Towards a Non-2PC Transaction Management in Distributed Database Systems

Qian Lin[†] Pengfei Chang[§] Gang Chen[§] Beng Chin Ooi[†] Kian-Lee Tan[†] Zhengkui Wang[¶*]

[†]National University of Singapore [§]Zhejiang University [¶]Singapore Institute of Technology

[†]{linqian, ooibc, tankl}@comp.nus.edu.sg [§]{changpeng3336, cg}@cs.zju.edu.cn [¶]zhengkui.wang@singaporetech.edu.sg

ABSTRACT

Shared-nothing architecture has been widely used in distributed databases to achieve good scalability. While it offers superior performance for local transactions, the overhead of processing distributed transactions can degrade the system performance significantly. The key contributor to the degradation is the expensive two-phase commit (2PC) protocol used to ensure atomic commitment of distributed transactions. In this paper, we propose a transaction management scheme called LEAP to avoid the 2PC protocol within distributed transaction processing. Instead of processing a distributed transaction across multiple nodes, LEAP converts the distributed transaction into a local transaction. This benefits the processing locality and facilitates adaptive data repartitioning when there is a change in data access pattern. Based on LEAP, we develop an online transaction processing (OLTP) system, L-Store, and compare it with the state-of-the-art distributed in-memory OLTP system, H-Store, which relies on the 2PC protocol for distributed transaction processing, and H^L-Store, a H-Store that has been modified to make use of LEAP. Results of an extensive experimental evaluation show that our LEAP-based engines are superior over H-Store by a wide margin, especially for workloads that exhibit locality based data accesses.

1. INTRODUCTION

The past decade has witnessed an increasing interest in adopting shared-nothing database technology to handle fast growing business data. By horizontally partitioning data across different physical machines, shared-nothing systems are highly scalable for non-transactional and analytical workloads which are "embarrassingly partitionable". For OLTP workloads, shared-nothing systems conventionally depend on the 2PC protocol for preserving the atomicity and serializability of distributed transactions [31]. Although these systems offer superior performance for local transactions (i.e., transactions whose data are hosted on a single node), overhead of processing distributed transactions (i.e., transactions whose data are spread over multiple nodes) can degrade the system performance significantly [10, 26, 30]. The key contributor to the degradation is the 2PC protocol [20, 26, 3, 29]. To commit a distributed transaction, the 2PC protocol requires multiple network round-trips between all participant machines which typically take a much longer time than the local transaction processing. As such, much work has been done to address this bottleneck [10, 30]. On the one hand, systems such as Calvin [30] seek to improve the scalability of distributed transaction processing by trading latency for a higher throughput [30]. On the other hand, fine-grained partitioning of the data can be performed offline to reduce the number of distributed transactions [10]. These techniques can reduce but cannot eliminate distributed transactions completely and hence the expensive distributed commit procedure must still be performed.

In this paper, we propose *LEAP* (Localizing Executions via Aggressive Placement of data) as a solution towards non-2PC distributed transaction processing in a shared-nothing architecture, particularly in our epiC system [19]. LEAP converts a distributed transaction into a local transaction to eliminate the expensive distributed commit procedure. To localize the execution of a distributed transaction, LEAP aggressively places all cross-node data needed for the transaction on a single node (i.e., the node issuing the transaction). By doing so, the transaction executor would physically hold all the data it uses during the transaction execution and can easily commit or abort the transaction without worrying about the distributed consensus as in the 2PC protocol. We design LEAP in anticipation of the wide availability of modern fast networks (e.g., 10 Gigabit Ethernet, InfiniBand) to provide latency and bandwidth guarantees.

LEAP is motivated by the following key observations. First, OLTP queries typically involve only a small number of data records [5, 14, 25]. Therefore, the number of records to migrate is small. Second, depending on applications, the data record size may be comparable with the size of a "sub-transaction". This means it is no more expensive to migrate records to the processing node than sending sub-transactions to multiple nodes for processing. Third, data access in real-world distributed transactions typically exhibits some form of locality. For example, if a person accesses his/her domestic account when traveling overseas, it is likely that he/she will repeatedly access the data in this account via a series of transactions issued from his/her place of travel.

LEAP always maintains a single copy of each data record. Each distributed server manages two meta-data structures:

*This work was done while Zhengkui was at NUS.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGMOD'16, June 26–July 1, 2016, San Francisco, CA, USA.

© 2016 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-3531-7/16/06.

DOI: http://dx.doi.org/10.1145/2882903.2882923

data table and owner table. The *data table* stores the application's records in memory, and the *owner table* is a key-value store where each pair associates a record in the data table (identified by its primary key) with its access owner (i.e., a transaction server serving read/write requests for that record). At any time, each data record only has a single access owner. The owner table is partitioned among all transaction servers using standard distributed key-value store techniques. To process a transaction, a processing node S first searches its local storage and checks whether the data is in the local storage. If yes, the transaction is a local transaction and thus can be performed immediately. Otherwise, the transaction is a distributed transaction, and S will then send requests to the owners of those records and ask them to yield data ownership. When S collects all the required data ownership, it processes the transaction locally.

The contributions of this paper are threefold:

- We propose LEAP, a transaction management scheme that localizes any distributed transaction processing to a single node. LEAP facilities efficient distributed transaction processing by eliminating the need of costly distributed commit protocol and inter-node intermediate results exchange during transaction processing.
- Based on LEAP, we propose a practical OLTP engine designed for a shared-nothing database architecture. Specifically, we provide the detailed techniques of enabling LEAP by using a ownership transfer scheme and ensuring concurrency control by employing a lock-based mechanism.
- We conduct extensive experiments to evaluate our LEAP-based systems against a 2PC-based system, H-Store, on a cluster of 64 machines. Results show that LEAP-based transaction processing outperforms that of 2PC-based approach in terms of throughput, latency and scalability.

The rest of the paper is organized as follows: Section 2 introduces distributed transaction processing using the 2PC protocol. In Section 3, we present our proposed LEAP protocol. In Section 4, we provide the system design of the LEAP-based OLTP engine. Section 5 presents results of an experimental study. We discuss related works in Section 6 and conclude in Section 7.

2. PRELIMINARIES

Transactions in a shared-nothing database are classified into two types: *local transactions*, which only access data within a single node, and *distributed transactions*, which access data spanning multiple nodes. Local transactions are efficient to process while distributed transactions are expensive due to the cost of the 2PC protocol typically used to enforce atomicity.

Figure 1 shows an example of 2PC-based transaction processing. This example will also be used to illustrate our proposed strategy in the next section. In Figure 1, the database has two partitions served by two nodes S_1 and S_2. A transaction T submitted at S_1 will access data records r_1 and r_2 stored in two partitions. Here, S_1 (the node where T is submitted) serves as the *coordinator* and S_2 as a *participant*. Then, a two-step strategy is employed to execute T.

(Step 1) Task Distribution: The coordinator first divides the transaction into multiple sub-transactions, each of which is sent to one participant for execution. In our example, T is divided into T_1 and T_2 where T_1 is executed at the coordinator and T_2 is executed at the participant. Therefore,

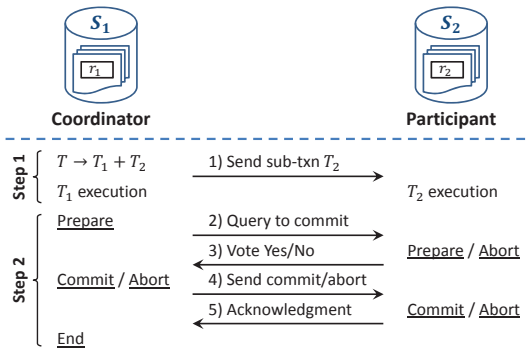


Figure 1: Distributed transaction processing with 2PC. The underlined operations need to force-write related log entries to stable storage.

in this step, the coordinator sends a message containing the sub-transaction T_2 to the participant for execution (message 1 in Figure 1).

(Step 2) Atomic Commit: When the transaction is ready to commit, it enters into the commit step consisting of the voting phase and the completion phase. In the voting phase, the coordinator polls all participants for their readiness to commit (message 2 in Figure 1), and waits for their responses. Meanwhile, after receiving the request, each participant replies with its vote (Yes or No) to the coordinator (message 3 in Figure 1). Before voting positively, the participant prepares to commit by persisting changes to stable storage. Otherwise, the participant aborts immediately. In the completion phase, the coordinator collects all the votes. If all the votes are Yes, the coordinator decides to commit and sends a global commit message to all the participants. Otherwise, it sends an abort message to all the participants (message 4 in Figure 1). Each participant acts accordingly based on the decision it receives and the acknowledgement message is sent to the coordinator.

The 2PC protocol is simple but introduces significant delay due to the communication costs and uncertainty. During the execution, all communications between the coordinator and the participants are achieved by inter-node messages. In our running example, each participant incurs at least four messages. Note that the protocol can still function correctly without the acknowledgement message (message 5 in Figure 1) which is used to enable servers to delete stale coordinator information. Furthermore, participants may be blocked for a long time when failures occur. This happens when the coordinator and at least one participant fail after all non-failed nodes voted Yes; these nodes have no clue about the global decision – it could be a commit decision (if the failed nodes have voted Yes) or an abort decision (if the failed nodes have not voted or voted No). Moreover, for complex multistep transactions, the 2PC-based processing may have to exchange intermediate results among the coordinator and the participants, which would incur additional inter-node message passing.

3. DISTRIBUTED TRANSACTION PROCESSING: THE LEAP WAY

In this section, we present LEAP for distributed transaction processing. We first give an overview of LEAP, and then describe the data structure and protocol to realize it.

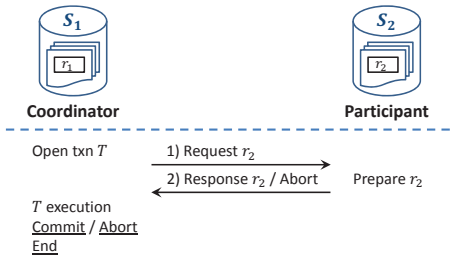


Figure 2: An overview of LEAP for distributed transaction processing.

3.1 The Big Picture

The main goal of LEAP is to eliminate the expensive 2PC protocol. What LEAP does is to do away with 2PC completely by turning all distributed transactions into local transactions. To achieve this, LEAP aggressively migrates data requested by a distributed transaction to a single node at runtime. In other words, the system dynamically places data at the nodes where the transactions need them.

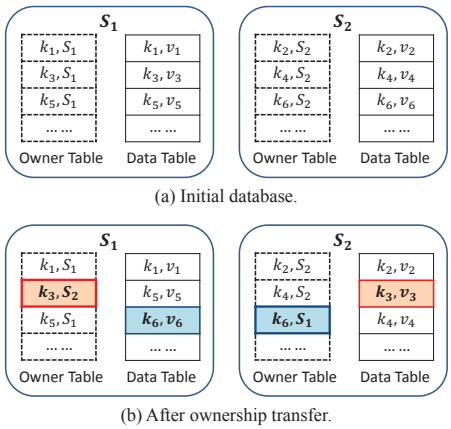
LEAP employs the concept of an *owner* of a record.

Definition 1 (Owner). Given a data record r, the owner of r is the single server node S where r is placed. Moreover, S is granted exclusive access to r.

By definition, except for the owner, no other servers can access the data record and, at any given time, each record is assigned to a unique owner. Therefore, if a node S_1 wants to access a record r owned by another node S_2, it has to issue a request to S_2 for ownership transfer. When S_2 gives up its ownership of r, it sends r to S_1; only then can S_1 access r, which turns out to be a local access.

Under LEAP, a distributed transaction is processed as follows: the coordinator (where the transaction is submitted) aggressively requests from the participants for the data to be migrated to its site; once these data are placed at the coordinator site, it executes the transaction locally. To illustrate this idea, we refer the reader to the previous running example again. In Figure 1, data are partitioned into two partitions that are placed at two nodes, S_1 and S_2, which own records r_1 and r_2 respectively. To execute the transaction T which accesses r_1 and r_2, as shown in Figure 2, the coordinator S_1 triggers a migration process to S_2 by requesting to be the owner of r_2 (i.e, message 1 in Figure 2). Once S_2 receives the request, it decides whether the transfer can be carried out. If the migration can be performed, S_2 transfers the ownership of r_2 to S_1, by transmitting to S_1 a ownership transfer granted message (i.e., message 2 in Figure 2) along with the requested data (i.e., r_2). At this point, S_1 owns all data requested by T and processes T locally. When T commits, it commits locally without interfering other distributed nodes. Comparing with 2PC, LEAP uses fewer messages (two in this example) to execute the transaction and thus reduces the processing latency.

At first sight, it seems that LEAP's aggressive dynamic data placement approach is costly. However, this strategy is practical and feasible due to three reasons. First, OLTP queries only touch a small number of data records. Therefore, the cost of transferring data is comparable to the communication overhead under 2PC (In 2PC, we need to transfer sub-transactions as well as the intermediate results). Second, advances in networking technologies such as 10 Gigabit Ethernet and InfiniBand continually reduces the time required for data transmission. Third, subsequent transactions in the same node that access the same records can avoid remote accesses. Therefore, it is worthwhile to trade a little bit more data transmission for fewer inter-node messages.



(a) Initial database.

(b) After ownership transfer.

Figure 3: Example of ownership transfer.

The LEAP design and the analytical model assume modern network infrastructure which offers sufficient bandwidth and low-latency communication. Such fast networks are widely available in most local area networks (LANs).

3.2 Managing Ownership and Data Placement

The first challenge to realize LEAP is to ensure that the ownership and data placement can be effectively managed. For example, how can we find the location of a record if it is being moved frequently? Towards this end, we decouple the meta-data (on ownership) from the data, and employ two types of tables: an *owner table* which maintains owner information and a *data table* which stores the actual data records. For each record, represented as a key-value pair $r = (k, v)$, we store two pieces of information in the aforementioned two tables: 1) a data owner record $r_o = (k, o)$ where o is the identifier (e.g., IP address) of the owner of r in the owner table, and 2) a data record $r_d = (k, v)$ in the data table. The owner table is partitioned across all nodes using any standard data partitioning techniques (e.g., hash or range) on k. The data table is also partitioned across the nodes; however, a data record r_d is stored at its owner's local data table.

Initially, a record r's owner is assigned to the node that hosts the owner record r_o, namely r_d is stored along with r_o. Subsequently, whenever ownership transfer occurs, the data record r_d will be moved to its new owner's local data table. The owner record r_o, on the other hand, is never moved; instead, it remains at the node where the record was initially placed. The owner record r_o is essentially used for tracking the current owner of r.

Figure 3.a shows the initial storage layout of a cluster, where the data and its corresponding owner information are stored in the same node in S_1 or S_2. Figure 3.b shows the updated storage layout after some ownership transfers. For instance, after record $\langle k_3, v_3 \rangle$ is migrated from S_1 to S_2, the owner information still remains in S_1 with an updated owner information. Similarly, transferring record $\langle k_6, v_6 \rangle$ to S_1 results in the new owner information in S_2.

To minimize storage utilization, there is actually no need to physically store the owner information when the records

and the owner information collocate. Only when a record is transferred to other nodes do we need to maintain its owner information in the owner table. In other words, by default the owner of one record is the node that holds its owner information. For example, under this optimized scheme, all entries in the owner tables of both S_1 and S_2 are eliminated in Figure 3.a, and only entries of $\langle k_3, S_2 \rangle$ and $\langle k_6, S_1 \rangle$ remain in the owner tables of S_1 and S_2 respectively in Figure 3.b. Furthermore, a good partitioning strategy of the keys may reduce the size of the owner table, as most records may remain in the node where they were initially placed.

3.3 Ownership Transfer

Our proposed LEAP scheme exploits an ownership transfer protocol. To introduce the ownership transfer protocol, we first provide the following definitions.

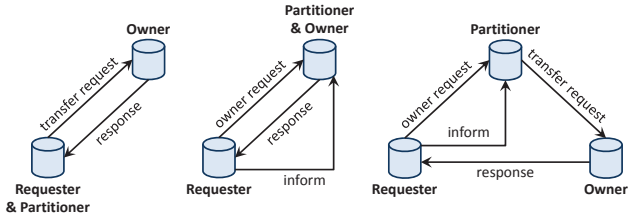
Definition 2 (Requester). A node is referred to as the *requester*, if the node requests data that are owned by other nodes.

Definition 3 (Partitioner). A node is referred to as the *partitioner* of a record r, if the node holds the ownership information of r.

Recall that given a transaction T, if the node is the owner of all the data that T requires, the transaction can be directly executed by the node without incurring any ownership transfer. Ownership transfer only happens when the node needs to request the data owned by other nodes. To transfer ownership from other nodes, the requester runs a ownership transfer protocol which conceptually consists of the following 4 steps.

- *Owner Request:* The partitioner figures out who the current owner of the requesting record is by performing a standard key-value lookup over the owner table.
- *Transfer Request:* After finding the current owner of the record, the ownership transfer request is sent to the owner to request for ownership transfer to the requester.
- *Response:* Once the owner receives the ownership transfer request, it checks the status of the requesting data. When the ownership cannot be granted to the requester (e.g., due to contention [1]), the owner sends a reject message to the requester. Otherwise, the owner sends a response message to the requester along with the requested data. Once the data is sent to the requester, it automatically becomes invalidated at the old owner's node that will be garbage collected in the future.
- *Inform:* After receiving the data, the requester informs the partitioner to update the ownership with respect to the ownership transfer. Note that during the above three steps, the ownership is not updated in the partitioner. Only when the ownership is transferred successfully and the inform message is received, then will the partitioner update the ownership accordingly. This guarantees the ownership consistency in terms of node failure during the ownership transfer. Moreover, the requester will also inform the partitioner about the failure of ownership transfer if either the owner request or the transfer request is rejected (e.g., due to concurrency control); and the partitioner will retain the ownership. Therefore, the ownership of data is maintained in a consistent state.

[1]Contention happens when multiple transactions request the same record and cannot be granted together. To prevent deadlock, one of the transactions has to abort. More details will be discussed in Section 4.1.2.



(a) RP-O. (b) R-PO. (c) R-P-O.

Figure 4: Ownership transfer cases.

Before a message is transmitted, it is logged locally at the sender site. When failure happens, any message loss is handled by resending the message after a predetermined timeout period. Note that the response step is the only step that may lead to data loss. To address this issue, we allow the original owner to maintain a backup of the invalidated data for a longer timeout until deletion. Therefore, when the response message is lost, the original owner can recover the data. Appendix D provides more details on how message loss is handled in LEAP.

The above four steps compose the general processing procedure for handling one ownership transfer. For ownership transfer, there are three different situations depending on who the requester, partitioner and owner are. Figure 4 illustrates these three situations, namely RP-O, R-PO and R-P-O. They may incur different overheads with respect to the number of remote messages generated. RP-O is when the requester is the same as the partitioner. In this case, the owner request step and the inform step are efficiently conducted within the requester itself, and the other two steps can be achieved by using two inter-node messages. R-PO arises when the partitioner remains the same as the owner but is different from the requester. In this case, each of the owner request, response and inform steps needs one inter-node message. In R-P-O, when the requester, partitioner and owner are three different nodes, each step utilizes one inter-node message.

Recall that once the requester obtains the ownership of the data, it automatically becomes the owner of the data. The owner will own and keep the data locally as long as the ownership is not transferred to others. This ensures the data can serve multiple instances of the same transactions at the owner node and avoids frequent data transfer.

For ease of discussion, we first consider data access via primary key. How to support data access via non-primary keys in the ownership transfer of LEAP is discussed in Appendix B.

3.4 Comparison with 2PC

Benefiting from the modern high speed network technology, the latency of network transmission becomes nearly indistinguishable for transferring small data (e.g., a tuple or an SQL statement). Given the fact that network I/O still remains orders of magnitude slower than main memory operation, the number of sequential messages passed mainly determines the cost of distributed in-memory transaction processing. As can be seen from Figure 1 and Figure 4, LEAP shows its superiority over 2PC in terms of the number of sequential messages passed.

3.4.1 Theoretical Analysis

To better appreciate LEAP-based transaction processing, we further provide a theoretical analysis with regards to the *latency expectation* of each individual transaction to evaluate the transaction processing efficiency of the two different techniques. Here, the latency consists of two components: transaction processing and transaction commit. We ignore the cost of sending query statement in the initial stage since its cost is the same for both LEAP and 2PC. Appendix C summarizes the notations used throughout the analysis.

To simplify the analysis and focus on the comparison between LEAP and 2PC, we make the following assumptions:

- **(A1)** Time for local processing is negligible compared with the distributed data communication. In other words, the cost of distributed data communication dominates the overall performance.
- **(A2)** The cost of sending an SQL query is the same as the cost of sending one tuple. This assumption generally holds due to the fast networking enabled by modern network infrastructure.
- **(A3)** Each OLTP transaction only accesses a small subset of the entire database, and does not perform full table scans or large distributed joins [25]. Specifically, we assume each transaction performs n data accesses whose footprints (e.g., keys of the tuples) are given in the transaction statement. In particular, for a LEAP transaction, all its data accesses can be processed in parallel since all its required tuples are known upfront, and thus the expected latency of one data access reflects the expected latency of all the n data accesses within the transaction.

Consider a partition-based database. We define data access to be *local* if the data being accessed was initially assigned to the node that the transaction is currently running on. Otherwise, data access is defined to be *remote*. Note that though LEAP would migrate data during transaction processing, the above definitions always refer to the initial state of database partitioning. Moreover, we further define whether a remote data access is with locality. Suppose node S_1 requests data r from node S_2. This remote data access is referred to be *with locality* if r will be only accessed by transactions in S_1 in the near future. In contrast, it is referred to be *without locality* if r will be frequently requested by nodes other than S_1.

First, we consider the LEAP-based transaction processing. Let $P(R)$ be the probability of data access being remote and $P(L|R)$ be the probability of remote data access with locality. Then the probabilities of different types of data access are calculated as follows:

- $P_l = 1 - P(R)$: probability of local data access.
- $P_{rl} = P(R) \cdot P(L|R)$: probability of remote data access with locality.
- $P_{rnl} = P(R) \cdot \big(1 - P(L|R)\big)$: probability of remote data access without locality.

Accordingly, we analyze the cost estimation of each type of data access. For local data access, based on assumption A1, the cost only includes accessing the data that are previously transferred to other nodes via remote data access without locality. Thus, the cost of the local data access can be calculated as

$$t_l = \alpha \cdot X \cdot P(R) \cdot \big(1 - P(L|R)\big)$$

where α is the average number of sequential messages passed within a protocol round-trip and X of one-time network communication is used for all kinds of message passing based on assumption A2. For LEAP, α can be 2, 3 or 4 according to the ownership transfer cases as shown in Figure 4.

Similarly, the same estimation for remote data access with locality is as follows:

$$t_{rl} = \alpha \cdot X \cdot P(R) \cdot \big(1 - P(L|R)\big)$$

This is because the accessed data is expected to become locally available as long as they have been migrated to the requesting node.

Finally, for remote data access without locality, the data are expected to be at the remote node. Although there is a chance that the accessed data may have been transferred to the requesting node through some prior data accesses, we ignore such a chance as it is expected to be rare. Thus, the cost for this type of data access is as follows:

$$t_{rnl} = \alpha \cdot X$$

Since LEAP converts every distributed transaction into a local transaction, transaction commit can be performed locally and with negligible cost. Therefore, under assumption A3, we can derive the latency expectation of LEAP-based transaction processing as the sum of different types of data access with their corresponding probabilities, i.e.,

$$\begin{aligned} T_{LEAP} &= t_l \cdot P_l + t_{rl} \cdot P_{rl} + t_{rnl} \cdot P_{rnl} \\ &= \alpha \cdot X \cdot P(R) \cdot \overline{P(L|R)} \cdot \big(2 - P(R) \cdot \overline{P(L|R)}\big) \end{aligned} \tag{1}$$

where $\overline{P(L|R)} = 1 - P(L|R)$ represents the probability of remote data access not preserving locality.

Next, we consider the 2PC-based transaction processing. Recall that 2PC is only needed when one transaction involves data access from more than one node. With assumption A1, we only need to deal with the case where the transaction processing involves multiple nodes. With n data accesses, the probability of a transaction being distributed is

$$P_{dt} = 1 - \big(1 - P(R)\big)^n$$

Furthermore, since 2PC-based distributed transaction processing may involve multiple threads residing in different nodes, a multistep transaction may need to exchange intermediate result among the distributed threads. Suppose i steps in a multistep transaction requiring intermediate result transfer, each of which involves at least one message passing. Therefore, we can derive the latency expectation of 2PC-based transaction processing as

$$\begin{aligned} T_{2PC} &= \beta \cdot X \cdot P_{dt} + i \cdot X \cdot P_{dt} \\ &= (\beta + i) \cdot X \cdot \Big[1 - \big(1 - P(R)\big)^n\Big] \end{aligned} \tag{2}$$

where β is the number of sequential messages passed within the 2PC procedure. As shown in Figure 1, β equals to 5 in a normal 2PC protocol [2].

The above analysis on latency expectation provides three insights for LEAP/2PC-based transaction processing. First, without considering the locality of remote data access (e.g., $P(L|R) = 0$), T_{LEAP} grows with $P(R) \cdot \big(2 - P(R)\big)$, whereas T_{2PC} grows with $1 - \big(1 - P(R)\big)^n$. This implies the superiority of LEAP over 2PC becomes significant when remote

[2]For simplicity, the log-writing cost is omitted in the cost.

data access increases (as illustrated in Appendix F). Second, remote data access with locality would benefit T_{LEAP}, whereas it does not affect T_{2PC}. Third, 2PC-based processing becomes more expensive when transactions involve intermediate result transfer, whereas the LEAP-based one does not suffer from such an issue.

4. LEAP-BASED OLTP ENGINE

In this section, we introduce the design of L-Store, a LEAP-based OLTP engine. Figure 5 shows the architecture of L-Store which consists of N nodes and one distributed in-memory storage. There are four main functional components in each node as follows.

- **Application Layer:** This layer provides the interface to interact with the client applications.
- **Storage Engine:** L-Store employs an in-memory storage. To facilitate efficient data access, lightweight indexes (e.g., hash index) are utilized to facilitate speedy retrieval of the data tables, the owner tables and the locks.
- **Transaction Engine:** This engine handles transaction execution. It adopts multi-threading to process transactions in order to increase the degree of parallelism. It interacts with the storage engine to fetch/write the data and the node management component to detect the data locks and send the ownership transfer request.
- **Node Management:** This layer manages the entire node including controlling the locks and running the LEAP protocol. It also serves as the agent for inter-node communication. Its implementation applies the actor model [17] to interact with the messages for ownership transfer.

Based on the above architecture, we further present the key designs on concurrency control, transaction isolation, and fault tolerance.

4.1 Concurrency Control

Concurrency control ensures that database transactions can be performed concurrently without violating data integrity. In this section, we introduce a lock-based distributed concurrency control scheme in the system. Evaluating L-Store with other concurrency control techniques such as multiversion-based protocols is left as a future work.

In each node of L-Store, the concurrency control is achieved by maintaining two data structures: data lock and owner dispatcher. The *data lock* is designed to control the ongoing data access. Considering each OLTP transaction typically accesses only a few tuples and is unlikely to clash with other concurrent transactions, the lock granularity is applied to tuples. The data lock scheme uses read-write lock mechanism [33]. The *owner dispatcher* is designed to handle concurrent ownership transfer requests of the same tuple from different nodes. When these concurrent requests are sent to the partitioner, the owner dispatcher in the partitioner would decide which one to process first according to the deadlock prevention strategy. The main property of the owner dispatcher is to guarantee that only one request is forwarded to the data owner to complete the data transfer.

4.1.1 Life Cycle of a Transaction

Now, we introduce the life cycle of one transaction execution in L-Store. To guarantee serial equivalence and high-level transaction ***isolation***, we employ the strict two-phase

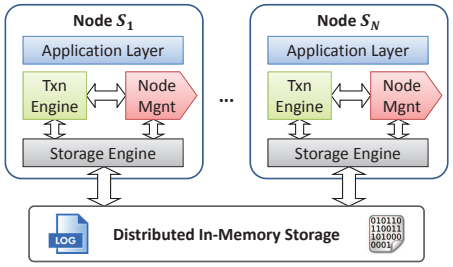


Figure 5: Architecture of L-Store.

locking scheme (S2PL) [31] which guarantees serializability and recoverability of transaction processing. In S2PL, a transaction holds on to all granted locks until it commits or aborts. Transaction execution proceeds in the following phases.

- **Begin:** In the initial phase, the transaction executor thread will initialize the transaction and analyze the execution plan (e.g., local/distributed transaction, number of intermediate steps, data required in the first step, etc).
- **Data Preparation:** Next, the transaction executor will check whether the required data is locally available. This is performed by checking the local in-memory storage. If the data is stored in the local memory, it indicates the current node is the owner and the transaction can be executed with the local data. The executor will then send the data request to the lock manager to lock the data for use. However, if any of the requesting data is not available locally, the executor thread will send an ownership transfer request to the corresponding partitioner to remotely grab the data.

  During this phase, if the executor receives one abort signal due to conflict through either ownership transfer protocol or deadlock prevention mechanism, the transaction will directly enter the abort phase.
- **Execution:** In this phase, a transaction can access or update any data item from the local store. It is sometimes impossible to predict at the start of a transaction which objects will be used. This is generally the case in interactive applications. Therefore, under such a scenario, the transaction may re-enter the previous phase to get the data in the middle of the transaction execution. When this phase finishes, it will enter the commit phase.
- **End: a.) Commit:** The transaction enters the commit phase when the whole transaction has completed successfully. All the data updates are applied in the local memory. In addition, the write-ahead logging technique is applied to write the transaction log into the distributed persistent storage. Once the log is written successfully, the transaction is considered to be completed and a commit flag is set to the transaction log in the end. Meanwhile, all the locked data are released.

  b.) Abort: On conflict, the transaction is rolled back to cancel all the updates during the transaction execution. Meanwhile, all the locked data is released.

  The commit and abort strategies provide the ***atomic*** transaction execution where either all operations in one transaction are committed or aborted.

4.1.2 Deadlock Prevention

Deadlock happens when two or more competing processes

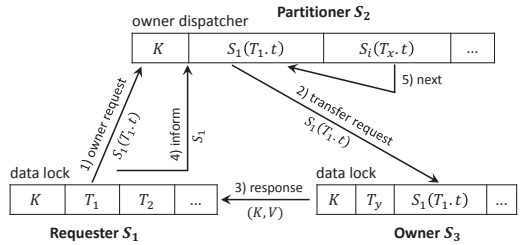


Figure 6: Ownership transfer with contention.

(i.e., transactions or ownership transfer requests in L-Store) are waiting for each other to finish, and thus neither ever does. To tackle this problem, we adopt a timestamp-based deadlock prevention technique [31] to manage requests for resources so that at least one process is always able to get all the resources it needs. In L-Store, each transaction is assigned a globally unique timestamp according to the time it arrives at the execution node (see Appendix J for implementation details). Based on the transaction timestamps, the deadlock prevention is implemented using the *Wait-Die* policy to handle conflicts. The intuitive idea is to set the priority of the transactions. An older transaction with a smaller timestamp is endowed with higher priority than a younger one. In the Wait-Die policy, higher-priority transaction waits for the data held by a lower-priority transaction, whereas lower-priority transaction aborts immediately if the data is held by a higher-priority transaction. To simplify the presentation, we define a Wait-Die based conflict handling operation that will be used in the rest of this section.

Suppose transaction T_1 is holding the lock of record K and transaction T_2 requests to access K. Under the Wait-Die policy, the conflict handling function CH between T_1 and T_2 is defined as follows.

$$CH(T_1, T_2) = \begin{cases} T_1 \text{ waits} & \text{if } T_1.t < T_2.t \\ T_1 \text{ aborts} & \text{if } T_1.t > T_2.t \end{cases} \quad (3)$$

where t is the timestamp of transaction.

Moreover, every aborted transaction is restarted with its initial timestamp so that starvation is avoided.

4.1.3 Handling Concurrent Requests

In L-Store, concurrent requests may arise in two situations. First, transactions within a node concurrently access the local data. Second, transactions concurrently access data that require ownership transfer. The second case also includes the situation where multiple nodes concurrently request the ownership of the same data.

The first category of the concurrent request can be easily handled by traditional centralized lock-based concurrency control mechanisms like the Wait-Die policy. L-Store differs from other lock-based distributed systems in handling the second category of concurrent requests.

For ease of presentation, we first assume that there is one transaction in the requester that needs a remote record. We will generalize to the case where multiple transactions at a node may request for remote records later. We will use the following running example in our discussion.

Requester Side: Suppose transaction T_1 in node S_1 requires one record K that is currently held in node S_3 as shown in Figure 6. Before S_1 sends the ownership transfer request, it first builds a new "request" lock item in the data lock to indicate that one ownership transfer request of K from S_1 has been sent. The difference between this request lock and other data lock is that this locked key does not have data in the local storage yet. For deadlock prevention, the timestamp of the request transaction is also attached to the message (i.e., $S_1(T_1.t)$).

Partitioner Side: In the partitioner side, there could be multiple ownership transfer requests sent from different nodes at the same time (i.e., $S_1(T_1.t)$ from S_1 and $S_i(T_x.t)$ from S_i). One of the main design principles is to process the request sequentially. In other words, at any time, only one request is sent to the owner and the others have to wait until the previous one finishes and sends out the inform message to the partitioner. This is to avoid the overheads of re-sending requests: if multiple requests are all sent to the owner, only one of them will be granted ownership while the rest have to resend again (since there is a new owner).

To prevent deadlock, the Wait-Die policy is also applied here to handle the conflict among different requests using Equation 3. To reduce the abort rate, another design principle is to choose the request attached with the biggest timestamp to process first. Assume $T_1.t > T_x.t$ and then $S_1(T_1.t)$ will be processed first. Here, the one that is being processed is analogous to holding one process "lock". And all the rest (e.g., $S_i(T_x.t)$) have to wait in the waiting queue.

In addition, before the partitioner S_2 receives the inform message acknowledging the completion of $S_1(T_1.t)$, all the new emerging requests for the same K apply the Wait-Die handling check by comparing with $T_1.t$ to determine whether it can wait in the queue or has to abort. After receiving the inform message, the owner dispatcher would pick another requests with the biggest timestamp from the waiting queue to process until all the requests have been processed.

Owner Side: After receiving the ownership transfer request of K, S_3 handles the request in a manner similar to a local transaction applying for an exclusive lock. First, it checks whether K is currently locked by another local transaction. If not, the data is directly granted to the request by sending one response message with the data to S_1. Otherwise, it enters the next step to determine whether the request can wait or abort. Assume T_y is locking K when $S_1(T_1.t)$ arrives. This step compares the timestamp of the transaction holding the lock and the timestamp attached in the request using the conflict handling function $CH(T_y, S_1(T_1.t))$ as defined in Equation 3. If the decision is abort, then the owner does not allow $S_1(T_1.t)$ to wait but reject the request via sending one abort response message to S_1.

Generalization: Now, we are ready to see how to handle the situation where multiple transactions at the requester accessing the same remote data. For instance, when transactions T_1 and T_2 both request K in S_1, a straightforward approach is to send multiple ownership transfer requests to the partitioner. However, this would incur high ownership transfer overhead, as the system has to handle two requests from the same node. To tackle this issue, our approach is to only send one request for the multiple transactions. The approach is similar to what is employed in the owner dispatcher, where the transaction with the biggest timestamp is chosen to process the request and the rest have to wait in a waiting queue. For instance, if $T_1.t > T_2.t$, T_1 is chosen to send the request and T_2 has to wait. Similarly, if a new transaction (e.g., T_z) emerging in S_1 requires K as well, it has to do the deadlock prevention using the $CH(T_1, T_z)$ to determine whether T_z should wait or abort.

Based on the message received from S_3, S_1 would take an appropriate action accordingly. When the message is an abort message, S_1 would abort the transaction T_1 and pick another transaction (i.e., T_2) with the biggest timestamp from the waiting queue to resend the ownership transfer request. Otherwise, the "request" lock is converted into a normal data lock to serve the local transaction. In this case, the data has been granted and can serve all the waiting transactions in the queue.

4.2 Fault Tolerance

L-Store is designed to run over large-scale clusters, and provides fault tolerance guarantee. As L-Store employs an in-memory storage, data stored in memory may be lost when data node crashes. To tackle this challenge, we adopt data logging and checkpointing techniques for data ***durability*** as what has been widely used in database literature [32].

Conceptually, L-Store only has "local" transactions execution in the sense that all transactions hold all the data in the same place. This enables the transaction processing to be logged efficiently. Unlike other shared-nothing databases that require multiple log writes during the 2PC as shown in Figure 1, L-Store only writes the update data log once at the commit phase for each transaction. Two important factors ensure the correctness of this logging mechanism. First, there is no need for distributed synchronization during the commit phase, as all the data and updates are in local storage. Second, partial update of the transaction during any node crash is not visible to transactions at other nodes, which ensures that no rollback operation is needed during failure recovery.

For checkpointing, adopting the distributed storage in L-Store simplifies the mechanism. The system makes periodic checkpoint to the distributed storage. When node crash happens, the latest version of the checkpoint and the data logs are utilized to perform the recovery together.

We employ Zookeeper [18] to detect node crashes and coordinate the recovery. When one node S_i crashes, how to handle the failed message delivery has been discussed in Section 3.3. Now, we present the recover mechanism of handling another two types of data. The first type is the data that is currently owned by S_i. The other type is the owner table that traces which nodes own the data initially from S_i.

Recovering owned data. When S_i crashes, the data in S_i are lost from the volatile storage. These data can be recovered by using the latest checkpoint and the transaction logs. The logs are replayed from the latest checkpoint to recreate the state of the node prior to the crash.

Recovering owner table. When S_i crashes, the owner table is lost as well. In order to recover the owner table, Zookeeper sends a request to every node to write the data that belong to S_i into the distributed storage. When a new node is ready to replace the failed S_i, it would retrieve all these data from the distributed storage on demand, in which case the new node becomes the owner for those data and starts to build its own owner table again.

For more details on how the LEAP design handles data loss, interested readers may refer to Appendix D.

5. PERFORMANCE EVALUATION

In this section, we present results of an experimental study to investigate the effectiveness of LEAP. We implement the LEAP-based OLTP engine in Java as a stand-alone system, namely *L-Store*, and compare it with the state-of-the-art distributed in-memory OLTP system, H-Store [21], whose distributed transaction processing relies on 2PC. L-Store supports multi-threaded processing for database partitions where the number of threads being used is independent of the number of partitions deployed. In contrast, H-Store applies single-threaded processing for each partition [21, 20, 26]. Towards fair comparison, we additionally integrate the core of L-Store into H-Store to replace its 2PC-based transaction processing while retaining the single-threaded processing for each partition. We term this modified H-Store as H^L*-Store*. The comparison between H^L-Store and H-Store aims to verify the benefits of the proposed LEAP strategy under the same degree of parallelism in transaction processing. The comparison between L-Store and H-Store further shows the superiority of LEAP-based transaction processing when a larger number of threads is exploited to optimize the system resource utilization.

5.1 Experimental Setup

Environment. Our experiments are conducted on an in-house cluster with 64 nodes. Each node consists of an Intel Xeon X3430 @ 2.4 GHz CPU running Centos 5.11 with 8 GB memory. The cluster is connected using 10 Gigabit Ethernet. By default, we use 16 nodes to run the experiments.

Workloads. Two benchmarks are used throughout the experiments: TPC-C and YCSB.

The TPC-C benchmark [1] is an industry-standard benchmark for evaluating the performance of OLTP systems. It consists of nine tables and five stored procedures that simulate a warehouse-centric parts-supply application. The specification of TPC-C benchmark [1] defines the workload as five stored procedures. In particular, the `NewOrder` and `Payment` transactions (comprising 88% of the total workload as defined in the specification) contain remote data access to the `CUSTOMER` and `STOCK` tables, and the remaining types of transactions interact with local data only. Thus, to focus on the study of distributed transactions, we only generate the `NewOrder` and `Payment` transactions with uniform distribution in the experiments. Meanwhile, we control the percentage and access pattern of remote data access in these two types of transactions for evaluation purposes.

The Yahoo! Cloud Serving Benchmark (YCSB) [8] was initially designed for testing key-value data stores. It consists of a single table and two key-value operations (read-/update). The table is partitioned based on the primary key. We extend the benchmark to add support for multistep transaction that is configurable for the number of intermediate results exchange among nodes. This is implemented by batching several key-value operations into steps and enforcing that the search keys of each step (except the first step) are derived from the result of the prior step. Each step involves data access to one arbitrarily chosen remote partition.

Metrics. The throughput is measured by counting the number of committed transactions within a time unit. The latency is measured by capturing the time span between transaction initialization and commit, which includes all the retrials of a transaction.

Settings. For the TPC-C benchmark, we deploy one warehouse per partition, and the total number of warehouses is proportional to the number of nodes used in the experiments. Figure 7 shows the impact of performance of H-Store

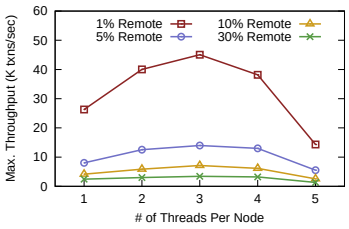
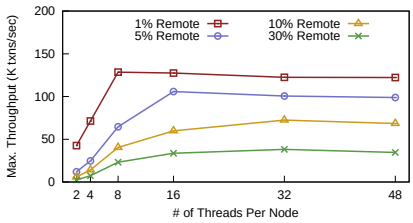
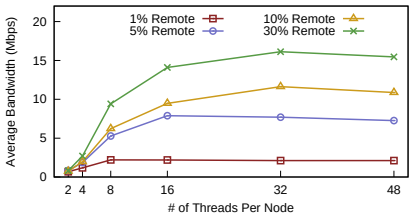


Figure 7: Impact of the number of threads used in H-Store.



(a) Throughput.



(b) Bandwidth.

Figure 8: Impact of the number of threads used in L-Store.

by varying the number of partitions deployed on a 4-node cluster. We show results for 1%, 5%, 10% and 30% remote data access. Recall that the single-threaded execution model in H-Store implies the number of threads per node for transaction processing equal to the number of partitions deployed in that node. From the results, we can see that with 4 cores in each node, H-Store achieves best performance by using one partition per core and allowing one core to do the system coordination, which is consistent with the setting from [21, 26]. Therefore, in our experiments, we deploy three partitions in each node for H-Store to perform optimally. Accordingly, we also deploy three partitions per node for both L-Store and H^L-Store [3].

For pure performance evaluation on transaction processing and transaction commit, we remove the settings of admission control, user initiated abort, transaction sleeping and timeout in the experimental workloads. Moreover, for all three systems (L-Store, H^L-Store and H-Store), whether a partition is local or remote is based on the node it resides in rather than its assigned core. We strictly enforce the remote data access to be inter-node with respect to the initial state of database partitioning (since LEAP would migrate data during transaction processing).

5.2 Impact of Multi-Threaded Processing

As L-Store supports multi-threaded transaction processing, this set of experiments aim to study how the usage of threads affects the system throughput and bandwidth.

Throughput. Figure 8.a provides the maximum throughput of L-Store for the TPC-C workload with 1%, 5%, 10% and 30% of remote data access when we vary the number of threads per node from 2 to 48 with the number of partitions per node unchanged. From the result, we can see that the throughput initially increases with increasing number of threads (from 4 to 32) but decreases or remains stable beyond 32 threads. For any thread running a distributed transaction, it is suspended during ownership transfer. Thus using more threads for more concurrent execution would reduce CPU idle time so that computation resources can be better utilized towards higher throughput. However, deploying too many threads would incur significant overhead of thread scheduling. Moreover, higher concurrency would bring in higher contention of data, which could lead to more transaction abort due to deadlock prevention. Transaction abort not only wastes the CPU cycles spent on partial processing but also consumes extra CPU cycles to perform transaction rollback. Hence, setting the number of threads to an appropriate range could balance the trade-off between efficiency and effectiveness of computation resource usage.

[3] H^L-Store is therefore set with three threads per node for transaction processing.

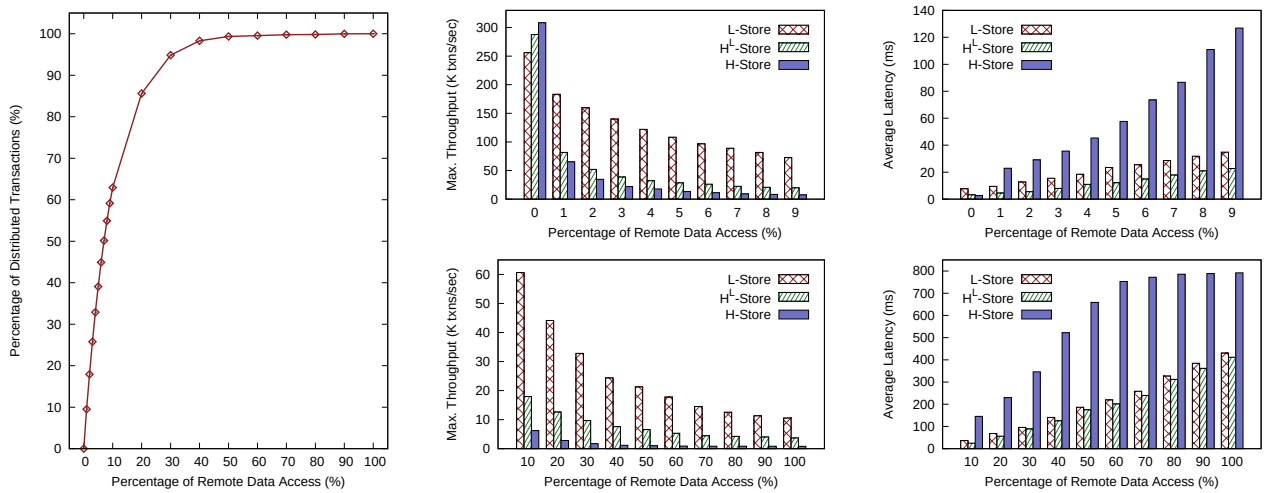
Based on the result shown in Figure 8.a, we set 32 threads per node as the default setting of L-Store for the rest of the experiments.

Bandwidth Usage. To study the communication cost incurred by inter-node message passing in L-Store, we further examine the bandwidth usage under different number of threads. Note that the cost of ownership transfer dominates the overall bandwidth. We gather three insights from the result given by Figure 8.b. First, higher percentage of remote data access results in higher bandwidth usage. This is expected as more remote data accesses incur more ownership transfers. Second, the bandwidth usage shows a similar curve as the throughput with respect to varying number of threads. This is because the bandwidth usage is basically proportional to the number of remote data accesses within a time unit, which is implicitly constrained by the system throughput. Third, the network traffic generated by the ownership transfer is small and acceptable. The result in Figure 8.b indicates that even with 32 threads, the bandwidth usage for running the workload with 30% remote data access is around 17 Mbps. With modern network infrastructure such as 10 Gigabit Ethernet, the bandwidth usage in L-Store is not likely to become a bottleneck even in a large-scale system setting.

5.3 Impact of Distributed Transactions

Next, we study the impact of distributed transactions by observing the maximum throughput and mean latency when the percentage of remote data access varies from 0% to 100% (without preserving locality of data access) for the TPC-C workload. Figure 9.b and Figure 9.c show that L-Store and H^L-Store have comparative throughput and latency with H-Store when all transactions are local, i.e., 0% remote data access [4]. On the other hand, with increasing percentage of remote data access, the throughput for all three systems decreases while the latency increases. This is expected as executing local transactions is much faster than processing distributed transactions. However, we observe that L-Store and H^L-Store outperform H-Store by a wide margin once distributed transactions are involved. For instance, L-Store (resp. H^L-Store) exhibits 2.5x to 20x (resp. 1.2x to 6x) speedup over H-Store when the percentage of remote data access varies from 1% to 50% as shown in Figure 9.b. As analyzed in Section 3.4.1, the performance of 2PC-based transaction processing is sensitive to the proportion of distributed transactions in the workload. Figure 9.a indicates that the percentage of distributed transactions grows rapidly with the increment of remote data access. For example, 1% remote data access results in about 10% distributed trans-

[4] Appendix G provides additional evaluation on the performance of local transaction processing.



(a) Distributed transactions. (b) Throughput. (c) Latency.

Figure 9: Impact of the percentage of distributed transactions.

Table 1: Transaction abort and trials in L-Store.

Remote	Abort Rate	1 Trial	2 Trials	$\geq$ 3 Trials
1%	3.43%	96.89%	2.95%	0.16%
5%	3.71%	96.52%	3.34%	0.14%
10%	3.49%	96.74%	3.15%	0.11%
30%	2.68%	97.62%	2.33%	0.05%

actions, 10% remote data access results in about 63% distributed transactions, and 50% of remote data access results in nearly 100% distributed transactions. The rapid increase in the number of distributed transactions greatly raises the latency of H-Store (as shown in Figure 9.c) and consequently leads to the dramatic decrease of its throughput (as shown in Figure 9.b). This is caused by the frequent invocation of the expensive 2PC protocol in H-Store for committing distributed transactions. On the contrary, we observe that the throughput of L-Store and H^L-Store decreases at a much slower rate as the percentage of remote data access increases. This confirms that LEAP enables L-Store and H^L-Store to handle distributed transaction processing much more efficiently than H-Store.

The throughput gap between L-Store and H^L-Store is due to the CPU utilization. When the maximum throughput is reached, both L-Store and H-Store utilize CPU at nearly 100%, whereas H^L-Store utilizes CPU at around 30%. This result is consistent with Figure 8.a where the LEAP-based processing suffers from blocking due to the usage of locks. On the other hand, H^L-Store provides lower latency than L-Store. This is because the processing in H^L-Store does not require thread scheduling.

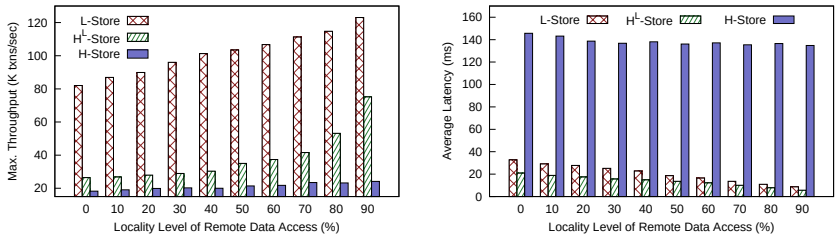
We further evaluate the concurrency control of the LEAP-based engine. Taking L-Store as the example, Table 1 shows the transaction abort rate and the distribution of different number of transaction trials for transactions to successfully commit. The results are collected after the system is saturated, i.e., running with the maximum throughput. As can be seen, the abort rate is lower than 4% in all the test instances. The aborts are caused by the Wait-Die policy in resolving data contention (i.e., deadlock prevention and ownership transfer failure). Moreover, every issued transaction gets committed eventually. For all the committed transactions, over 96% successfully commit without any prior abort, and around 3% get to commit in their second trial of execution. The maximum number of transaction trials observed in the experiments is 6. These results clearly show that LEAP is effective in not only having low abort rates, but the number of retrials for aborted transactions is also acceptable.

5.4 Impact of Locality of Remote Data Access

Using the TPC-C workload containing remote data accesses, we further study how the locality of these remote data accesses affect the performance. Recall that a remote data access is considered to be with locality if the data being requested at the remote partition are exclusively accessible to the requesting node, and a remote data access is considered to be without locality if the data is accessed randomly over the global database. This set of experiments are conducted using the TPC-C workload with 10% remote data access. Figure 10 provides the throughput and latency of L-Store, H^L-Store and H-Store as we vary the locality level from 0% to 90%. The locality level is simulated as the probability of remote data access being with locality during transaction generation.

As can be seen from the results, LEAP-based systems (i.e., L-Store and H^L-Store) outperform 2PC-based system (i.e., H-Store) by a big margin when the locality level is high. This is expected as LEAP enables the migrated data to be reaccessed by subsequent transactions in the same node without additional remote requests. Consequently, the percentage of remote data access is significantly reduced. High locality of remote data access benefits high possibility of trivially turning a remote data access into a local data access without issuing any remote request once the data is available in the requesting node due to prior migration. On the other hand, while the performance of LEAP-based processing (i.e., in L-Store and H^L-Store) increases with the increment of locality of remote data access, the change of locality is agnostic to the 2PC-based processing (i.e., in H-Store). This is because, in the 2PC-based processing, remote data access would always remain remote as the data would not be automatically repartitioned.

For micro evaluation on ownership transfer of LEAP, interested readers may refer to Appendix E.



(a) Throughput. (b) Latency.

Figure 10: Impact of the locality level of remote data access.

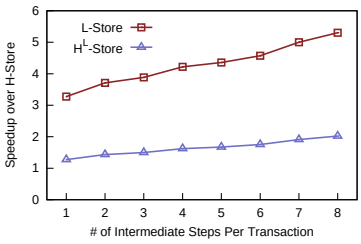


Figure 11: Impact of the number of intermediate results.

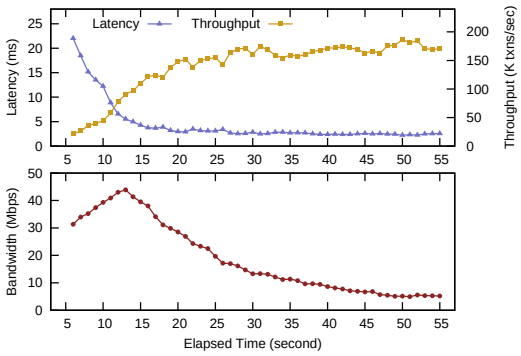


Figure 12: Automatic data repartitioning.

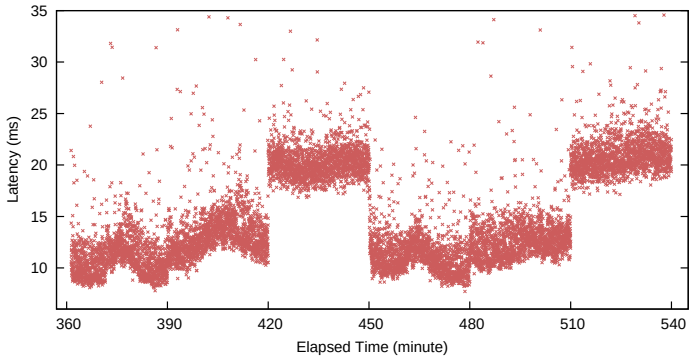


Figure 13: Impact of access pattern.

5.5 Impact of Intermediate Result Transfer

Next, we study the performance impact of intermediate result transfer within the distributed transaction processing. This set of experiments are conducted by running our extended YCSB workload with 10% remote data accesses. Figure 11 shows the speedup of L-Store and H^L-Store over H-Store, when we vary the number of steps involving intermediate result transfer within each transaction. Each step involves one remote data access and one tuple of intermediate result. From the result, we can see that as the amount of intermediate result transfer increases, the LEAP-based processing exhibits increment of speedup over the 2PC-based processing. For instance, the speedup of L-Store (resp. H^L-Store) over H-Store increases from 3.2x (resp. 1.2x) to 5.1x (resp. 1.9x) when the number of intermediate result steps varies from 1 to 8. The performance degradation of the 2PC-based processing is caused by the increased inter-node communication for intermediate result transfer. On the other hand, since LEAP always gathers the required data of a transaction for processing within a single thread, it naturally avoids the transfer of intermediate results within the transaction. This further confirms the superiority of LEAP-based distributed transaction processing.

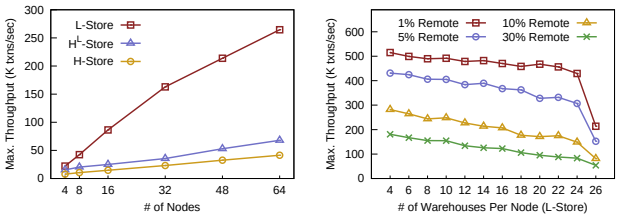
5.6 Impact of Access Pattern

As LEAP exploits data migration to convert distributed transactions into local transactions, a side-effect of data migration in LEAP is to automatically repartition the database with respect to the workload. We conduct an experimental study to verify the effectiveness of database auto-repartitioning within the LEAP-based transaction processing. Figure 12 illustrates the real-time latency, throughput and bandwidth usage changes when H^L-Store runs the TPC-C workload with initially 10% remote data access and 100% locality of remote data access. Transaction generator feeds in 18,000 transactions per second. At the beginning, the latency is relatively high due to the frequent ownership transfer. This makes sense because the distributed transactions are aggressively migrating data from remote partitions to local partitions. As expected, as time goes by, the bandwidth usage drops dramatically and reduces to almost zero in the end. This confirms that all the remote data have been migrated to the same node with respect to the locality of remote data access. As a consequence, subsequent transactions contain no remote data access and thus the system retains high throughput and low latency.

Besides the locality of remote data access, the pattern of data access will also impact the system performance. Figure 13 shows the real-time latency along L-Store's long-time execution with periodic alternation of three data access patterns: each remote data access is limited to (1) one, (2) two and (3) all stated partitions respectively. Each access pattern lasts for 30 minutes. The result is a segment sample of a 3-hour run. As can be seen, LEAP can adapt to the changes of data access pattern towards good performance. This results from the fact that data contention increases from (1) to (3). In case (1), each data ownership is mainly accessible from two nodes, while it may be accessed by three nodes (resp. all nodes in the cluster) in case (2) (resp. case (3)). When the number of nodes requesting the same data increases, the contention increases accordingly and thus incurs more overhead of ownership transfer.

5.7 Scalability

Finally, we evaluate the scalability of L-Store, H^L-Store and H-Store when the size of the cluster changes. Figure 14.a shows the throughput changes for running the TPC-C workload with 10% remote data access as the number of nodes varies from 4 to 64. The results illustrate that L-Store scales much better than H-Store. For instance, the throughput of H-Store with 64 nodes is only 8 times faster than the



(a) Over nodes. (b) Over database sizes.

Figure 14: Scalability.

one with 4 nodes. On the other hand, we observe that the throughput of L-Store grows almost linearly when the number of nodes increases from 4 to 32. Although H^L-Store manifests similar scalability as H-Store, H^L-Store greatly underutilizes the CPU resources due to its restricted usage of threads (recalling that both H^L-Store and H-Store allocate a single thread for each partition). On the other hand, both L-Store and H-Store fully utilize the CPU resources when reaching the maximum throughput.

Furthermore, we evaluate the system scalability with increasing size of database. To this end, we still use the TPC-C workload as above but raise the database size beyond 3 warehouses per node. The experiments are run on a 64-node cluster by varying the amount of warehouses deployed. We first test H-Store and find that it cannot scale beyond 320 warehouses (i.e., 5 warehouses per node). This is consistent with the result as shown in Figure 7. Next, we test L-Store by augmenting the database size until the system reaches its limit of in-memory processing. The results are shown in Figure 14.b, with the percentage of remote data access set to 1%, 5%, 10% and 30%. As can be seen, with 512 GB memory capacity of the cluster, L-Store can scale out up to 1536 warehouses (i.e., 24 warehouses per node) whose data volume is around 450 GB in total. This confirms that L-Store is highly scalable when the cluster memory is sufficient.

6. RELATED WORK

There has been much interest in developing scalable distributed databases. One class of systems achieves good scalability by removing the transactional support, such as Dynamo [13] and Cassandra [23]. Another batch of systems only provides limited transactional support. For example, Bigtable [7] supports single-row transactional updates, and Megastore [4] offers transaction access over a small subset of a database. These systems provide linear outward scalability by sacrificing the complete transactional support.

H-Store [21] and its commercial successor VoltDB [27] are distributed main-memory systems that support ACID properties through data partitioning and replication. While these systems process local transactions efficiently, their performance degenerates with increasing number of distributed transactions due to usage of the expensive 2PC protocol.

Calvin [30] is a transaction scheduling and data replication layer that uses a deterministic ordering guarantee to reduce the contention costs associated with distributed transactions. Calvin does not use any distributed commit protocols; instead, it employs a deterministic locking mechanism that executes (sub)transactions in the same order across all nodes. However, this results in high transaction delay as the transactions can only be executed after agreeing on a transaction execution order.

L-Store is similar in spirit to G-Store [11]. Both systems transfer all the keys forming a group to a single node in order to allow efficient multi-key access. However, G-Store is specially designed for web application (e.g., online gaming) rather than OLTP workload. It requires the group to collect all the keys before the execution, which does not work in OLTP workload as the keys may not be known in advance. In G-Store, transactions are limited to one group and the techniques are tailored for long-lived transactions, while L-Store is designed to OLTP workload where transactions are short-lived.

Our work is also related to the research on data migration and database partitioning. Zephyr [15] performs live migration in a shared-nothing transactional database. It aims to reduce the system downtime while migrating tenants between hosts. Pavlo et al. [26] proposed an automatic database partitioning algorithm based on an adaptation of the neighborhood search technique, and a cost model that estimates the coordination cost and load distribution for a sample workload. E-Store [28] provides a two-tier data placement strategy to prioritize hot data and adopts a live reconfiguration technique [16] to repartition the database. ElasTraS [12] recursively partitions the database with tree-structured scheme towards minimizing remote data accesses, and adopts the 2PC protocol to handle distributed transactions. L-Store differentiates itself from these works by automatically transferring the data to the computation according to the transaction execution request.

Non-2PC distributed transaction processing has been studied in the context of shared-data databases [6, 24]. For example, Loesing et al. [24] proposed a design of distributed transaction processing by leveraging the in-memory shared-data storage. It shares similarity with L-Store in the mechanism of transaction processing such that it gathers all required data of a transaction from the storage to a node and processes the transaction locally. With shared-data storage, this design can achieve load balancing since the execution of each transaction is independent of the location of data. However, this design requires the shared-data storage to provide atomic data access primitives to resolve contention among parallel updates. This renders the solution less applicable to vanilla in-memory storage systems that are not necessary to guarantee atomicity of data access. In contrast, in L-Store, the logic of transaction processing and concurrency control are decoupled from the underlying storage layer, making L-Store friendly to all kinds of distributed in-memory storage. In Appendix I, we provide an experimental comparison between LEAP and the scheme in [24].

7. CONCLUSION

We proposed a distributed transaction management scheme called LEAP which facilitates low latency and good scalability. LEAP always converts a distributed transaction into a local transaction to eliminate the expensive commit protocol. Based on LEAP, we developed a distributed in-memory OLTP engine L-Store with full ACID support. L-Store enables a good scalability with a shared-nothing database engine and no single-node failure problem by employing a distributed in-memory storage. Furthermore, we presented a lock-based distributed concurrency control scheme to handle concurrent requests in L-Store. The experimental evaluation highlighted the superiority of the LEAP-based distributed transaction processing over 2PC-based systems like H-Store.

Acknowledgments

We would like to thank Dawei Jiang for initiating the idea of this research in February 2014, and Chang Yao and Meihui Zhang for their early tests. We would also like to thank our reviewers and shepherd of the paper, and H. V. Jagadish, for their insightful feedback that helped improve the paper. This research is funded by the National Research Foundation Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme with the SP2 project of the E2S2 programme, and the National Research Foundation, Prime Minister's Office, Singapore under its Competitive Research Programme (CRP Award No. NRF-CRP8-2011-08).

APPENDIX

A. REFERENCES

[1] Tpc-c. http://www.tpc.org/tpcc/.

[2] A. Adya et al. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proc. of SIGMOD*, 1995.

[3] P. Bailis et al. Coordination avoidance in database systems. In *Proc. of VLDB*, 2015.

[4] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, 2011.

[5] Y. Cao et al. Es2: A cloud data storage system for supporting both oltp and olap. In *Proc. of ICDE*, 2011.

[6] S. Chandrasekaran et al. Shared cache - the future of parallel databases. In *Proc. of ICDE*, 2003.

[7] F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, 2008.

[8] B. F. Cooper et al. Benchmarking cloud serving systems with ycsb. In *Proc. of SOCC*, 2010.

[9] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *Proc. of OSDI*, 2012.

[10] C. Curino et al. Schism: A workload-driven approach to database replication and partitioning. In *Proc. of VLDB*, 2010.

[11] S. Das et al. G-store: A scalable data store for transactional multi key access in the cloud. In *Proc. of SOCC*, 2010.

[12] S. Das et al. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems*, 38(1):5:1–5:45, 2013.

[13] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP*, 2007.

[14] D. E. Difallah et al. Oltp-bench: An extensible testbed for benchmarking relational databases. In *Proc. of VLDB*, 2014.

[15] A. J. Elmore et al. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proc. of SIGMOD*, 2011.

[16] A. J. Elmore et al. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proc. of SIGMOD*, 2015.

[17] C. Hewitt et al. A universal modular actor formalism for artificial intelligence. In *Proc. of IJKAI*, 1973.

[18] P. Hunt et al. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. of USENIX ATC*, 2010.

[19] D. Jiang et al. epic: An extensible and scalable system for processing big data. In *Proc. of VLDB*, 2014.

[20] E. P. Jones et al. Low overhead concurrency control for partitioned main memory databases. In *Proc. of SIGMOD*, 2010.

[21] R. Kallman et al. H-store: A high-performance, distributed main memory transaction processing system. In *Proc. of VLDB*, 2008.

[22] J. F. Kurose et al. *Computer networking: a top-down approach*. Pearson Education, 2012.

[23] A. Lakshman et al. Cassandra: Structured storage system on a p2p network. In *Proc. of PODC*, 2009.

[24] S. Loesing et al. On the design and scalability of distributed shared-data databases. In *Proc. of SIGMOD*, 2015.

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

[26] A. Pavlo et al. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proc. of SIGMOD*, 2012.

[27] M. Stonebraker et al. The voltdb main memory dbms. *IEEE Data Engineering Bulletin*, 36(2):21–27, 2013.

[28] R. Taft et al. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. In *Proc. of VLDB*, 2015.

[29] K.-L. Tan et al. In-memory databases: Challenges and opportunities from software and hardware perspectives. *ACM SIGMOD Record*, 44(2):35–40, 2015.

[30] A. Thomson et al. Calvin: Fast distributed transactions for partitioned database systems. In *Proc. of SIGMOD*, 2012.

[31] G. Weikum et al. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Elsevier, 2001.

[32] C. Yao et al. Adaptive logging for distributed in-memory databases. *CoRR*, 2015.

[33] H. Zhang et al. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.

B. DATA ACCESS VIA NON-PRIMARY KEYS

Our proposal in LEAP with data access via primary key can be easily extended for transactional workloads with data access via non-primary keys using secondary indexes. To this end, existing techniques of building and manipulating distributed index can be exploited to construct the secondary indexes. Each secondary index entry records the primary key of the data record so that the secondary indexes facilitate identifying the primary key via non-primary keys. As a consequence, data access via the non-primary key is done in two steps: retrieving the primary key through the secondary index and then accessing the data through the LEAP protocol using the obtained primary key. Therefore, the corresponding process of ownership transfer can reduce to the one described in the paper.

C. SYMBOLIC NOTATIONS

Table 2 lists the notations used in Section 3.4.1.

Table 2: List of notations.

Sym.	Description
T	Latency expectation of a transaction.
n	Number of data accesses within a transaction.
i	Number of steps dependent on intermediate results.
R	Event of remote data access.
L	Event of remote data access appearing locality.
X	Cost of one-time network communication.

D. FAULT TOLERANCE SCHEME

As an addendum to Section 4.2, we shall further elaborate on the fault tolerance of LEAP, particularly on its handling of data and message loss.

Handling Message Loss. The LEAP design handles message loss with two levels of guarantees. First, in the transmission level, LEAP relies on TCP for inter-node message passing, which has been shown to be reliable [22].

Second, in the protocol level, LEAP adopts a timeout-and-resend strategy. Every owner/transfer request is accompanied with a timeout setting. When there is a timeout waiting for a corresponding response, the owner/transfer request will be resent. Note that the response to an owner request is either the requested data or a transaction abort signal, and the response to a transfer request is an inform message indicating whether the ownership transfer succeeds or not. With the resent requests, the process of ownership transfer can be replayed once message loss occurs. However, resending requests may introduce the issue of duplicate requests. To address this issue in LEAP, duplicate messages are filtered out at the receiver site based on the processing logic. Referring to Figure 6, we enumerate the cases of duplicate elimination with respect to the four types of messages in the general case of ownership transfer (i.e., R-P-O).

- Suppose S_2 receives an owner request from S_1, e.g., $S_1(T_1.t)$ for K. If S_2 detects that $S_1(T_1.t)$ is already in the waiting queue for K or S_1 is already the owner of K, the ownership request will be ignored. Otherwise, S_2 will add $S_1(T_1.t)$ into the waiting queue for K, or send a corresponding transfer request to S_3, or send an abort signal of T_1 to S_1.
- Suppose S_3 receives a transfer request $S_1(T_1.t)$ for K from S_2. If S_3 detects that $S_1(T_1.t)$ is already in the waiting queue for K, the transfer request will be ignored. Otherwise, S_3 will either transfer (a copy of) K to S_1 or send an abort signal of T_1 to S_1. Note that after K is transferred out of S_3, S_3 still maintains a backup of K for a longer timeout until deletion.
- Suppose S_1 receives the requested K or an abort signal of T_1. After T_1 completes (i.e., commits or aborts), S_1 will inform S_2 whether the ownership transfer of K succeeds or not. Note that no duplicate elimination is performed in this case.
- Suppose S_2 receives an inform message from S_1 for K. If S_2 detects that the ongoing ownership transfer is not for S_1, the inform message will be ignored. Otherwise, based on the inform message, S_2 will update the owner table entry for K and invoke a pending owner request, if any, from the waiting queue for K.

In brief, the message-passing-based ownership transfer can be guaranteed by the reliable transmission of TCP and the timeout-and-resend strategy of LEAP.

Handling Data Loss. Considering the general case of ownership transfer (i.e., R-P-O), we discuss all possible failures in requester, partitioner and owner during the protocol. Note that the protocol starts with the owner request being sent by the requester and ends with the inform message being confirmed by the partitioner. Our current implementation of L-Store only considers the mechanism of handling single-node failure.

- *Requester failure* may occur in the cases based on the status of receiving the response message (associating either the requested data or a transaction abort signal).
- *Partitioner failure* may occur in the cases based on the status of receiving the owner request, sending the transfer request and receiving the inform message.
- *Owner failure* may occur in the cases based on the status of receiving the transfer request and sending the response message.

Due to space constraint, we only elaborate on two cases, and the remaining ones follow the similar processing pattern. For example, suppose the partitioner fails before the owner request arrives. In this case, once the requester waiting for the response message is timed out, it will resend the owner request. This procedure repeats until the partitioner resumes. Once the partitioner resumes and receives the owner request, the protocol proceeds as normal. For another example, suppose the requester fails before the response message arrives. In this case, once the partitioner waiting for the inform message is timed out, it will resend the transfer request and then the owner will resend the response message with (a backup copy of) the data. This procedure repeats until the requester resumes. Once the requester resumes and receives the response message, the protocol proceeds as normal.

Moreover, we specially elaborate on node recovery with respect to its data. While the data owned by the failed node can be recovered through the log, it is possible that some data have been transferred out prior to the node failure:

- If the data transferred out belongs to the failed node (i.e., the failed node is the partitioner of the data), then the new owner of the data will flush the data (after its running transaction completes) to the distributed storage after it is informed about the failure. When the failed node resumes, it will retrieve the data from the distributed storage on demand (i.e., when the data is requested but the owner table entry is missing, which by default means the data locates at the partitioner). In other words, the data is "indirectly" transferred back to the partitioner.
- If the data transferred out does not belong to the failed node (i.e., the failed node is not the partitioner of the data), then the data is owned by a new owner and the recovery of the failed node (i.e., the old owner) does not affect the protocol.

E. EVALUATING OWNERSHIP TRANSFER

Figure 15 illustrates the breakdown of ownership transfer cases referring to Figure 4. The experiment is conducted on a 16-node cluster by running L-Store with the TPC-C workload. Each result is measured after the system has been saturated for 15 minutes. Figure 15.a and Figure 15.b show

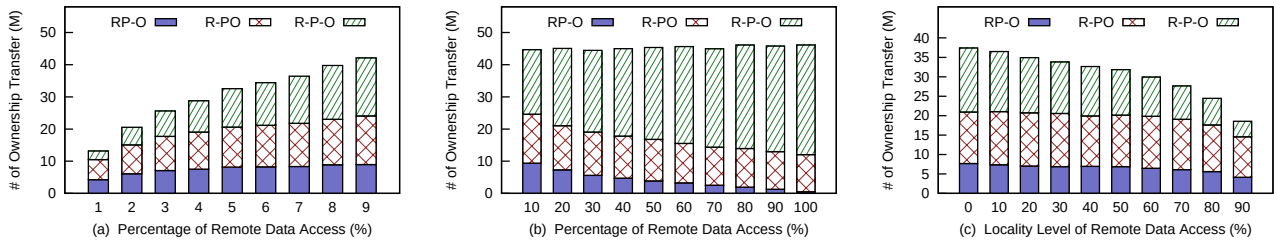


Figure 15: Breakdown of ownership transfer cases. (Case types referring to Figure 4)

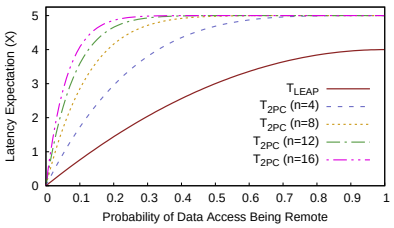


Figure 16: Comparison of latency expectation.

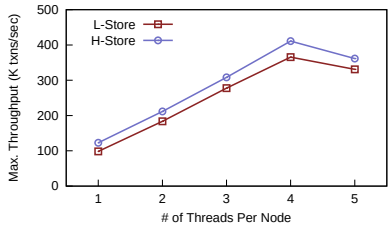


Figure 17: Performance of local transaction processing.

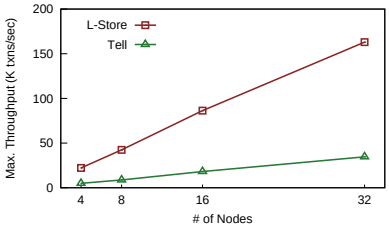


Figure 18: Comparison of scalability with Tell [24].

the results when varying the percentage of remote data access from 1% to 100% with the locality level set to be 0%. When remote data accesses are rare (e.g., less than 10% as in Figure 15.a), the number of ownership transfers increases obviously along the increment of remote data access. When remote data accesses become frequent (e.g., more than 10% as in Figure 15.b), the total amount of ownership transfers remains nearly the same in spite of the increment of remote data access. This is due to the exploitation of locality of remote data access in LEAP. (Although the locality of all data accesses in Figure 15.a and Figure 15.b are not advisedly preserved, the randomization of data access naturally contributes some degree of locality.) Moreover, the portion of R-P-O of ownership transfer increases steadily along the increment of remote data access. This explains the performance loss of LEAP-based transaction processing when remote data access rises but the total amount of ownership transfer remains stable, because R-P-O of ownership transfer is most expensive comparing with RP-O and R-PO.

Furthermore, Figure 15.c shows the results when varying the locality level from 0% to 90% with 10% remote data access. The rise of locality decreases the amount of ownership transfer, especially those of R-P-O. This confirms the gain of the LEAP protocol due to the locality of data accesses.

F. LATENCY EXPECTATION

Figure 16 illustrates the comparison of latency expectation between LEAP-based and 2PC-based transaction processing. The plots are based on Equation 1 and Equation 2 with $\alpha = 4$ and $\beta = 5$ respectively. The locality of remote data access is ignored, i.e., $P(L|R) = 0$. Each transaction only contains single step (i.e., $i = 1$). Recall that n is the number of data access within a transaction and X is the estimated average cost of one-time network communication.

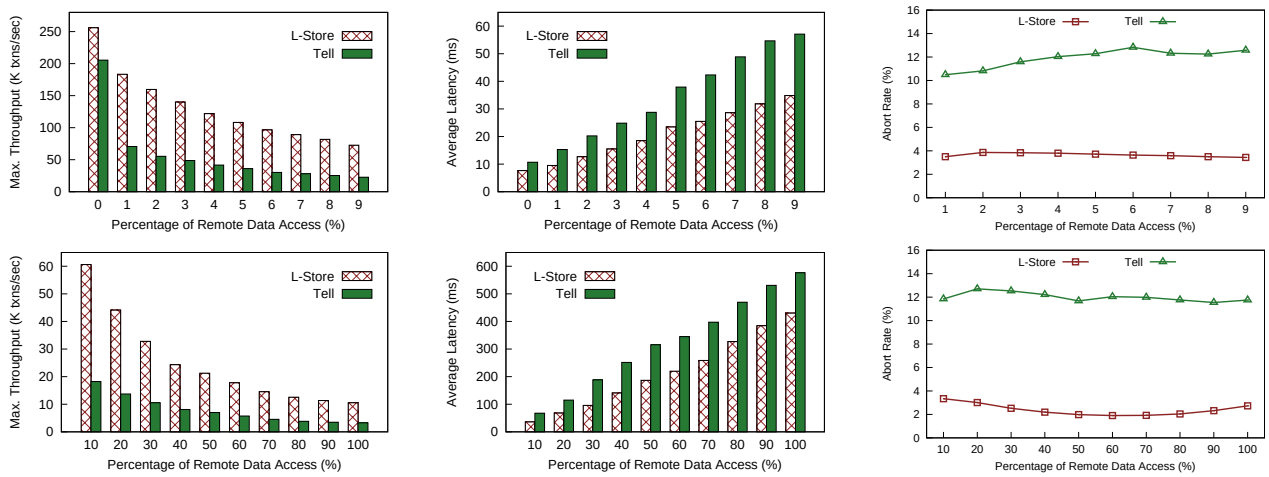
G. LOCAL TRANSACTION PROCESSING

To evaluate how local transactions affect the overall performance, Figure 17 provides the result of maximum throughput when L-Store and H-Store process the TPC-C workload with local transactions only. The experiment is conducted on a 16-node cluster. The percentage of remote data access is set to 0% (i.e., all generated transactions are local), and the number of threads per node varies from 0 to 5. From the result we obtain two insights. First, H-Store outperforms L-Store with local transaction only workloads. The performance difference is mainly contributed by the overhead of locking. To handle data contention, H-Store applies the warehouse-granule locks along with its single-threaded processing for each warehouse, whereas L-Store applies the tuple-granule locks in the design to support multi-threaded processing. Moreover, once H-Store figures out that a transaction is local, it does not actually perform locking [20, 26] since only one thread is manipulating the data in the same warehouse and thus no contention exists. This makes H-Store more efficient than L-Store when processing local transactions. Second, the scalability curves of both H-Store and L-Store share a similar pattern. Especially, the performance peak appears with 4 threads per node. This is rational because each node is equipped with 4 physical cores. Specially, for H-Store, the thread for coordinating the processing of distributed transactions is always idle since no distributed transaction is generated. As a consequence, all the cores can fully participate in processing local transactions.

Additionally, we measure the latency under the same settings as above. With the number of threads per node varying from 1 to 4, the latency remains around 3.2 ms and 2.6 ms for L-Store and H-Store respectively. This is because each thread processes local transactions independently and no thread scheduling is performed over the cores. However, with 5 threads per node, the latency increases to 3.5 ms for L-Store and 3.1 ms for H-Store due to the overhead of thread scheduling.

H. MEMORY USAGE OF OWNER TABLE

Figure 20 shows the memory consumption of owner table when L-Store runs the TPC-C workload in a 16-node cluster. The percentage of remote data access varies from 0% to 100%, and the locality level of remote data access is set to 0% (i.e., no locality is preserving). Each result is measured



(a) Throughput. (b) Latency. (c) Abort Rate.

Figure 19: Comparison of performance with Tell [24].

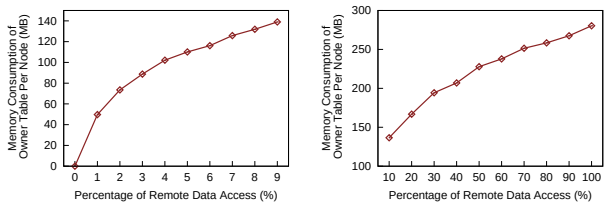


Figure 20: Memory consumption of owner table.

after the system has been saturated for 15 minutes. As can be seen from the result, the owner tables do not consume any memory when the system processes local transactions only (i.e., with 0% remote data access). This confirms the memory optimization as discussed in Section 3.2. Furthermore, the memory consumption increases along the increment of remote data access. This is expected with respect to the ownership transfer protocol.

I. COMPARISON WITH SHARED-DATA OLTP SYSTEM

Here, we shall evaluate L-Store against a state-of-the-art shared-data OLTP system, Tell [24]. For Tell, each physical node consists of two logical nodes: one processing node and one storage node [5]. Additionally, two nodes are exclusively used as the commit manager and the management node, and they are not counted in the experiments. We conduct experiments by running the TPC-C workload.

We first compare the performance with various settings of remote data access. Figure 19 shows the results of throughput, latency and abort rate by varying the percentage of remote data access from 0% to 100%. L-Store outperforms Tell in throughput and latency due to the following two reasons. First, although L-Store and Tell share similarity in turning every distributed transaction into a local one for processing, Tell does not exploit any locality of data access since it always retrieves data acquired by the transactions from the storage. Second, Tell resolves write-contention in the storage layer when transactions commit. This leads to high abort rate of Tell in comparison with L-Store, as shown in Figure 19.c. The high abort rate of Tell inversely affects its throughput and latency, since the throughput refers to only the committed transactions and the latency includes all the retrials of the aborted transactions.

Next, we compare the scalability. The result in Figure 18 shows that Tell also exhibits nearly linear scalability as L-Store does. This is expected as the design of Tell is specially proposed for good scalability [24].

J. ASSIGNING GLOBAL TIMESTAMPS

Current implementation of L-Store assigns a timestamp to each issued transaction of the format of $\langle node_time, node_id \rangle$. Specifically, *node_time* is the local wall-time in nanoseconds when the transaction arrives at the execution node, and *node_id* is the unique node identifier. We assume each node can accept at most one transaction per nanosecond, which is practical in general. Thus, the timestamp of each transaction is globally unique.

The timestamp ordering is determined by the comparison between timestamps. For transactions in the same node, their timestamps are naturally ordered by the *node_time*'s. For transactions in different nodes, the timestamp ordering is primarily based on *node_time* and secondarily based on *node_id*. Although this is an easy-to-implement approach, it suffers from two disadvantages. First, all clocks of the nodes have to be synchronized to make the timestamp ordering among different nodes meaningful. This is known to be hard in practice, especially for maintaining the synchronization strictly all the time. In the experiments of this paper, we only synchronize the clocks at the beginning of each running instances. Second, the comparison based on *node_id* is biased towards nodes with identifiers of high preference. But its impact is limited since it is only used for breaking ties when the *node_time*'s appear to be equal.

An alternative solution is to assign a global order for each transaction through a central coordinator [20]. But the central coordinator could be a bottleneck of the system. Using multiple coordinators with synchronization protocols (e.g., loosely synchronized clocks [2] or TrueTime [9]) could mitigate the issue. We leave the optimized implementation to future work, and only adopt the simple method as described above in this paper.

[5] In the experiments, we only count the physical nodes.