurrency control, but also relies on GPS and atomic clocks to serialize transactions at a global scale. This setup is different from the one used in our evaluation, where the focus is on using high-performance networks to achieve low-latency communication between all system components in a single geographic location. Chubby [9] is lock service designed to provide coarse-grained reliable locking. The design emphasis is on ensuring high-availability for a small amount of locks. This scenario is different from the locking mechanisms used in database systems that focus on achieving a high throughput for a large number of uncontended locks. Furthermore, using coarse-grained locks is not suited for some database workloads, for example coarse-grained locking is suboptimal for transactions that need to access a few specific items.

FaSST [21] is an RDMA-based system that provides distributed in-memory transactions. Similar to our system, FaSST uses remote procedure calls over two-sided communication primitives. The authors pay special attention to optimizing their system to use unreliable datagram messages in an InfiniBand network. Unlike our implementation, this system uses a combination of optimistic concurrency control (OCC) and 2PC to provide serializability. Although the evaluation does not include the TPC-C benchmark, the system is able to outperform FaRM on other workloads (more on FaRM below).

DrTM [39] is an in-memory transaction processing system that uses a combination RDMA and hardware transactional memory (HTM) support to run distributed transactions on modern hardware.

6.3 Quantitative Comparison

Table 2 shows an overview of selected related work. The performance numbers are taken from the original publications. As a best effort, for systems that only implement a subset of the TPC-C workload (marked in the table by a star-symbol), we converted to number of successful *new order* transactions per second (SNOT/s) by assuming that the missing transactions execute at the same speed as the mix of the implemented ones. Since the performance of some schemes decreases with increasing core count, we take the highest achieved throughput as peak performance. For the paper by Harding et al. [17], we use the numbers with 1,024 warehouses, which are better than the numbers with four warehouses presented in the same paper. The paper provides an evaluation for the most popular distributed concurrency control mechanism: Two-Phase Locking No-wait (2PL-NW), Two-Phase Locking Wait-die (2PL-WD), optimistic concurrency control (OCC), multi-version concurrency control (MVCC), timestamp ordering (TO), and the mechanism used by Calvin [35].

In the following, we describe the compromises done by the systems in Table 2. Databases, including FaRM [13] (4.5 million SNOT per second), implementing optimistic concurrency control (OCC) without keeping multiple versions verify at the end of each transaction that no read nor write set of concurrent transactions intersect with its write set. This means that the read and write sets of all transactions need to be kept during the lifetime of the longest-running concurrent query, which limits on how long that period can be [26].

If several versions of each record are stored (MVOCC), such as in Silo [36] (315 thousand SNOT per second), the read sets of read-only transactions do not need to be tracked. Read-only transactions can be arbitrarily long. However, this is not the case for read-write transactions.

Multiversion concurrency control (MVCC) combined with timestamps (TO) handles long-running read-only transactions. Long-running read-write transactions may be problematic or impossible. For example, HyPer [24] (171 thousand SNOT per second) forks long-running transactions into a new process that sees the snapshot of the virtual memory at the time of its fork and cannot do any updates. Furthermore, transactions must be written as stored procedures in order to classify them as long or short-running in advance.

NAM-DB [42] (6.5 million SNOT per second) allows updates in long-running transactions, but checks only for write-write conflicts,

Table 2: Recent work on large-scale concurrency control

System/Paper	Mechanism	Cores (Machines)	TPC-C Performance (SNOT/s)
Our work	2PL-BW (Ser)	4.1k (128)	9.5M
	2PL-BW (RC)	4.1k (128)	14M
	TO (Ser)	4.1k (128)	11M
HyPer [24]	TO+MVCC	8 (1)	171k
Silo [36]	MVOCC	32 (1)	315k
FaRM [13]	OCC	1.4k (90)	4.5M
NAM-DB [42]	TO+MVCC	896 (56)	6.5M
DrTm [39]	HTM	480 (24)	2.4M
Evaluation of Dist. CC.* [17]	TO+MVCC	512 (64)	410k
	2PL-NW	512 (64)	300k
	OCC	512 (64)	100k
	TO	512 (64)	430k
	2PL-WD	512 (64)	340k
	Calvin [35]	512 (64)	380k
Staring into the Abyss* [41]	2PL-DD	1k (1)	760k
	2PL-NW	1k (1)	670k
	2PL-WD	1k (1)	-
	TO	1k (1)	1.8M
	TO+MVCC	1k (1)	1.0M
	OCC	1k (1)	230k
	H-Store [23]	1k (1)	4.3M

thus giving up serializability in favor of snapshot isolation. While snapshot isolation is widely used, it is not without problems [38].

The other MVCC mechanisms from Table 2 achieve serializability by locking new versions until commit time and aborting on updates of records with newer reads. This can lead to starvation in presence of medium or heavy contention because the longer transactions run, the more likely it is that other transactions access their (future) write set. If a single version of the data is kept (TO without MVCC), the problem is even more pronounced as it significantly extends the read and write set.

Recent work on concurrency control proposes to deterministically order data accesses in order to avoid any form of synchronization. While in H-Store [23] (4.3 million SNOT per second), an early system following this idea, this approach did not work with unpartitioned workloads due to the coarse-grained partition locking, newer systems such as Calvin [35] (380 thousand SNOT per second) overcome this problem. Both systems need to know the read and write set of each transaction beforehand (or detect them in a dry-run). This assumption can only be made for stored procedures and is impractical for long-running queries.

In contrast, locking avoids the above-mentioned compromises. As the performance comparison in Table 2 shows, this mechanism does not introduce a significant overhead. Our throughput of 21 million – 9.5 million *new order* – transactions per second is among the highest reported and shows that 2PL and 2PC, in combination with modern hardware, are a viable solution for implementing a scalable concurrency control mechanism.

7. CONCLUSIONS

In this paper, we provide a new baseline for distributed concurrency control at large scale. To that end, we have implemented a lock table and commit protocol taking advantage of the features offered by modern networks, mainly RDMA.

This work shows that conventional Two-Phase Locking and Two-Phase Commit are a viable solution to implement the highest levels of transaction isolation, namely serializability, while being scalable. Furthermore, this approach does not impose any restrictions on the workload in terms of lock modes supported, structure of the transactions, deterministic behavior, or support for long-running transactions. Using MPI to implement low-latency message passing mechanisms, we show that our implementation is able to take advantage of the scale-out architecture used in our evaluation. Provided that there is little contention, local as well as remote locks can be acquired within a few microseconds. The performance of this concurrency control mechanism is expected to improve significantly in the near future as interconnects receive richer interfaces that enable the developer to save communication round-trips [2, 19].

Since many database systems use a conventional lock table, our findings can also be used to scale out existing systems requiring a low-overhead distributed concurrency control mechanism that can sustain a high throughput and take advantage of the parallelism offered by large systems.

Acknowledgments

The work presented in this paper was originally carried out as part of the doctoral studies of Claude Barthels. Ingo Müller and Konstantin Taranov made significant changes and additions in the revised version of the paper that was completed after Claude Barthels graduated and had taken a position in industry. This project has been funded in part by a grant from Oracle Labs.

References

[1] G. Alonso, D. Kossmann, and T. Roscoe. SwissBox: An architecture for data processing appliances. In *CIDR*, pages 32–37, 2011.

[2] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. DPI: The Data Processing Interface for Modern Networks. In *CIDR*, 2019.

[3] C. Barthels, G. Alonso, and T. Hoefler. Designing Databases for Future High-Performance Networks. *IEEE Data Eng. Bull.*, 40(1):15–26, 2017.

[4] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-Scale In-Memory Join Processing Using RDMA. In *SIGMOD*, pages 1463–1475, 2015. DOI: 10.1145/2723372.2750547.

[5] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed Join Algorithms on Thousands of Cores. *PVLDB*, 10(5):517–528, 2017. DOI: 10.14778/3055540.3055545.

[6] R. Belli and T. Hoefler. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In *IPDPS*, pages 871–881, 2015. DOI: 10.1109/IPDPS.2015.30.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman, 1987.

[8] M. Besta and T. Hoefler. Slim Fly: A Cost Effective Low-diameter Network Topology. In *SC*, pages 348–359, 2014. DOI: 10.1109/SC.2014.34.

[9] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *OSDI*, pages 335–350, 2006.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, pages 261–264, 2012.

[11] S. Di Girolamo, P. Jolivet, K. D. Underwood, and T. Hoefler. Exploiting Offload-Enabled Network Interfaces. *Micro*, 36(4):6–17, 2016. DOI: 10.1109/MM.2016.56.

[12] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *NSDI*, pages 401–414, 2014. URL: http://dl.acm.org/citation.cfm?id=2616448.2616486.

[13] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*, pages 54–70, 2015. DOI: 10.1145/2815400.2815425.

[14] P. W. Frey and G. Alonso. Minimizing the Hidden Cost of RDMA. In *ICDCS*, pages 553–560, 2009. DOI: 10.1109/ICDCS.2009.32.

[15] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *SC*, 53:1–53:12, 2013. DOI: 10.1145/2503210.2503286.

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992. ISBN: 9780080519555.

[17] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An Evaluation of Distributed Concurrency Control. *PVLDB*, 10(5):553–564, 2017. DOI: 10.14778/3055540.3055548.

[18] J. Hilland, P. Culley, J. Pinkerton, and R. Recio. RDMA Protocol Verbs Specification, 2003.

[19] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell. sPIN: High-performance streaming Processing In the Network. In *SC*, pages 1–16. ACM Press, 2017. DOI: 10.1145/3126908.3126970.

[20] Jeff Barr. Now Available – Elastic Fabric Adapter (EFA) for Tightly-Coupled HPC Workloads, Apr. 2019. URL: https://aws.amazon.com/blogs/aws/now-available-elastic-fabric-adapter-efa-for-tightly-coupled-hpc-workloads/.

[21] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI*, pages 185–201, 2016.

[22] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. *SIGCOMM*, 44(4):295–306, 2014. DOI: 10.1145/2740070.2626299.

[23] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. R. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *PVLDB*, volume 1 of number 2, pages 1496–1499, 2008. DOI: 10.14778/1454159.1454211.

[24] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, pages 195–206, 2011. DOI: 10.1109/ICDE.2011.5767867.

[25] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. *SIGARCH*, 36(3):77–88, 2008. DOI: 10.1145/1394608.1382129.

[26] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *TODS*, 6(2):213–226, 1981. DOI: 10.1145/319566.319567.

[27] F. Liu, L. Yin, and S. Blanas. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. In *EuroSys*, pages 48–63, 2017. DOI: 10.1145/3064176.3064202.

[28] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *SIGMOD*, pages 37–50, 2017. DOI: 10.1145/3035918.3035959.

[29] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquín, and D. Kossmann. Fast Scans on Key-Value Stores. *PVLDB*, 10(11):1526–1537, 2017. DOI: 10.14778/3137628.3137659.

[30] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In *ICDE*, pages 1194–1205, 2016. DOI: 10.1109/ICDE.2016.7498324.

[31] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed Query Processing over High-speed Networks. *PVLDB*, 9(4):228–239, 2015. DOI: 10.14778/2856318.2856319.

[32] P. Schmid, M. Besta, and T. Hoefler. High-Performance Distributed RMA Locks. In *HPDC*, pages 19–30, 2016. DOI: 10.1145/2907294.2907323.

[33] Tejas Karmarkar. Availability of Linux RDMA on Microsoft Azure, 2015. URL: https://azure.microsoft.com/es-es/blog/azure-linux-rdma-hpc-available/.

[34] M. ten Bruggencate and D. Roweth. Dmapp - An API for One-sided Program Models on Baker Systems. In *Cray User Group*, 2010.

[35] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*, 2012. DOI: 10.1145/2213836.2213838.

[36] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013. DOI: 10.1145/2517349.2522713.

[37] F. V. Van Wig, L. A. Kachelmeier, and K. N. Erickson. Comparison of High Performance Network Options: EDR InfiniBand vs.100Gb RDMA Capable Ethernet. In *SC (Poster)*, 2016.

[38] T. Wang, R. Johnson, A. Fekete, and I. Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB*, 26(4):537–562, 2017. DOI: 10.1007/s00778-017-0463-8.

[39] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*, pages 87–104, 2015.

[40] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed Lock Management with RDMA: Decentralization without Starvation. In *SIGMOD*, 2018. DOI: 10.1145/3183713.3196890.

[41] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB*, 8(3):209–220, 2014. DOI: 10.14778/2735508.2735511.

[42] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The End of a Myth: Distributed Transactions Can Scale. *PVLDB*, 10(6): 685–696, 2017. DOI: 10.14778/3055330.3055335.

[43] E. Zamanian, X. Yu, M. Stonebraker, and T. Kraska. Rethinking database high availability with RDMA networks. *PVLDB*, 12(11):1637–1650, 2019. DOI: 10.14778/3342263.3342639.

[44] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *SIGMOD*, pages 741–758. ACM Press, 2019. DOI: 10.1145/3299869.3300081.