Elastic scalable transaction processing in LeanXcale

Ricardo Jimenez-Peris, Diego Burgos-Sancho, Francisco Ballesteros, Marta Patiño-Martinez, Patrick Valduriez

Information Systems 108 (2022) 102043

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Elastic scalable transaction processing in LeanXcale[☆]

Ricardo Jimenez-Peris[a], Diego Burgos-Sancho[a,b,*], Francisco Ballesteros[c], Marta Patiño-Martinez[b], Patrick Valduriez[d]

[a] LeanXcale, Spain
[b] Universidad Politécnica de Madrid, Spain
[c] Laboratorio de Sistemas, URJC, Spain
[d] Inria, University of Montpellier, CNRS, LIRMM, France

ARTICLE INFO

Article history:
Received 24 June 2021
Received in revised form 26 March 2022
Accepted 28 March 2022
Available online 31 March 2022
Recommended by Dennis Shasha

Keywords:
Transaction processing
Transaction management
NewSQL database system
Scalability
Elasticity
Cloud
TPC-C

ABSTRACT

Scaling ACID transactions in a cloud database is hard, and providing elastic scalability even harder. In this paper, we present our solution for elastic scalable transaction processing in LeanXcale, an industrial-strength NewSQL database system. Unlike previous solutions, it does not require any hardware assistance. Yet, it does scales linearly to 100s of servers. LeanXcale supports non-intrusive elasticity and can move data partitions without hurting the quality of service of transaction management. We show the correctness of LeanXcale transaction management. Finally, we provide a thorough performance evaluation of our solution on Amazon Web Services (AWS) shared cloud instances. The results show linear scalability, e.g., 5 million TPC-C NewOrder TPM with 200 nodes, which is greater than the TPC-C throughput obtained by the 9th highest result in all history using dedicated hardware used exclusively (not shared like in our evaluation) for the benchmark. Furthermore, the efficiency in terms of TPM per core is double that of the two top TPC-C results (also the only results in a cloud).

1. Introduction

Recent data-intensive applications in the cloud require very high-throughput transactions over big data at unprecedented scale. Examples of such applications can be found in e-advertising, IT monitoring, IoT, smart grid, and industry 4.0. For instance, the AdWords business in e-advertising [2] is update intensive and can have millions of suppliers monitoring their maximum cost-per-click (CPC) bid for their offer and updating a petabyte database, while millions of other users and potential customers would perform search queries. IT monitoring on the other hand is insert intensive, with concurrent queries to recover full time series of monitored items (equipments, applications, etc.) or the last measures from millions of monitored items. Depending on the IT business, the data ingestion rates can be huge, ranging between 100 thousand rows per second to 100 million rows per second in databases that can reach petabytes of data.

To deal with such massive scales and adapt to the changing needs of applications in terms of resources, a cloud database system must provide scalability. This can be obtained through a combination of vertical scaling, by making servers bigger (e.g., adding processors, IO bandwidth, and memory) and horizontal scaling, by adding more scale-out servers in a shared-nothing cluster. Shared-nothing is cost-effective as servers can be off-the-shelf components connected by a network and can be used in public clouds. Thus, it can be fully exploited by a distributed database system [3], with data partitioning onto multiple server nodes as the basis for parallel data processing, and distributed transaction processing to update data partitions.

In order to adapt to workload changes and be able to process greater (or lower) transaction rates, scalability must also be elastic, by dynamically provisioning (or decommissioning) servers to the cluster and increasing (or decreasing) the global capacity. Elasticity requires live data migration [4], e.g., moving or replicating a data partition from an overloaded server to another, while the system is running transactions. Furthermore, live data migration must be efficient, with low impact on performance and minimal service interruption.

Scaling ACID transactions is hard, and providing elastic scalability even harder. Traditional techniques from distributed database systems [3], such as multiversioning and snapshot isolation, provide a good basis to increase concurrency between read and write transactions, but do not scale. Furthermore, the 2PC

☆ Part of the system and method described in this paper is protected by a patent application of Jimenez-Peris and Ballesteros (2022) [1].
* Corresponding author at: Universidad Politécnica de Madrid, Spain.
E-mail addresses: rjimenez@leanxcale.com (R. Jimenez-Peris), diego.burgos@leanxcale.com, diego.burgos.sancho@alumnos.upm.es (D. Burgos-Sancho), nemo@lsub.org (F. Ballesteros), mpatino@fi.upm.es (M. Patiño-Martinez), patrick.valduriez@inria.fr (P. Valduriez).

https://doi.org/10.1016/j.is.2022.102043

protocol that is used to coordinate distributed transactions is blocking in case of failures, and also a bottleneck since it makes commitment longer due to the two phases.

Elasticity requires the ability to move data partitions across servers with full ACID consistency while they are being updated. Guaranteeing full transaction consistency without creating a bottleneck in a scalable solution is hard. Traditional solutions rely on offline reconfiguration where the data partitions being reconfigured are not allowed to be updated while they are transferred across servers. However, this approach results in low availability of the data being moved.

One solution that is often used is relaxing some of the ACID properties, at the expense of more complex application programming. For instance, to avoid 2PC, microservice frameworks resort to sagas [5] which relax the atomicity property and require the user to provide compensating transactions. On the other hand, NoSQL systems, such as BigTable, Cassandra, and HBase, trade atomicity and consistency for scalability, with limited atomicity at the level of a single row. This makes it relatively easy to provide scaling without having to enforce ACID properties. Elasticity can also be handled in a simple way since row level atomicity is trivial, while ACID properties become much harder since the consistency of all rows modified by a transaction, very likely across different servers, has to be maintained. In the application examples above such as e-advertising, relaxing some ACID properties would simply make the application incorrect, e.g., for instance, missed CPC bids. Scaling ACID transactions has been addressed recently (see Section 2). Some solutions make strong assumptions regarding the database, e.g., which has to fit in main memory, or the availability of some special purpose hardware, such as RDMA or specialized clocks. Another solution is within the context of NewSQL, combining the scalability and availability of NoSQL with the consistency and usability of SQL [6]. Elasticity of OLTP is typically achieved outside the database system using a framework, for instance, E-store [7].

In this paper, we present our solution for elastic scalable transaction processing in LeanXcale, an industrial-strength NewSQL database system. Our solution does not make any specific assumptions regarding the database size nor does it require any hardware assistance. Our approach is based on several principles, which we divide in three groups: scalable, efficient and elastic transaction management. Scalable transaction management avoids the single-node bottleneck with a novel decomposition of the ACID properties. Scalability is made efficient in a large-scale cluster by reducing communication and CPU costs. Elastic transaction management deals with non-intrusive elasticity, that is, without hurting the quality of service (QoS) of transaction management. We show the correctness of LeanXcale transaction management.

LeanXcale has a relational key–value store, KiVi, which supports non-intrusive elasticity and can move data partitions without affecting transaction processing. This is based on an algorithm that guarantees snapshot isolation across data partitions being moved without actually affecting the processing over them. KiVi is also crucial for the efficiency of the solution since it is a relational key–value data store that has implemented all operators from relational algebra, but Join, and thus it enables to push down all operators below joins to the storage engine highly reducing the movement of tuples between storage engine and query engine. To validate our solution, we provide a thorough performance evaluation using the LeanXcale DBMS product with several benchmarks and micro-benchmarks on Amazon Web Services (AWS).

The paper is organized as follows. Section 2 discusses related work. Sections 3–5 describe each a group of principles: scalability (Section 3), efficiency (Section 4) and elasticity (Section 5). Section 6 shows the correctness of our solution. Section 7 presents the performance evaluation. Section 8 concludes.

2. Related work

There has been much work on providing scalability and elasticity for single node database systems in the context of multi-tenant deployments in the cloud [4]. In contrast, we focus on scalability and elasticity for single large-scale OLTP applications that are deployed onto multiple nodes in a distributed database system.

Scaling ACID transactions in cloud distributed database systems has been addressed recently, with four main approaches: main memory, cluster replication, hardware assistance, and NewSQL. Main memory database systems, such as SAP HANA [8] and VoltDB [9], keep all data in memory, which enables processing transactions and queries at the speed of memory without being limited by I/O bandwidth and latency. They can scale out in a shared-nothing cluster, but with some issues such as weakening isolation, losing the capability of queries and updates across data partitions or introducing the overhead of 2PC. But the main limitation is that the entire database and all intermediate results must fit in memory, which makes it unpredictable when data, transactions and queries evolve over time. Furthermore, the cost of a main memory database is extremely high compared to one that keeps data on persistent storage and a fraction of data in a cache. For many applications, e.g., e-advertising or IT monitoring, such cost is just prohibitive and not economically viable.

Cluster replication [10,11] is used in open-source SQL database systems such as MySQL or MariaDB. Full replication, i.e., having a full copy of the database on each node, is used to distribute the read workload across nodes. The most successful approaches are based on 1-copy snapshot isolation [12], the equivalent of 1-copy serializability [13], but for snapshot isolation. This approach yields logarithmic scale out [14], i.e., exponential cost to scale out, which makes it useful for read-intensive workloads, but at the expense of full replication. LeanXcale outperforms cluster replication by relying on a distributed transaction processing approach that scales out linearly to a large number of nodes.

Hardware assistance can be exploited to remove the inherent bottleneck of transactions, which is caused by the serial processing of commits combined with the latency and CPU overhead of processing transaction messages. Recently, several research prototypes have been proposed to scale transactions using the next generation of RDMA-enabled networking technology, e.g., FaRM [15] and FaRMv2 [16], FaSST [17] and NAM-DB [18]. Using RDMA, data can be moved from one server to another directly, bypassing the CPU. FaRM [15] provides scalability, availability, and serializability for committed transactions but not for aborted transactions. FaRMv2 [16] extends FaRM to provide snapshot isolation to all transactions with opacity (strict serializability for both committed and aborted transactions). This is achieved using a novel timestamp-ordering protocol that leverages the low latency of RDMA to synchronize clocks. FaSST [17] also provides scalability and serializability using an efficient RPC abstraction on top of unreliable datagram message-passing. However, it does not provide snapshot isolation, which provides better performance for read-heavy workloads and is more common in practice than serializability. NAM-DB [18] provides scalability and snapshot isolation, not availability.

In contrast, LeanXcale does not require any hardware assistance. However, it could well exploit RDMA-enabled networking technology, to bypass the CPU in processing transaction messages. This would make our solution even better. RDMA-enabled solutions are not yet widely available in the cloud due to limitations of the number of servers that can be connected through RDMA. However, when it becomes widely available in the cloud, LeanXcale will be enriched to exploit it, which will make our innovations even more efficient.

NewSQL systems typically use a NoSQL key–value store layer under a transaction management layer, in order to provide horizontal scaling. Spanner [19] is an advanced NewSQL database system with full SQL and scalable ACID transactions. It was initially developed by Google to support the e-advertising AdWords application. It uses traditional locking and 2PC and provides serializability as isolation level. To avoid the high contention between large queries and update transactions resulting from locking, Spanner also implements multiversioning, which removes read/write conflicts. However, multiversioning does not provide scalability. In order to avoid the bottleneck of centralized certification, updated data items are assigned timestamps (using real time) upon commit. For this purpose, Spanner implements an internal service called TrueTime that provides the current time and its current accuracy. To make TrueTime reliable and accurate, it uses both atomic clocks and GPS since they have different failures modes that can compensate each other. To avoid deadlocks, Spanner uses a wound-and-wait approach, thereby eliminating the bottleneck of deadlock detection. Storage management in Spanner is made scalable by leveraging BigTable, a wide column data store.

LeanXcale also implements multiversioning and avoids the bottleneck of centralized certification by assigning timestamps to updated data items upon commit. However, unlike Spanner, which uses real-time timestamps, LeanXcale uses logical timestamps, which is simpler and cheaper. LeanXcale's architecture is also different than most other NewSQL database systems that rely on a NoSQL key–value store, e.g., Spanner is built atop BigTable. LeanXcale has a relational key–value store, KiVi, that enables efficient SQL processing since all algebraic operators below a join in a query are pushed down to the data store, avoiding much data movement with the query engine.

Elasticity is typically achieved outside the database system using a framework. For instance, E-store [7] is an elastic partitioning framework designed for distributed database systems. It automatically scales resources in response to changes in an application's workload and enables the database system to move data across nodes. E-store has been integrated with H-store [20], a distributed main-memory database system (which has evolved into VoltDB). In contrast, LeanXcale's elasticity is fully integrated within the database system, leveraging the ability to move data partitions across servers without disrupting data processing.

3. Scalable transaction management

In this section, we first introduce LeanXcale's architecture with transaction management in mind. Then, we present our three principles for scalable transaction management, avoiding the single-node bottleneck of traditional transaction management. The first principle (decoupling the ACID properties) enables scaling out each property independently in a composable manner. The second principle (decoupling update visibility and atomic commit) removes the bottleneck of sequential commit processing, thus enabling the parallel processing of very high numbers of commits. The third principle (waiting for updates to be visible for session consistency) provides session consistency without introducing a bottleneck.

3.1. LeanXcale Architecture

The LeanXcale distributed database system has three layers:

- **KiVi storage engine**. KiVi is a distributed relational key–value store, combining the best of key–value stores (data partitioning, horizontal scaling) and relational data stores (relational schemas and algebraic operators such as predicate filtering, aggregation, grouping, and sorting).
- **Transaction manager**. It is a distributed transaction manager able to scale out linearly to a large number of nodes. And the main focus of this paper.
- **SQL Query engine**. The query engine is also distributed and can scale out both OLTP and OLAP workloads.

Using a key–value store as storage engine has been used in several NewSQL database systems such as Spanner (which uses BigTable), or Splice Machine [21] and EsgynDB [22] that both use HBase [23]. This approach enables to scale out the storage layer to very large levels by using horizontal partitioning across storage instances. However, it comes at the expense of inefficient SQL query processing, because the interaction between the query engine and storage engine becomes inherently distributed, thus resulting in high communication overhead. For instance, consider an aggregate operation over the column, e.g., sum, of a large table of 1 billion rows. The execution of the aggregation would require scanning all the servers in parallel, and then sending the 1 billion scanned rows over the network to the query engine that will aggregate them and produce the single row result. Such execution results in high communication.

KiVi enables to push down all algebraic operators in a query plan below joins to multiple KiVi instances. The algebraic operators are executed locally at each KiVi instance, thus avoiding moving all scanned rows to the query engine for processing, and only sending the relevant rows. In the previous example, instead of moving 1 billion rows, a query execution would just move a single value per server, the local sum, i.e., 100 values in total. Since transactions may include queries to read data, KiVi's ability to perform algebraic operators yields major reduction of query latency and thus better transaction response time.

In addition, KiVi uses a variant of LSM Tree [24] that is more efficient in read operations than the SSTables (String Sorted Tables) used by key–value stores, which have to read many (tens of) files that contain an overlapping range of primary keys.

The query engine is distributed. It implements both inter-query parallelism and intra-query parallelism [3], in particular, intra-operator parallelism, and can have an arbitrary number of instances. To scale out OLTP workloads, each query engine instance takes care of processing a subset of the queries.

The transaction manager is also distributed and has a set of different components (see Section 3.2). It is highly scalable, able to process many millions of update transactions per second. The transaction manager is integrated within the client KiVi layer, thus providing LeanXcale the capabilities of an ACID relational key–value store.

3.2. Decoupling the ACID properties

In this section, we describe our approach for decoupling the ACID properties to scale out. Let us first recall the ACID properties [25]:

- **Atomicity** provides an all-or-nothing behavior for all the updates of a transaction.
- **Consistency** requires correct code for the transactions that guarantees that if the data is consistent at the start of the transaction, it should remain consistent at the end of the transaction, i.e., satisfies the database integrity constraints.
- **Isolation** provides synchronization atomicity, that is, guaranteeing that the execution of transactions in parallel provides certain consistency guarantees over the data (e.g., serializability or snapshot isolation).
- **Durability** guarantees that the updates of a transaction, once committed, are not lost in the advent of failures.

LeanXcale leverages **multiversion concurrency control (MVCC)** [25] to avoid the contention introduced by locking between queries that read many rows and updates across these rows. In multiversioning, data is not updated in place, but instead, new versions of the rows are created with the new updates (see Algorithm 1, lines 5–7). In MVCC, a **commit timestamp (CTS)** is used to label each version.

Algorithm 1 KiVi Storage Engine

```
1: $BD \leftarrow 0$
2: function READ(table, PK, STS) → Tuple
3:     $tuple \leftarrow t \in BD | (table = t.table \wedge PK = t.pk \wedge t.cts \leq STS) \wedge \nexists t2 \in BD | (table = t2.table \wedge PK = t2.pk \wedge t2.cts \leq STS \wedge t2.cts > t.cts)$
4:     Return $tuple$
5: procedure WRITE(Tuples, CTS)
6:     for $t = (table, PK, Cols) \in Tuples$ do
7:         $BD \leftarrow BD + (table, PK, Cols, CTS)$
```

LeanXcale provides snapshot isolation as its isolation level. Snapshot isolation provides a slightly lower isolation than serializability. In particular, it is possible to have an anomaly called write skew. It involves a constraint across two rows (or more) and writing different rows involved in the constraint by the two transactions. Let us illustrate with an example. Consider a bank that allows a person having two bank accounts, say b_1 and b_2, to have a negative balance in any of the two accounts as far as the global balance remains positive. Two concurrent withdrawals, each from a different account, can lead to a negative global imbalance. A withdrawal operation W of w units from b_1 would do: 1a) bb_1 = R(b_1), 2a) bb_2 = R(b_2) 3a) if bb_1 + bb_2 - $w \geq 0$ then W(b_1, bb_1 - w) while the withdrawal operation from b_2 would do: 2a) bb_1 = R(b_1), 2a) bb_2 = R(b_2) 3a) if bb_1 + bb_2 - $w \geq 0$ then W(b_2, bb_2 - w). If the original balances are bb_1=10, bb_2=15 and the two withdrawals are w_1=20 and w_2=25, an interleaving 1a, 2a, 1b, 2b, 3a, 3b would produce bb_1=-10, bb_2=-10, thus a negative balance of -20 units.

Snapshot isolation is the highest isolation offered by some leading database vendors. There are a number of techniques in the literature to attain serializability on top of snapshot isolation, e.g., [26,27] . In order to avoid write skew, LeanXcale provides select for update, which could be used too by the standard snapshot isolation.

Atomicity is attained in MVCC by making private the new versions of the data generated by the updates of the transaction, i.e., invisible to concurrent transactions. Thus, atomicity is taken care of by the component that orchestrates the transaction life cycle. In LeanXcale, this component is the **local transaction manager (LTM)** that is deployed on each node with transactional capabilities. LTMs are also in charge of orchestrating the commits of the transactions.

Consistency does not require specific techniques as it is the responsibility of the application developer (to provide correct code), with the support of traditional integrity control.

Isolation is also enforced through MVCC, with snapshot isolation provided as isolation level. We further decompose isolation into isolation of writes and isolation of reads. In snapshot isolation, reads never conflict. Thanks to multiversioning, they just read the version corresponding to the snapshot associated to the transaction (see Algorithm 1, lines 2–4), i.e., the oldest version of the row with a CTS lower or equal than the **start timestamp** of the transaction also called **snapshot of the transaction**. Thus, the only task to be performed is to detect write–write conflicts, which is the role of **conflict managers**. However, reads need be performed on a consistent snapshot of the data that guarantees snapshot isolation. This is achieved by two components, **commit sequencer** and **snapshot server**, which are described in detail in the next section.

MVCC traditionally requires certification at the beginning of the commit processing. In LeanXcale, certification is performed at the time a row is updated and if there is a conflict, one of the transactions is aborted using a first updater wins approach. Thus, only transactions that do not have conflicts (i.e., have already passed the certification) enter the commit stage. Furthermore, a transaction enters the commit stage with an assigned CTS that sets its serialization order. With this approach, certification is fully distributed and does not require a centralized certifier, which is one of the main scalability bottlenecks of MVCC-based solutions.

Durability requires a mechanism to persist changes of a transaction independently of the changes performed in the data. LeanXcale adopts the no undo/redo approach [25]. The component that takes cares of durability is the **logger**. Writesets are kept in memory to implement this policy. However, if the writeset size becomes too large, it is persisted on disk.

3.3. Decoupling update visibility and atomic commit

In this section, we describe our principle of decoupling update visibility and atomic commit to enable scaling out. Processing commits sequentially is the main bottleneck of traditional database systems [13]. The reason that the traditional solution performs commit processing tasks sequentially is to ensure that newly started transactions observe a correct snapshot.

We adopt a radically different approach, by untangling commit processing and leveraging multiversioning and snapshot isolation. In particular, we maintain two counters: a snapshot counter, initialized to zero and a commit counter, initialized to one. The commit counter is used as usual to assign increasing CTSs to committing transactions. At start time, the transaction receives the current value of the snapshot counter as the start timestamp.

Let us describe in more details our solution to commit processing. The commit of a write transaction, say T_i, proceeds in four sequential steps: (1) T_i receives its CTS from the **commit counter** that **is incremented**; (2) T_i's writeset is written into the redo log that is flushed to stable storage. Then, the transaction is **durable** and the client is acknowledged that commit was successful; (3) the new versions of the data are written to the data store. Once the data is in the data store, the transaction is **readable** and the transaction manager is informed; (4) the transaction manager **updates the snapshot counter, when there are no gaps** in the serialization order until this CTS, so at this point, the transaction is **visible**. Gap-freedom is essential to consistency since the gap-free prefix grows monotonically in the order of the CTSs and each snapshot observes only a fraction of this key–value data store, more precisely a longer prefix with a bigger value of the snapshot.

Only step 1 is made atomic and quite fast: an integer increment. Logging can be performed later. Delaying logging only impacts the latency of the transaction commit. As we will see later, it enables parallelizing the logging process (step 2) in order to attain high logging throughput. The transaction response time only includes these two steps (1 and 2), not the time needed to store the updated data. Again, this is interesting to mask the latency of a distributed data store and open up opportunities for more efficient update propagation. When all the data modified by a transaction is updated in the data store (step 3), the transaction enters the readable state. Thus, the updated data could be read from the data store if the right snapshot is used. However, in order to guarantee consistency, these data are still **invisible** to new transactions. The visibility state is reached in step 4, when the snapshot counter is updated and becomes equal or greater than the CTS.

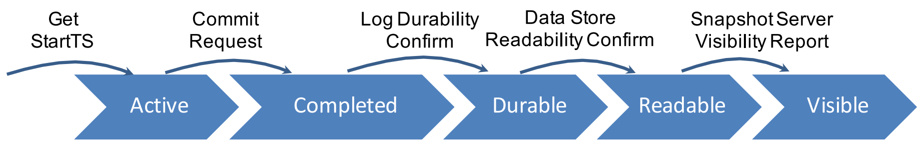

Fig. 1. Transaction Phases.

Table 1
Untangled Transaction Commit.

CTS notifications	11	15	12	14	13
Evolution of snapshot counter	11	11	12	12	15

Given that the commit is not executed atomically, it is possible that T_i receives a CTS smaller than T_j while T_j stores its updates in the data store before T_i. Had updates become visible immediately, transactions would potentially observe an inconsistent snapshot containing T_j updates but missing T_i updates. We take this into account and only advance the snapshot counter to the value X when the updates of all transactions with a CTS smaller or equal to X are readable, i.e., when there are no gaps in the serialization order. Thus, only when the snapshot counter becomes equal or greater than T_j's CTS that the updates of T_j will become visible. Thus, they can be observed by newly started transactions with a start timestamp equal or greater than T_j' CTS. In other words, a transaction T_i is fully committed when the snapshot takes a value equal or higher than the CTS of T_i. Until that point, its changes are invisible, and from then on, they become visible to transactions that get a start timestamp equal or higher than its CTS.

The snapshot counter represents the longest gap-free prefix of committed transactions. It is not incremented by one each time a transaction completes its commit since it would lead to the visibility of a prefix with gaps and thus, inconsistent reads. When the snapshot counter is equal to the CTS $C(T_j)$ of transaction T_j means that data versions created by T_j and all transactions with CTS less than $C(T_j)$ are durable as well as readable from the data store (i.e., stored in the data store layer, but not necessarily persisted on disk).

Fig. 1 shows the transaction phases during execution, with the untangling of the commit phase. Upon receiving its start timestamp, a transaction becomes active. Once it has received its CTS, it has completed. After writing the updates to the log, it is **durable**. Once the updates have been propagated to the data store, they become **readable**. Finally, once the snapshot counter has advanced to include the CTS of the transaction, its updates become **visible**.

Let us illustrate the evolution of commit and snapshot counters with the example in Table 1. Let us assume that the commit and snapshot counters have value 10. Then, five transactions (assigned with CTSs 11 to 15) start to commit in parallel. The first row in Table 1 gives the order in which the notifications that the transaction updates have become "durable and readable" are received.

But since commits are done in parallel, they can happen in any arbitrary order. The second row shows the evolution of the snapshot counter with each "durable and readable" notification. Upon receiving the notification of CTS=11, the snapshot counter can be incremented safely from its current value, 10, to 11, because there is no gap. Then, the snapshot counter receives the notification for CTS=15. But since there are gaps between 11 and 15, it cannot be incremented. Otherwise, it could lead to inconsistent reads. For instance, assume the snapshot counter is incremented to 15. Then, a transaction could observe the updates of transaction with CTS=15 but not those of transactions with CTS=12 to 14. However, it could observe later on the updates of transactions with CTS=12 to 14 once they become "durable and readable", thus leading to inconsistent reads.

When notification for CTS=12 is received, the snapshot counter is advanced to 12. However, with the notification for CTS=14, the snapshot counter cannot be advanced since 13 is still missing. Finally, when the notification for CTS=13 is received, the snapshot counter can be advanced to 15 since there are no gaps until that value.

In summary, our solution processes each commit as a pipeline of tasks, thus enabling all commits to proceed in parallel. Thus, commit processing can be scaled out to very large levels. Snapshot isolation is preserved by regulating the visibility of committed updates through the snapshot counter that allows observing only gap-free prefixes of committed transactions.

3.4. Waiting for updates to be visible for session consistency

Our solution returns the commit to the client when durability is guaranteed, but before the updates of the transaction become readable and visible. This may violate session consistency, also known as "read your own writes" [28], i.e., a client might not read its own writes across different transactions. Let us consider two consecutive transactions from one client, T_1 and T_2. T_1 updates x and commits. T_2 starts before T_1's update is visible, and thus, receives a start timestamp smaller as T_1's CTS. Therefore, when T_2 reads x, it will not receive the version created by T_1.

Session consistency can be implemented by delaying the start of new transactions after the commit of an update transaction until the snapshot counter reflects the CTS of that committed update transaction. Only then, the LTM can assign the start timestamp to a new transaction in that session. This delay can be a few tens of milliseconds, which is negligible for OLTP response time, which is in the subsecond range. This delay is only incurred by a client when it starts a transaction immediately after committing an update transaction. After committing a read-only transaction (typically around 90% of the transactions in a typical OLTP workload) there is no delay. Furthermore, if a client application performs some activity between the commit of an update transaction and the start of a new transaction, this delay can be masked. Note that, although we talk in terms of delay with respect to the time a commit becomes durable, all activities are done much earlier than in traditional database systems that simply commit transactions in sequence, waiting until the updates are visible before starting the next transaction in the current session and any other session.

4. Efficient transaction management

In this section, we present three additional principles to make transaction management efficient (and scalable), complementing the three principles in the previous section. The two first principles address the limitations of single-node bottlenecks introduced by monolithic transaction processing. The first principle (distribution and parallelization) applies to the components devoted to each ACID property, yielding linear scale out. The second principle (proactive timestamp management) substantially reduces the amount of communication and transaction latency due to communication. The third principle (asynchronous messaging and batching) helps reducing even more the overhead of distribution.

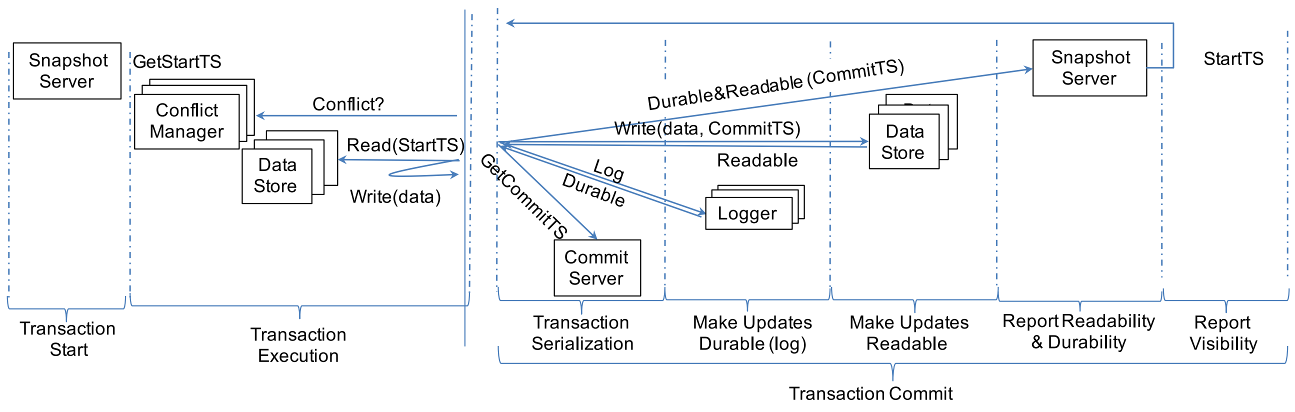

Fig. 2. Lifecycle of a Transaction.

4.1. Distribution and parallelization

In Section 3.2, we decomposed the ACID properties and scaled out them independently. The principle of distribution is to assign the relatively independent tasks executed by the transaction manager to the different independent components identified in the previous section: LTMs, conflict managers, loggers, snapshot server, and commit sequencer. For some of these components, the tasks can be parallelized into many component instances that can run on different nodes, thus, increasing scalability even further. Fig. 2 summarizes the entire transaction life cycle and the interaction between components.

Algorithm 2 Local Transaction Manager (LTM)

```
1: procedure BEGIN($clientId$)
2:     $tid[clientId] \leftarrow getNextTID()$
3:     $writeset[clientId] \leftarrow \emptyset$
4:     $sts[clientId] \leftarrow snapshotServer.getSnapshot()$
5: function READ($table, pk, clientId$) → Tuple
6:     if $pk \in writeset[clientId]$ then
7:         $tuple \leftarrow dataManager.ReadFromWriteset(writeset[clientId], table, pk)$
8:     else
9:         $tuple \leftarrow dataManager.Read(table, pk, sts)$
10:    Return $tuple$
11: procedure WRITE($table, pk, cols, clientId$)
12:    if $\neg conflictManager.Conflicts(table, pk, sts[clientId])$ then
13:        $writeset[clientId] \leftarrow writeset[clientId] + (table, pk, cols)$
14:    else
15:        $localTransactionManager.Abort(clientId)$
16: procedure COMMIT($clientId$)
17:    $cts[clientId] \leftarrow commitSequencer.GetCTS()$
18:    $conflictManager.Commit(tid[clientId], cts)$
19:    $logger.Log(tid[clientId], writeset[clientId], cts[clientId])$
20:    $storageServer.Write(tid[clientId], writeset[clientId], cts[clientId])$
21:    $snapshotServer.DurableAndReadable(cts[clientId])$
22: procedure ABORT($clientId, tid$)
23:    $snapshotServer.Abort(tid[clientId])$
24:    $conflictManager.Abort(tid)$
```

The transaction lifecycle is managed by the LTMs (see the LTM pseudocode in Algorithm 2). LTMs are in charge of getting the start timestamp of a transaction (Begin), using the right snapshot for reads (Read) and updating the private versions (Write), and processing the different steps of the commit (Commit) that involves logging its updates, making them readable and finally reporting to the snapshot server about the durability and readability of the transaction and providing its CTS. Since such lifecycle management can be performed independently for different transactions, they can be parallelized as well by having several independent LTMs, each collocated with the query engine (see Fig. 2).

Conflict management is provided by an independent component, the conflict manager, which is parallelized across many instances. Its function is the same as conflict detection in any transaction processing system providing snapshot isolation, detecting write–write conflicts among concurrent transactions. However, unlike traditional systems that detect conflicts during validation, the conflicts are detected before reaching the commit stage.

Each conflict manager instance running on a different node (see Algorithm 3) is responsible for a subset of data record keys, i.e., a bucket. Record keys (containing a unique table identifier and key) are hashed and assigned to a bucket using the function: $bucket$ = hash($tableID, key$) mod $numberOfBuckets$. The bucket is the unit of distribution for the conflict manager and each conflict manager is in charge of a number of buckets. Each bucket is handled by a single conflict manager. The conflict manager keeps at most two values per data item: the CTS of the last committed version and the start timestamp of an active transaction updating the data item, if any. Along with the distribution of conflict managers, this avoids the conflict manager being a bottleneck when it handles the whole set of keys, which can be huge [20].

Algorithm 3 Conflict Manager

```
1: $conflicts \leftarrow \{\}$
2: function CONFLICTS($pk, tid, sts$) → Boolean
3:     if $ConflictsWithAnyConcurrentTxn(pk, sts)$ then
4:         Return $True$
5:     else
6:         $Store(conflicts, pk, tid, sts)$
7:         Return $False$
8: procedure COMMIT($tid, cts$)
9:     $SetUpdatesToCommitted(tid, cts)$
10: procedure ABORT($tid$)
11:    $DiscardUpdates(tid)$
```

Before a transaction T_i can execute an update on a data record, a conflict request is sent to the conflict manager that is responsible for this record. A conflict is detected if the conflict manager has previously accepted a request from a concurrent transaction (either active or committed). For each transaction, its LTM keeps the information about how many conflict managers have been involved. Upon successful completion of a transaction commit, all the involved LTMs are informed about its CTS so that each conflict manager updates the information about the conflicts and can perform the proper checks for future transactions.

Fig. 3 shows the interaction and the information maintained by the conflict manager that is responsible for rows x and y. For each row, it keeps track of the last committed transaction T_i that has updated the row, and possibly an active transaction T_j that has updated the row since then. When a new transaction T_k checks for a conflict on x, it checks whether an active transaction T_j exists that has updated x and if not, if $C(T_i)$ of the last committed transaction that updated x is greater than $S(T_k)$. If any of the two is true, there is a conflict. Otherwise, there is no conflict and T_k is added as an active transaction for x.

In Fig. 3, transaction T_1 checks for a conflict on x, T_2 for a conflict on y, and T_3 for a conflict on z. There are no conflicts. T_3 checks for a conflict on y and T_2 is active and changes y. Therefore, T_3 has a conflict. Then, T_2 confirms the commit and information about T_2 can be discarded. When later on, a transaction T_4 checks for a conflict on y, since $S(T_3) \geq C(T_2)$, there is no conflict, and T_3 is added to x's entry. This approach scales out conflict management.

Durability provided by the loggers can also be handled independently. The redo records of a transaction are pushed to a logger and made durable before the commit acknowledgment is returned to the user. The logger is distributed and parallelized by creating as many logger instances as needed to handle the required throughput. Each logger takes care of a fraction of log records (see Algorithm 4). Loggers are totally independent and do not require any coordination. There is no requirement to log records in any particular order, thus, it becomes possible to log in parallel with no coordination. Log records are inserted into the logger's buffer. The buffer content is flushed at the maximum rate allowed by the underlying storage, thus minimizing the latency of logging. Note that coordination is not needed across loggers because log records are labeled with the CTS and when redoing upon recovery, a log record is not replayed if the current row in the database has a CTS higher than the recovery log record. Thus, idempotence of recovery is guaranteed.

Algorithm 4 Logger

```
1: procedure Log(tid, writeset, cts)
2:     Persist(tid, writeset, cts)
```

4.2. Proactive timestamp management

Proactive timestamp management addresses a potential bottleneck. We separate the beginning of a transaction from its commit by providing two independent counters: the commit and snapshot counters. We provide two independent services: a commit sequencer that maintains the commit counter, and a snapshot server that maintains the snapshot counter. Note that the work performed by these components is tiny per transaction. Thus, the main cost will become the communication with other components to request start timestamps and notify readability of transactions. The requests for CTSs are served by the local commit sequencer while the requests for start timestamps are sent to the snapshot server. The snapshot server is contacted by the LTM once the updated data of a committed transaction is applied at the data store. Thus, the snapshot server keeps track of the most recent snapshot (i.e., update visibility) for which all updates are readable in the data store layer. Timestamp management becomes the ultimate bottleneck of this approach. Therefore, it is critical to perform this task in a lean way that minimizes its cost. So far in this paper, requests for start and commit timestamps are sent in a reactive way, i.e., only when they are actually needed.

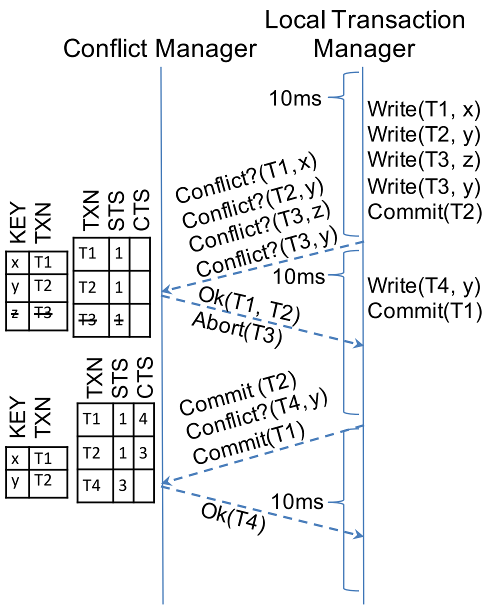

Fig. 3. Example of Conflict Management.

We can avoid this by adopting a proactive approach to timestamp management and making proactive the commit sequencer, snapshot server, and conflict manager.

4.2.1. Proactive commit sequencer

Once a transaction enters the commit phase, it is guaranteed that it does not conflict with any other transaction (it has already been validated). The commit timestamp is only needed to tag the updates with that CTS. Therefore, if two transactions request a CTS, it does not matter which timestamp gets each transaction. So, using Algorithm 5, the commit sequencer can send proactively (before they are requested) a range of CTSs to each LTM. The LTMs can then assign CTSs from their local range to commit transactions, which avoids the synchronization with the commit server on a per transaction basis. Thus, there is no latency in assigning a CTS, with only one message per conflict manager every period.

Algorithm 5 Commit Sequencer

```
1: $nextCTS \leftarrow 0$
2: function GetCTS → Integer
3:     Return $nextCTS++$
```

Fig. 4 depicts this proactive approach. First, the commit sequencer proactively provides the query engine nodes with CTSs. It sends a range of CTSs periodically. CTSs are assigned to transactions from this range locally at the LTM without any communication with the commit sequencer. In the figure, the commit server first sends a range of CTSs, B_1, to the LTM. When T_1 requests to commit, it receives the first timestamp of the latest range, i.e., 0. Then, when T_2 asks for a CTS, it receives timestamp 1. Upon the arrival of a new range of CTSs from the commit server, B_2 the range of unused timestamps will be simply discarded (2 in the example). Therefore, when T_3 requests the commit, it receives timestamp 3.

The reason for switching to the new range instead of using the old one is to advance CTSs in sync with timestamps globally in the system, and therefore does not delay the freshness of

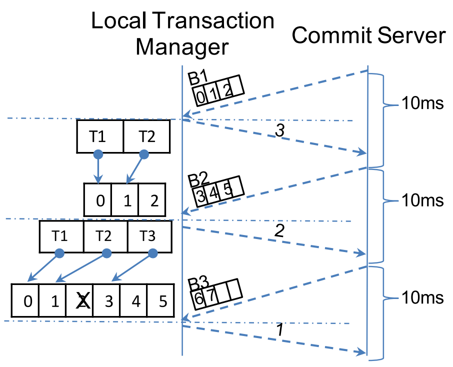

Fig. 4. Proactive Commit Sequencer.

Algorithm 6 Snapshot Server

```
1:  $snapshot \leftarrow 0$
2:  $nonConsecutiveCTSs \leftarrow \emptyset$
3:  procedure DurableAndReadable($cts$)
4:      if $cts = snapshot + 1$ then
5:          $snapshot \leftarrow snapshot + 1$
6:          while $snapshot + 1$ in $nonConsecutiveCTS$ do
7:              $snapshot ++$
8:              $nonConsecutiveCTS \leftarrow nonConsecutiveCTS - \{snapshot\}$
9:      else
10:         $nonConsecutiveCTSs \leftarrow nonConsecutiveCTSs + \{cts\}$
11: function getSnapshot → Integer
12:     Return $snapshot$
```

the current snapshot. If one LTM processes transactions faster, it will receive larger timestamps while a slow LTM still uses low-value timestamps. This will delay the advancement of the global snapshot counter. As a result of discarding CTSs, the snapshot server is not only informed about the timestamps of committed transactions but also about unused CTS ranges, so that it can advance the snapshot counter every period independently of the relative speed of LTMs.

In the proactive approach, a period is defined for the interaction with the commit sequencer and snapshot server. Every period, the commit sequencer provides a new range of CTSs to the LTMs. Initially the range size is the same for all nodes. After each period (e.g., 10 ms), the commit sequencer is informed about the number of transactions that committed in the past period at each LTM. This commit sequencer uses this information as an estimate for the range size for the next period for each node.

4.2.2. Proactive snapshot server

In the reactive approach, each time a new transaction is started, a message is sent to the snapshot server to request the current snapshot and a message with the current snapshot is sent to the requesting LTM. This means a cost of two messages per started transaction and the latency of two messages. In the proactive approach, the snapshot server proactively reports the current snapshot counter to the LTMs for every period (see Algorithm 6). This counter is used by newly started transactions, thus avoiding message exchange on a per-transaction basis. Therefore, a single message is sent every period to each LTM.

Recall that the snapshot server must also be informed about CTSs whenever updates have been applied at the data store, indicating that they have become readable. Instead of sending these readability confirmations per single transaction, we report a full range of CTSs, within the same range used by the commit sequencer. Note that the transaction latency is not affected by this batching, since the user application is notified as soon as the transaction has become durable.

4.2.3. Proactive conflict manager

Checking conflicts on a per updated row basis has a high cost since two messages are exchanged. For this reason, in the proactive mode, conflict checks are batched and sent every period. Thus, the conflict management communication cost is amortized across as many rows as possible. Fig. 3 shows how batching is applied to the example. Thus, transactions do not wait for messages to be exchanged to receive a start and commit timestamp. Furthermore, commit timestamp ranges and start timestamps are distributed proactively in a periodic manner to avoid two synchronous message exchanges on the transaction response time. The snapshot server also interacts periodically with query engines to collect readability notifications and report the latest snapshot timestamp. Thus, the only way to saturate these components will be with a very large-scale system that would send too many messages per period. This issue can be easily overcome using a multicast tree of nodes to distribute this information at very large scale, e.g., as in content distribution networks, yielding seamless scalability to our timestamp management.

4.3. Asynchronous messaging and batching

Each transaction needs several messages for transaction management, to perform conflict detection, reading and writing data, and logging. As these message exchanges are synchronous, i.e., the transaction waits for them to be completed, they become part of the critical path of the transaction response time. Although these tasks are performed synchronously, we have adopted an asynchronous approach whenever possible to remove message latency from the transaction response time. Another issue related to distribution and the inherent message exchange is the cost of messaging. For a large-scale system, it is important to minimize this cost as much as possible to reduce the distribution overhead.

We have already seen how timestamp management latency can be fully removed from the transaction response time by our proactive timestamp management. Conflict management latency can also be mostly removed from the transaction response time. We do this by checking for conflicts asynchronously. Additionally, to reduce the distribution overhead, we apply batching of requests and responses extensively. In typical batching approaches, latency is traded for throughput, which increases response times. We apply batching combined with asynchrony to avoid a negative impact on response time.

In the naive solution, an update is only executed after the conflict manager checks for conflicts. This delays transaction execution considerably. Every update causes a round-trip message delay. Additionally, it incurs the CPU cost of a round trip message exchange at the sending and receiving sides. We avoid this by performing conflict management mostly in an asynchronous way and overlapping it with transaction execution to hide its latency. We send conflict detection requests asynchronously to the conflict managers and immediately continue with the update before receiving the accept/reject response. Only when the commit is initiated, the transaction might wait for all outstanding responses to conflict requests. Note that if a negative response arrives in the middle of transaction execution, we can immediately abort the transaction.

This asynchrony does not only allow keeping response times short but also enables to apply batching to conflict management without affecting response time. Conflict detection requests are buffered, with one buffer per conflict manager. Periodically (e.g., every 10 ms), the buffers are sent to the corresponding conflict managers. When a conflict manager receives a buffer, it performs all checks and returns all responses in a single message. This reduces considerably the total number of messages sent, without affecting the response time as the delay is hidden by the concurrent transaction execution. Only the waiting for the last batch can affect the overall response time of a transaction.

Reads are necessarily synchronous. However, in our approach, writes can be performed asynchronously and we take advantage of this possibility. Writes from different transactions can be batched to be sent to KiVi instances. By trading off some delay in the update propagation, it then becomes possible to batch all the updates propagated to a particular data store instance, thus increasing the overall efficiency.

5. Elastic transaction management

Efficient, scalable transaction management (as provided by our approach so far) is useful for static workloads for which data partitioning can be well designed. However, when the workload changes dynamically, performance can degrade dramatically. For instance, in a deployment with 100 servers, the load may suddenly concentrate on the data handled by one server, thus limiting the performance to that of a centralized system. In order to adapt to workload changes and be able to process greater (or lower) transaction rates, scalability must be elastic, but also non-intrusive, i.e., without hurting the QoS of transaction management.

In this section, we present three additional principles to attain non-intrusive elasticity of transaction management and the underlying components, namely, transaction manager (LTM, logger, conflict manager), query engine, and KiVi. The first principle (non-intrusive moving of data partitions) is the first building block. The second principle (dynamic load balancing of transactional components) is the second building block, leveraging on the previous one. The third principle (transactional elasticity) extends dynamic load balancing with the capability of adding or removing nodes with transactional component instances when the average cluster load gets either too high or too low.

5.1. Non-intrusive moving of data partitions

This principle is the ability to move data partitions to other servers[1] in a non-intrusive way, i.e., when being accessed by transactions while guaranteeing the ACID properties. This principle is supported by the capability of splitting transactional data partitions dynamically and moving them across servers also dynamically while fulfilling the ACID properties. Transactional data partitions are represented by means of a B+ tree that is updated as an LSM tree, in batches from a cache. A transactional data partition is thus the combination of a B+ tree and a cache. The split of the data partition involves partitioning both the B+ tree and the cache. The splitting process is as follows. First, the metadata indicates that the data partition is going to be split into two data partitions, with their key ranges. The key at which the split is performed is the middle key in the root of the B+ tree. Then, the B+ tree root node is split into an atomic process into two different root nodes, each with half of the children of the root. If the root node has a single key, the two children of the root become the two new trees. The cache is not split. Instead, when it is propagated to the B+ tree, it is propagated to the two new trees. When the cache starts to be propagated, two empty caches are created, one for each tree. In this way, the split process is totally online and does not stop or disrupt transaction processing.

Dynamic movement of data partitions is achieved by an algorithm that moves a data partition without stopping the ongoing transactions, not even those updating the data partition being moved. The algorithm starts by telling the metadata server that the data partition is going to be moved from the current origin KiVi server to the destination KiVi server. Then after, it notifies both the origin and destination KiVi servers about the data partition movement. The destination KiVi server can start buffering all the changes that might arrive for the data partition being moved. Then, all the instances of the client side of KiVi are notified about the data partition movement. Each client instance upon notification starts sending changes not only to the origin data partition, but also to the destination data partition. Once all client instances have been notified, then all changes are being sent to both origin and destination KiVi servers. Then, the origin KiVi server is asked to start sending the data partition to the destination KiVi server. Note that the KiVi client side configuration is not atomic, that is, all clients are configured concurrently and the final order in which they get configured can be any. Thus, the changes being sent to the destination KiVi server can be different from each client side. What is important is that all client sides are configured to send to the origin and destination servers, before starting to move the data partition, which guarantees that no updates can be missing.

The data partition being sent can be updated in the meantime. Thus, some changes might be incorporated in the sent data partition while some other changes will not. The process to send a copy of the data partition sends all the rows in the partition by traversing the corresponding leaves of the B+ tree. Changes to a row are atomic, i.e., a particular row is moved before or after a change, but not during the change itself. Since multiversioning is used, a change is always implemented as an insertion, so a row is sent either before it has been inserted or after.

At the destination KiVi server, rows are inserted using the regular processing using a cache and a B+ tree. At some point, all the rows on the origin data partition have been sent to the destination KiVi server. Then, the destination KiVi server is notified about this fact and starts applying the row changes that has been buffered and are still buffering.

An important aspect for data consistency is to guarantee the idempotence of all row changes. This is achieved by taking advantage of combining the primary key of the row with the commit timestamp. This makes a unique identifier for a particular row change. Thus, when applying the buffered row changes, if a row with the same primary key and commit timestamp is found, then the buffered row change is just discarded since the change was already applied. When a change is actually applied we say that it has been *effectively applied*. Also, since the buffering of changes started before starting to send a copy of the data partition, it is guaranteed that no change is missing. Any change after finalizing the reconfiguration of the clients is guaranteed to be already buffered at the destination server. In extreme loads, where the destination KiVi server never catches up, there is a temporal limit, upon which the processing over the partition would be paused to enable to complete the movement and finalize the reconfiguration. In practice, this mechanism is almost never needed.

5.2. Dynamic load balancing of transactional components

The dynamic load balancing of the transactional components (query engine, transaction manager, KiVi storage engine) depends on the nature of the state they manage: session state, temporal state, or persistent state. Session state is the state related to a

1 The method and system for online data partition movement described in this paper is protected by a patent application [1].

client session. Both the query engines and LTMs that are collocated have session state. They keep client sessions connected to them that start and commit transactions and submit SQL statements. The way to balance the load across query engines and LTMs is by moving sessions across their instances. We move sessions only when there is no transaction active in the session. Thus, moving sessions requires reconfiguring the client connection to send new transactions to a different query engine and LTM. The only state that is moved is the timestamp of the last committed update transaction needed for session consistency.

Conflict managers keep a temporal state. This state is related to the updates performed over the different keys, which has to be remembered for some time to detect write–write conflicts. The unit of distribution in conflict managers are buckets. We use hashing and then get the modulo by the number of buckets to assign a bucket to a key. Then, each conflict manager is responsible for a set of buckets. Since this information is just needed for some time, the way we deal with dynamic load balancing is by giving the responsibility of a bucket of a loaded conflict manager to a less loaded conflict manager. A naïve solution would be to send the bucket state from one conflict manager to the other and buffer the conflict requests on the receiving conflict manager. Instead, we exploit the temporality of conflict information and just send conflict requests to the old and new conflict manager for that bucket. The replies are sent only by the old conflict manager. At some point, all conflict information is in the two conflict managers. Then, LTMs are reconfigured to get the replies for the moved bucket from the new conflict manager and send only the conflict requests to the new conflict manager and remove the bucket information from the old conflict manager.

There are two components with persistent state: loggers and KiVi. However, although they are stateful, loggers can be handled almost as session state components because of two features. The first feature is that loggers are append only. The second feature is that logging information can be stored in any logger and only metadata information needs to be kept to relevant loggers in order to do recovery efficiently. To move the load from a highly loaded logger to a less loaded logger, we just need to reconfigure one LTM that sends logs to the highly loaded logger to send the logs to the less loaded logger. This information is also stored in the metadata to know that the less loaded logger is involved to recover transactions from the associated LTM from a particular commit timestamp.

The load balancing of KiVi leverages the principle of non-intrusive moving of transactional data partitions. When a KiVi instance becomes highly loaded, one of its data partitions is moved to a less loaded KiVi instance. Such moving is transparent to applications. Load balancing is a hard problem by itself. When a single resource is involved (e.g., CPU or IO bandwidth), there are greedy solutions that provide an optimal solution. This is the case for all transactional components but KiVi. A KiVi instance can be CPU, memory or IO bound. Unfortunately, multi-resource load balancing is an NP hard problem [29]. We have designed a greedy multi-resource load balancing that handles multiple resources with a solution very close to the optimal in affordable computation time [30].

5.3. Transactional elasticity

Transactional elasticity combines the capabilities of splitting transactional data partitions, moving them and deciding where to move them based on the dynamic load balancing algorithm. The algorithm proceeds as follows. Once a new server has been provisioned and registered in the metadata, then a number of data partitions can be moved to it. When a particular data partition becomes too big or overloaded, it is split and can be moved if needed to another KiVi server. Elasticity requires setting thresholds of the average server load on the different dimensions of the load, CPU, memory and IO bandwidth. There is an upper threshold for each resource (e.g., 80% CPU) that, when overcome, triggers the provisioning of a new server. The data partitions moved to the new server are decided by the dynamic load balancing algorithm. There is a lower threshold for each resource. When the average cluster resource utilization of all resources after removing one server is below the lower threshold, then the less loaded KiVi server is chosen and the dynamic load balancing algorithm decides to which other KiVi servers to move each partition. When the server does not have any data partition, then it is decommissioned. Note that there is a highly available version of LeanXcale that provides replication of all components. However, this topic is outside the scope of this paper.

6. Correctness

In this section, we show that LeanXcale transaction management is correct, i.e., it correctly provides snapshot isolation and 1-copy transactional data movement.

6.1. Snapshot isolation level correctness

We start with some basic definitions. Then, we prove that LeanXcale provides snapshot isolation and session consistency.

Definition 6.1 (*Data Items and Transactions*)**.** A transaction T_i is a sequence of read and write operations bracketed by a start operation s_i and a commit (c_i) or abort (a_i) operation. A data item x starts with its unborn version x_{init} and finishes with its tombstone version x_{dead}. A transaction T_i creates a version x_i of an object x when executing a write over it, denoted by $w_i(x_i)$. The reading of a particular data item version j from a transaction T_i is denoted by $r_i(x_j)$. When a transaction T_i has written a data item x_i, it becomes committed when T_i commits. A data item is read and/or written at most once within a transaction.

Definition 6.2 (*Transaction History*)**.** Let $\mathcal{T}$ be a set of transactions. A history H over $\mathcal{T}$ represents a particular execution of the transactions in $\mathcal{T}$. A history defines two orders:

1. The time precedes order, denoted by $\prec_t$, over the transaction operations, characterized as:
   (a) Every transaction T_i in $\mathcal{T}$ has a start operation (s_i) and a commit (c_i) or abort operation (a_i). All operations of committed transactions appear in H.
   (b) All operations of a transaction T_i are totally ordered according to the order in which they were executed. If o_{ij} was executed before o_{ik} then $o_{ij} \prec_t o_{ik}$.
   (c) If a transaction T_j reads a data item x_i written by T_i then $w_i(x_i) \prec_t r_j(x_i)$.
   (d) Given two transactions T_i and T_j, it follows that either $s_j \prec_t c_i$ or $c_i \prec_t s_j$.
2. A version total order, denoted by $\ll$, among the versions of a data item where x_{init} is the first version and if x_{dead} exists, is also the last version.

Definition 6.3 (*Snapshot Read* [31])**.** All reads performed by a transaction T_i occur at its start point, i.e., if $r_i(x_j)$ occurs in history H, then:

(1) $c_j \prec_t s_i$
(2) If $w_k(x_k)$ also occurs in H ($j \neq k$), then

- Either $s_i \prec_t c_k$

- Or $c_k \prec_t s_i$ and $x_k \ll x_j$

Definition 6.4 (*Snapshot Write* [31]). For any two committed transactions T_i and T_j in H that write the same object x:

(1) Either $c_i \prec_t s_j$ or $c_j \prec_t s_i$
(2) If $c_i \prec_t s_j \prec_t c_j$ then $x_i \ll xj$ and if $c_j \prec_t s_i \prec_t c_i$ then $x_j \ll x_i$

We now provide a definition of snapshot isolation derived from the one provided in [31] without using the GILD formalism.

Definition 6.5 (*Snapshot Isolation*). A transaction history H over $\mathcal{T}$ fulfills Snapshot Isolation when:

1. No aborted reads. For any aborted transaction T_i in H, any committed transaction T_j cannot read any object modified by T_i.
2. No intermediate reads. For any committed transaction T_i in H, no transaction T_j can read a version of an object x written by T_i that was not the final modification of x.
3. No circular dependencies. H cannot contain circular information flow, i.e., if T_i reads or writes a version written by T_j, T_j cannot read or modify a version written by T_i.
4. H fulfills Snapshot Read (Definition 6.3).
5. H fulfills Snapshot Write (Definition 6.4).

Definition 6.6 (*Session*). A session S_i is a subset of the transactions participating in a history H, such that $S_i \subseteq \mathcal{T}$ with a total order among the transactions in the session, denoted by $\prec_{S_i}$. All sessions defined in a history H are disjoint, i.e., $\forall S_i, S_j \in \mathcal{S}, i \neq j. S_i \cap S_j = \emptyset$.

Definition 6.7 (*Session Consistency*). A transaction processing system provides session consistency when it is guaranteed that all transactions within a session S_m of a history H observe all the updates of previous transactions in that session, i.e., $\forall T_i, T_j \in S_m, i \neq j, T_i \prec_{S_m} T_j \implies c_i \prec_t s_j$.

Let us now prove that our solution provides snapshot isolation (Definition 6.5). In order to do so, we first prove that a concurrent execution is equivalent to a snapshot isolation execution in the commit timestamp order. We first prove that there are no aborted reads (Definition 6.5.1), no intermediate reads (Definition 6.5.2), nor circular dependencies(Definition 6.5.3). Then, we prove that we guarantee the snapshot read property (Definition 6.3). Finally, we also prove that the conflict detection algorithm satisfies the snapshot write property (Definition 6.4), providing the first updater wins version of it.

Finally, since the isolation levels deal with the ordering of individual transactions but not session consistency, we prove that LeanXcale also provides session consistency (Definition 6.7). This adds an extra requirement for the visibility of updates across transactions within the same session, i.e., that transactions in a session observe the writes of all previous update transactions in that session.

Lemma 6.1. *LeanXcale does not suffer aborted reads.*

Proof. Since LeanXcale uses private versions for the updates of transactions before a transaction is readable, no transaction can ever read an object written by an aborted transaction because the aborted transaction will never become either durable or readable. □

Lemma 6.2. *LeanXcale does not suffer intermediate reads.*

Proof. Also due to the use of private versions, it is impossible for any transaction to read any intermediate result. Lines 6–7 in Algorithm 2 show that private versions are only accessible to the active transaction in a session that created them. Only final results will become readable by other transactions and this only happens when the transaction is committing. □

Lemma 6.3. *LeanXcale does not exhibit circular dependencies.*

Proof. The proof is by contradiction. There are only four possible cycles: read–read, read–write, write–read, and write–write:

- Assume that T_i reads x from T_j ($r_i(x_j)$) and T_j reads y from T_i ($r_j(y_i)$). For T_j to read x from T_i, T_i should have committed and thus, its updates would be visible to T_j. Thus, T_j would have a start timestamp equal or higher than the CTS of T_i, $cts_i \leq sts_j$. If T_i has read data from T_j, then $cts_j \leq sts_i$. Since $sts_j < cts_j$ and $sts_i < cts_i$, we would get: $cts_i \leq sts_j < cts_j \leq sts_i < cts_i$, which leads to a contradiction.
- Assume T_i reads x from T_j ($r_i(x_j)$) and T_j writes y that was already written by T_i ($y_j \gg y_i$). For T_i to read x from T_j, T_j should have committed and thus, be visible for T_i. Thus, T_i would have a start timestamp equal or higher than the CTS of T_j: $cts_j \leq sts_i$. For T_j to write a version $y_j \gg y_i$, T_j can only do while T_i is not yet committed, in which case it would be aborted by the conflict manager, or after T_i has already committed. In the latter case, either T_j is concurrent to T_i and then the conflict manager would still abort T_j because of the write–write conflict between concurrent transactions or T_j starts after the commit of T_i which implies that the start timestamp of T_j is equal or higher than the CTS of T_i: $cts_i \leq sts_j$. In the latter case, since $sts_i < cts_i$ and $sts_j < cts_j$, we would get: $cts_j \leq sts_i < cts_i \leq sts_j < cts_j$, which leads to a contradiction.
- Assume that T_j reads x from T_i ($r_j(x_i)$) and T_i writes y that was already written by T_j ($y_i \gg y_j$). For T_j to read x from T_i, T_i should have committed and thus, be visible for T_j. Thus, T_j would have a start timestamp equal or higher than the CTS of T_i: $cts_i \leq sts_j$. For T_i to write a version $y_i \gg y_j$, T_i can only do while T_j is not yet committed, in which case it would be aborted by the conflict manager, or after T_j has already committed. In the latter case, either T_i is concurrent to T_j and then the conflict manager would still abort T_i because of the write–write conflict between concurrent transactions or T_i starts after the commit of T_j, which implies that the start timestamp of T_i is equal or higher than the CTS of T_j, i.e., $cts_j \leq sts_i$. For the latter, since $sts_j < cts_j$ and $sts_i < cts_i$, we would get: $cts_i \leq sts_j < cts_j \leq sts_i < cts_i$, which leads to a contradiction,
- Assume T_i writes x_i after T_j writes x_j, $x_i \gg x_j$ and T_j writes y_j after T_i writes y_i, $y_j \gg y_i$. For the former, both transactions could be concurrent or sequential. If they are concurrent, T_i would be aborted due to the write–write conflict. If they are sequential, then we would have: $cts_j \leq sts_i$. For the latter, similarly, both transactions can be concurrent or sequential. In the concurrent case, T_j would be aborted by the conflict manager. In the sequential case, we would have: $cts_i \leq sts_j$. Since $sts_j < cts_j$ and $sts_i < cts_i$, we would get: $cts_j \leq sts_i < cts_i \leq sts_j < cts_j$, which leads to a contradiction.

Since all possible cases lead to a contradiction, the lemma is proven. □

Lemma 6.4. *LeanXcale transaction management satisfies the snapshot read property (Definition 6.3).*

Proof. The proof is by contradiction. Assume the snapshot read property (Definition 6.3) is not satisfied. Then, there are two cases:

1. $r_i(x_k)$ and ($c_k \prec_t c_j \prec_t s_i$ and $x_k \ll x_j$). Thus, a transaction T_i reads an older version (x_k) that is not the last one before its snapshot. Then, although the current snapshot when T_i started was the one corresponding to the CTS of T_j, T_i read x_k that corresponds to an earlier snapshot. However, this is impossible since when the snapshot reaches the CTS of T_j, all transactions with CTS lower than or equal to the CTS of T_j are already durable and readable. Lines 5–8 in Algorithm 6 show that the snapshot only progresses where there are no gaps in the serialization order in terms of durable and readable transactions. Thus, the versions of their updated data are already in the KiVi servers. This is shown in lines 20–21 in Algorithm 2, where `snapshotServer.DurableAndRedable` is only invoked after `logger.Log` and `dataManager.Write`. This makes the transaction durable and readable. Therefore, when performing the read operation, it is guaranteed that the KiVi servers get the oldest version that has a version number lower than or equal to the snapshot, corresponding to the CTS of T_j (see lines 3–4 in Algorithm 1). So, this leads to a contradiction.
2. $r_j(x_i)$ and ($c_k \prec_t s_j \prec_t c_i$ and $x_k \ll x_i$). Thus, T_j reads a younger version, x_i written by T_i, which is after T_j's snapshot sts_j. There are five cases (all the possible states of a transaction from ongoing to visible):
   (a) x_i is still private. In this case, it is impossible that any other transaction reads it, thus leading to a contradiction. Lines 6–7 in Algorithm 2 show that only a session can read from the private versions stored in the session local writeset.
   (b) T_i is durable but x_i has not yet been written in a KiVi server. As in the previous case, x_i could not be read by T_j from storage, thus leading to a contradiction. This is shown in lines 16–20 in Algorithm 2: since line 20 has not yet been executed, x_i only exists in the session local writeset so it cannot be read by any other transaction.
   (c) x_i is in storage, but T_i is not yet readable. Since T_i is not readable, the snapshot is lower than the CTS of T_i and thus can never be chosen by the storage engine as version to be read, which leads to a contradiction This is shown in lines 16–21 in Algorithm 2: since line 21 has not yet been executed, x_i is readable since it is written on the KiVi server but because the CTS has not yet been notified as durable and readable and thus the snapshot is lower than that CTS, no transaction can read (see lines 2–4 in Algorithm 1).
   (d) T_i is readable but not visible. Although it is readable, T_i is still not visible. Since the snapshot is lower than T_i's CTS, it can never be chosen by the storage engine as the version to be read, which leads to a contradiction (see lines 2–4 in Algorithm 1).
   (e) T_i is visible. The snapshot server provides newly started transactions with a snapshot equal to or higher than the CTS of T_i, cts_i. However, we have $sts_j < cts_i$, so the KiVi server will never return x_i as version of x for the read operation, since it only returns versions with a timestamp equal to or lower than sts_j, which leads to a contradiction (see lines 2–4 in Algorithm 1).

Since all possible five cases lead to a contradiction, the lemma is proven. □

Lemma 6.5. *LeanXcale transaction management satisfies the snapshot write property (Definition 6.4).*

Proof. The proof is by contradiction. Assume that the snapshot write property (Definition 6.4) is not satisfied. Then, there are two cases:

1. Two concurrent transactions modify the same row and then commit. This leads to a conflict that is detected by the conflict manager that handles the row key. One transaction update notification will be processed before the other by this conflict manager. The notification that is received first, and thus also processed first, will be successful (see lines 5–7 in Algorithm 3). The other transaction will fail and the transaction will be aborted (see lines 3–4 in Algorithm 3 and lines 14–15 in Algorithm 2). Since all conflicts are solved before starting the commit, this leads to a first updater wins strategy and guarantees that two concurrent transactions cannot both update the same row and commit, which yields a contradiction.
2. Two non-concurrent transactions, T_i and T_j, produce versions in reverse order that they have been committed, i.e., $c_i \prec_t c_j$ and $x_j \ll x_i$. Since $c_i \prec_t c_j$, T_i has necessarily a CTS lower than T_j because LeanXcale uses the commit timestamp order as commit order (see Algorithm 6). This commit order is applied to the data versioning where the commit timestamp ordering is used to choose the version to be read (see lines 2–4 in Algorithm 1). Thus, we have $c_i \prec_t c_j \implies x_i \ll x_j$, which leads to a contradiction.

Since the two cases result in a contradiction, the lemma is proven. □

Theorem 6.6. *LeanXcale transaction management provides snapshot isolation (Definition 6.5).*

Proof. Snapshot isolation (Definition 6.5) requires to fulfill five properties. Lemmas 6.1, 6.2, 6.3, 6.4, 6.5 prove each of the properties, thus, LeanXcale transactional management satisfies snapshot isolation. □

Lemma 6.7. *LeanXcale transaction management provides session consistency (Definition 6.7).*

Proof. The proof is by contradiction. Assume that session consistency is not guaranteed in session S_m. Thus, a transaction $T_k \in S_m$ will not observe a snapshot that includes the updates of all update transactions in S_m that committed before T_k in S_m. Violating session consistency means that T_k, instead of reading a version x_j of $T_j \in S_m$, reads a row with a version x_i written by T_i that committed before T_j, i.e., $r_k(x_i), x_i \ll x_j, C_i \prec_t C_j \prec_t S_k$.

Since the session consistency mechanism enforces that the start timestamp sts_k of any new started transaction T_k is greater than the CTS cts_j of any transaction $T_j \in S_m, T_j \prec_{S_m} T_k$, we have: $C_j \prec_t S_k$. For this to happen, T_j should become durable and readable before the start of T_k and the snapshot server has to advance the snapshot to a value equal to or higher than cts_j. The snapshot getting the value of cts_j or higher means that all committed transactions with CTSs lower than or equal to cts_j were also durable and readable. Thus, it is impossible that T_k reads row x_i, since LeanXcale guarantees that the biggest version that will be read by T_k has a CTS lower than or equal to the snapshot associated to the reading transaction, sts_k, which means that it will return a version of x that is x_j or older, but not younger. This leads to a contradiction and thus, the lemma is proven. □

6.2. 1-copy transactional data movement

Elastic transaction management relies on the movement of transactional data partitions during transaction processing. LeanXcale guarantees full consistency during the movement of a transactional data partition. We formalize the concept as 1-copy transactional data movement, meaning that the behavior of the system during data movement is equivalent to that without any data movement with a single data partition not being moved.

Lemma 6.8. *No updates lost.*

Proof. The data partition movement starts by configuring all client sides to send updates to both the origin and destination KiVi servers. Suppose for the sake of contradiction that at least one update is lost. If the update is lost, it can only be because the update did not reach the destination KiVi server since the destination KiVi server would apply all buffered updates. Thus, the update would have been sent before the client side that produces it was reconfigured. This leads to a contradiction since the data partition movement would only start after the client side was reconfigured to send to both the origin and destination KiVi servers. □

Lemma 6.9. *Idempotence of updates.*

Proof. Updates are identified by the primary key of the row they modify and the CTS. If an update that is applied on the origin KiVi server is also buffered at the destination KiVi server, and thus re-applied, the row with that primary key would be labeled as a version equal or higher to the CTS of that transaction (higher in the case that some updates happened on the origin KiVi server before the state of the row was sent to the destination KiVi server) and that update would be discarded. Thus, updates are idempotent and only effectively applied once in the order of the CTSs. □

Lemma 6.10. *Synchronous replica after data partition movement is completed.*

Proof. After the data partition is moved, i.e., all the rows from the data partition have been copied to the destination KiVi server, the destination KiVi server has a copy of the data partition with a fraction of the updates that happened after all client sides were reconfigured to send to the origin and destination servers. Furthermore, the destination KiVi server has all the updates that have been sent since the data partition was moved (see Lemma 6.8) and continues to receive all the updates over that data partition as part of each transaction. Thus, once all buffered updates have been applied, it becomes a synchronous replica of the origin data partition. □

Theorem 6.11. *Data partition movement satisfies 1-copy guarantees.*

Proof. From Lemmas 6.8–6.10, it follows that no updates are lost and that duplicate updates are applied only once. It also follows that after the data partition movement is completed, both origin and destination data partitions are updated synchronously. Thus, the process guarantees that the data partition behaves as 1-copy that is not moved. □

7. Performance evaluation

This section evaluates the scalability and elasticity of transaction processing in LeanXcale using the TPC-C benchmark and microbenchmarks on Amazon Web Services (AWS). After introducing our experimental setup, we present our experimental results.

7.1. Experimental setup

Cluster configuration. For the TPC-C benchmark, we deploy LeanXcale using two kinds of nodes: metadata nodes and data nodes. The metadata nodes contain instances of metadata components (configuration manager, conflict manager, logger, …). The data nodes contain the query engine, LTM and KiVi servers. We use a cluster of up to 200 AWS i3en.2xlarge servers as data nodes and up to 20 AWS i3en.2xlarge servers as metadata nodes. Each i3en.2xlarge instance has 8 vCPUs (4 physical cores with hyperthreading), 64 GB of memory and 2 NVMe disks of 2,5TB. For load injection, we use 1 AWS server for every 20 LeanXcale data servers.

LeanXcale deployment. One node is used for the meta-data components across all configurations from 1 to 200 servers. Each data node has 4 instances of KiVi servers, and one instance of the query engine and LTM. There is one metadata node for every 20 data nodes or fraction of them. Each metadata node has one instance of the Apache Zookeeper configuration manager, 4 instances of the conflict manager, and 8 instances of the logger. Each 4 loggers are assigned to one of the two disks in each AWS instance. There is a minimum of 5 zookeeper instances, so there are always at least 5 metadata nodes.

TPC-C dataset deployment: The TPC-C dataset is deployed as follows. Each server node handles 2,000 data warehouses. All the TPC-C tables are partitioned using the warehouse id, except the read-only table Item that is replicated at all nodes.

7.2. Scalability

We run TPC-C with 4 configurations of 1, 20, 100 and 200 data servers in AWS. The experiments (see Fig. 5) show linear scale out from 1 to 200 servers, reaching 11 million transactions per minute (TPM). In particular, we reach about 5 million New Order TPM, which is greater than the TPC-C throughput obtained by the 9th highest result in all history. Out of these 9 results only 5 are in a cluster deployment [32].

For TPC-C, there are only two results released in a cloud data center, which are also the two top ones. These results are from Alibaba Cloud Elastic using 65,394 and 6,720 cores, respectively, yet used exclusively (not shared with other users as we do) for the benchmark. Trying with those high-end configurations was beyond our economic capabilities. However, LeanXcale's efficiency per core (TPM per core) is much higher, with 22700+ new order TPM per core versus 10800+, i.e., double that of Alibaba. The other systems use highly tuned bare metal hardware (SANs with enough memory to keep all data in memory) while we run on a public cloud with shared instances.

Fig. 6 shows the average latency (response time) of different TPC-C transactions (NewOrder, Delivery, Payment, StockLevel and OrderStatus) for the largest deployment, i.e., with 200 AWS instances. Note that the latencies are quite below the SLA required by the benchmark (5 s for all transactions except StockLevel that is 20 s). Another important observation is that the latencies are pretty steady. Latencies for low loads are under millisecond. The greater latencies shown in Fig. 6 are due to the system getting close to saturation, which yields larger request queues and thus corresponding waiting times.

7.3. Elasticity

To evaluate the elasticity of the different transactional components described in Section 3, we exercise several microbenchmarks, one per component (logger, conflict manager, KiVi server). We also study the effectiveness of load balancing across KiVi servers and the global elasticity of all components.

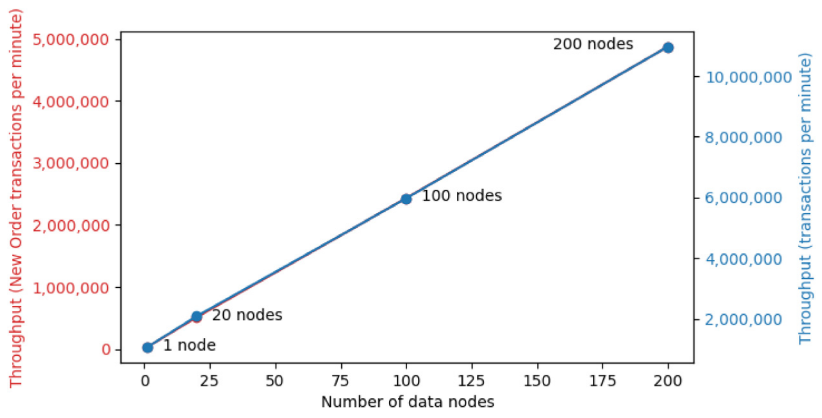

Fig. 5. TPC-C Scalability Results for 1, 20, 100 and 200 Nodes.

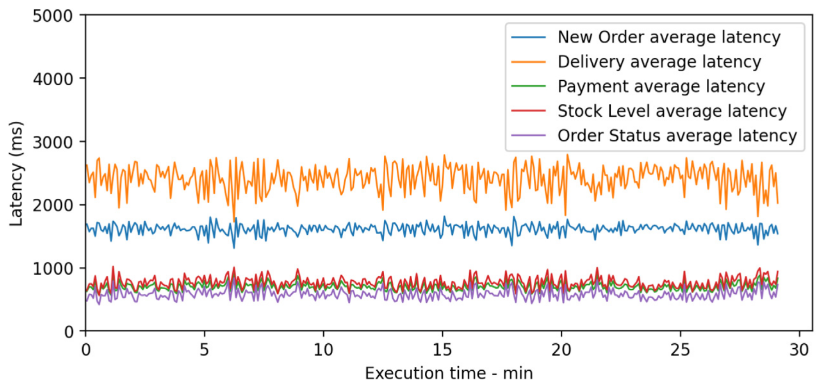

Fig. 6. Transaction Latency for 200 Server Deployment.

7.3.1. *Logger elasticity*

It is exercised by increasing the update transaction load in terms of number of warehouses (actually, the client threads of the warehouses) during around 15 min until the threshold for triggering logger elasticity is reached. Then, the load is kept stable for half an hour and then, the load is started to be reduced. The results are shown in Fig. 7. On the left *y* axis, we can see the number of messages processed by each logger (initially, one logger). On the right *y* axis, we can see the load in terms of number of warehouses, where each warehouse has 10 client threads as mandated by TPC-C. The load is increased from 4,000 until 40,000 warehouses. When the load goes beyond 30,000 warehouses, which results in over 40,000 log messages per second, elasticity is triggered, and a new logger is provisioned (orange line). Then, half of the LTMs connect to the new logger and disconnect from the initial one. This process is quite fast, taking only a few seconds to commute to the new logger. This is possible because LeanXcale transaction management does not have any requirement in terms of log ordering. Once the switch of half of the LTMs is completed, both loggers get half the load of the log messages. Finally, around minute 42, the load starts to decrease. When the load gets below 20,000 log messages per logger, elasticity is triggered again and all LTMs sending load to the second logger switch back to the first logger. In a few seconds, the process is completed, and the second logger decommissioned. Then, the first logger gets the full load, which is below 40,000 log messages per second and goes down to 5000 messages per second.

7.3.2. *Conflict manager elasticity*

The micro-benchmark for conflict manager elasticity is similar to that for loggers. The load is increased in terms of warehouses (actually, the client threads allocated to each warehouse) until the elasticity threshold is reached. Then, the load is sustained for half an hour and then decreased until reaching the initial load. The results are depicted in Fig. 8. The load is shown with double *y* axis as before. On the left *y* axis, the load is shown in terms of conflicts per second received by each conflict manager (initially one, see the orange line). On the right *y* axis, the load is shown in terms of warehouses. The increase in warehouses has a faster effect on the load received by the conflict managers in terms of conflicts per second, since they are received along the transaction instead of at the end as with the loggers. When the load goes above 160,000 conflicts per second, a new conflict manager is provisioned. Unlike with logger elasticity, the process takes longer, but there is the advantage that no state is transmitted across conflict managers. Each of them receives all the conflict information and at some point, the conflicts known only by the first conflict manager are forgotten because they are obsolete. Thus, at that point, the second conflict manager knows all the needed conflicts and the responsibility of half of the conflict buckets is transferred to the second conflict manager. Then, the first conflict manager starts receiving half the load. The process for provisioning the second conflict manager until the first conflict manager handles half the load takes about two minutes. Then, after 30 min of constant load, the load is decreased. This process takes a little less than the first two minutes, since the load is smaller and it takes less time for the information kept by the second conflict manager to become totally obsolete, at which point, all its buckets are transferred to the first conflict manager that starts to handle the full load. The responsibility of conflict buckets just requires sending configuration messages to all LTMs. The advantage of this process is that it does not require to stop the processing at any point, thus not disrupting the QoS of transaction management.

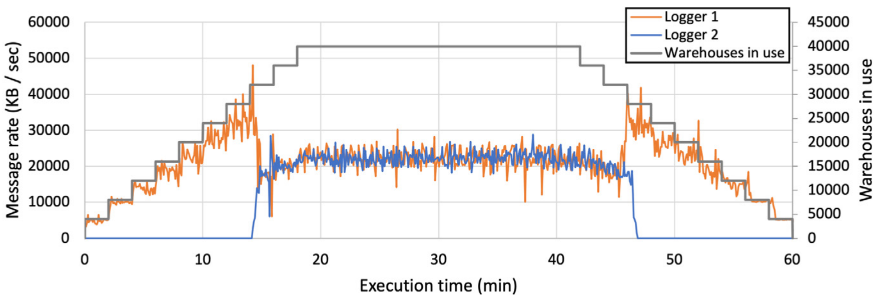

Fig. 7. Logger Elasticity.

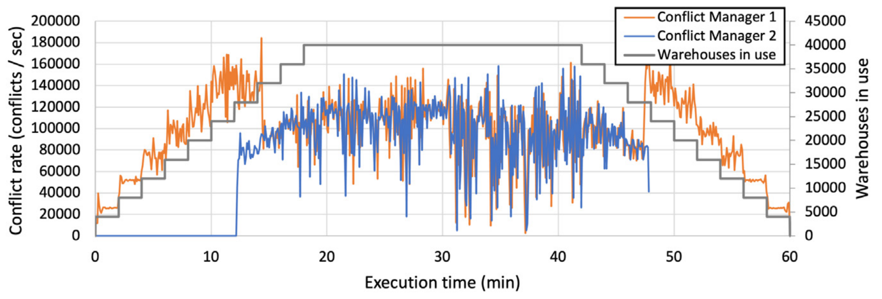

Fig. 8. Conflict Manager Elasticity.

7.3.3. KiVi Server elasticity

In this micro-benchmark, we show the elasticity of the KiVi servers. The results are shown in Fig. 9. The top chart shows the CPU usage of the two servers and the bottom chart the evolution of the overall throughput and response time. We run one KiVi server in charge of 200 warehouses. Initially, only the load for the first 100 warehouses is generated. Five minutes later, the load for the other 100 warehouses starts to be injected. Each table partition corresponds to 100 warehouses. This increases the load in the KiVi server above the elasticity threshold, which triggers the provisioning of a new KiVi server five minutes later with another KiVi server. One of the table partitions starts to be transferred to the second KiVi server. The transfer is completed in less than 1.5 min, including the time it takes for the second KiVi server to process all the buffered requests during the transfer. As can be seen in the figure, the only effect in the QoS is a slight increment in average response time of 10 ms during the state transfer across the two KiVi servers. The throughput is kept steady all the time.

7.3.4. Load balancing across KiVi servers

We show how the load of table partitions is balanced across KiVi servers. We focus on KiVi server load balancing, which is the most challenging since it is stateful. The micro-benchmark is run with 2 KiVi servers. The results are shown in Fig. 10. Each table partition contains information of 40 warehouses. The first KiVi server is responsible for 9 table partitions and the second one for 7 table partitions. There are 80 warehouses for which no load is injected, corresponding to 2 table partitions for which KiVi server 1 is responsible. Thus, each KiVi server is receiving exactly the same load corresponding to 7 table partitions, i.e., 280 warehouses. Ten minutes after the start of the experiment, the load starts to be injected in the two table partitions (i.e., 80 warehouses) without any load. This causes the first KiVi server to get significantly more load than the second one. Then, 2 min later, load balancing is triggered and one table partition is moved from the first KiVi server to the second one. The table partition movement takes one minute to be completed. At this point, each KiVi server is handling half the workload. As can be seen during the table partition movement, there is a load increase in both servers. The first server receives more load and sends a copy of one of the table partitions to the second server. The second server gets a load increase because it receives a copy of the table partition. Thus, it must also buffer the requests corresponding to this table partition, and when the copy is finished, it has to apply the buffered updates. The full process takes around 3 min and at that point, the load of both servers is fully balanced, each one processing 160 warehouses.

7.3.5. Global elasticity

To evaluate the elasticity of all components, we deploy one node with data populated for 1,200 TPC-C warehouses. This node has one instance of each transactional component (KiVi server, logger, conflict manager, query engine, commit sequencer, snapshot server and LTM). Then, we start injecting load corresponding to the first 200 warehouses. Every 15 min, the load for another 200 new warehouses is added to the overall workload. Each load increase makes the overall load to go above the threshold, which triggers the provisioning of a new node. Each new node contains an instance of all components. After provisioning a new node, the load is balanced across all nodes. The warehouses still with no load remain as they are while the rest of the load corresponding to the active warehouses is balanced across all servers.

Fig. 11 top left shows the overall throughput and average latency. There are a couple of peaks of about 50 ms every time the load is increased. The first peak corresponds to initial tasks for balancing the data to the new server. The second peak corresponds to the finalization of these tasks, mainly to apply all the buffered updates to the data partitions of the moved tables. The process of moving all data across all servers (KiVi servers, conflict managers) is completed in about 5 min. Note that the load increase is done in steps, while in a real workload, the change in load would be smoother, thus giving us more time to adapt.

The throughput evolution of the conflict managers is shown in Fig. 11 bottom right. When the load is increased, the number of conflicts per second in the conflict managers increases. This increase is each time smaller, since the load increase is shared across all conflict managers. Such behavior stems from the fact

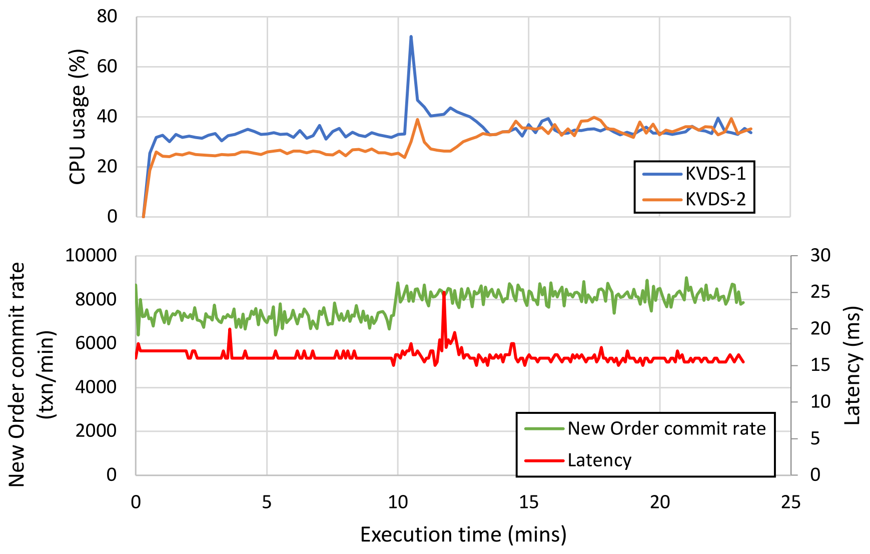

Fig. 9. KiVi Server Elasticity.

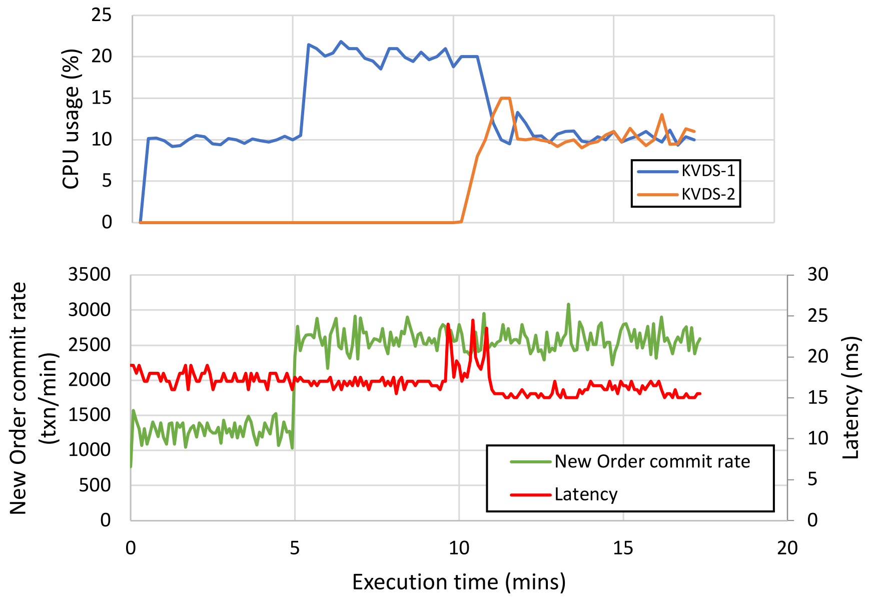

Fig. 10. KiVi Server Load Balancing.

that the conflicts are hashed so their load is more or less evenly distributed across buckets.

Fig. 11 top right shows the throughput evolution in the loggers. Each load increase is actually received by the first node. Thus, the impact on the load is always the same, since the logger of the first node gets its load doubled each time. This is because we increase the load on warehouses without load and these warehouses are not actually moved until they receive load.

Fig. 11 bottom left shows the evolution of CPU usage of the KiVi servers. As for loggers, the load increase falls in the first KiVi server since it is the one that contains the data partitions of warehouses with no load. This doubles the load of the first KiVi server. In the new KiVi server, there is a first period where the load is low, as it is just buffering updates and receiving the data partition messages. Then, its CPU usage raises to the level of the other KiVi servers.

In summary, this experiment shows the elasticity of all transactional components that, by working together, deliver the elasticity of the LeanXcale database system as a whole. Such elasticity is non intrusive so that the QoS is barely affected during the elastic reconfigurations.

8. Conclusion

In this paper, we presented our solution to elastic scalable transaction processing in LeanXcale, an industrial-strength NewSQL database system. Unlike previous solutions, it does not require any special hardware assistance, such as RDMA-enabled networks or specialized clocks. Furthermore, its architecture is different than most NewSQL database systems such as Spanner that rely on a NoSQL key–value store. Our approach to elastic scalable transaction management is based on a number of principles, which we divided in three groups: scalable, efficient and elastic transaction management.

We proved that LeanXcale transaction management is correct, i.e., it correctly provides snapshot isolation and 1-copy transactional data movement.

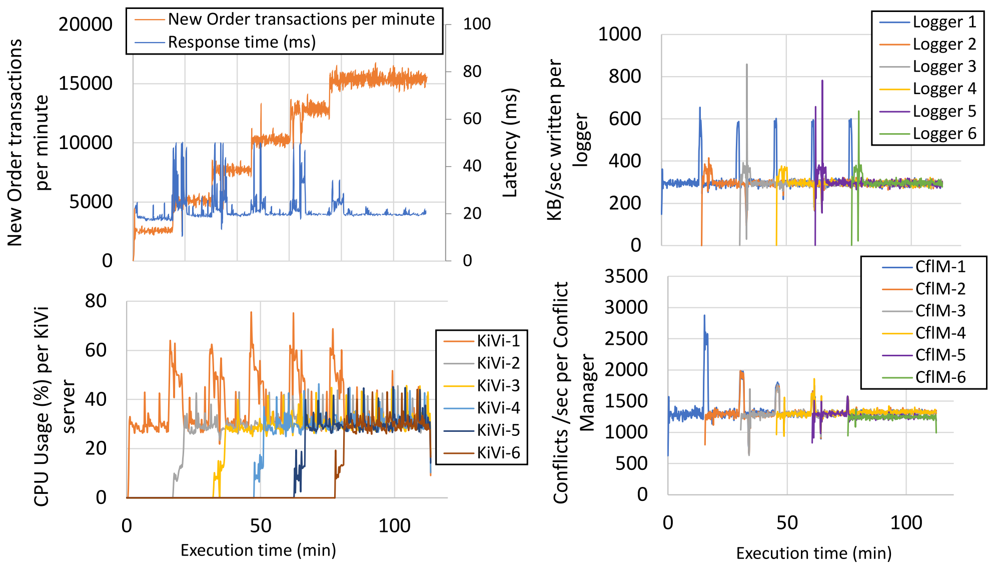

Fig. 11. Overall Throughput and Latency.

Finally, we provided a thorough performance evaluation of our solution on AWS. The experimental results using the TPC-C benchmark show that LeanXcale can scale linearly from 1 to 200 servers and reach 5 million of NewOrder TPM and 11 million TPM overall. Furthermore, they also show double efficiency per core (i.e., 2*TPM per core) compared to the two top TPC-C results from Alibaba, which are the only results in a cloud data center but with no shared instances.

Elasticity is demonstrated individually for all transactional components (logger, conflict manager, KiVi server) and globally. In particular, it is shown that all servers (i.e., loggers, KiVi servers, …) can be scaled up and down without impacting the QoS as well as balancing the load across KiVi servers.

As far as we know, this is the only industrial-strength solution for transaction processing that provides both scalability and elasticity in the cloud without resorting to any special hardware, with its performance evaluation in a public cloud with shared instances.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially funded by the European Commission under the projects Infinitech (contract #856632), Cybele (contract #825355) and PHYSICS (contract #101017047), the Comunidad de Madrid under the industrial doctorate grant #IND2018_TIC-9846 and EDGEDATA P2018/TCS-4499 and, by the Spanish Research Council under the projects QoSDATA (PID220-119461GB-100/AEI/10.13039/501100011033) and CloudDB (contract TIN2016-80350-P).

References

[1] R. Jimenez-Peris, F. Ballesteros, Method and system for online data partition movement, 2022, European Patent Office. File application number: #EP22382126.5. Filing date: 16/02/2022.
[2] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, K. Littlefield, P. Tong, F1 - the fault-tolerant distributed rdbms supporting google's ad business, in: SIGMOD, 2012.
[3] T. Özsu, P. Valduriez, Principles of Distributed Database Systems, fourth ed., Springer, 2020.
[4] D. Agrawal, S. Das, A.E. Abbadi, Data Management in the Cloud: Challenges and Opportunities, Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.
[5] H. Garcia-Molina, K. Salem, Sagas, SIGMOD Record (1987) 249–259.
[6] P. Valduriez, R. Jimenez-Peris, M.T. Özsu, Distributed database systems: The case for newSQL, Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS) (2021) 1–15.
[7] R. Taft, E. Mansour, M. Serafini, J. Duggan, A.J. Elmore, A. Aboulnaga, A. Pavlo, M. Stonebraker, E-Store: Fine-grained elastic partitioning for distributed transaction processing systems, Proceedings of the VLDB Endowment (PVLDB) (2014) 245–256.
[8] F. Färber, S.K. Cha, J. Primsch, C. Bornhövd, S. Sigg, W. Lehner, SAP HANA database: data management for modern business applications, ACM SIGMOD Record (2011).
[9] M. Stonebraker, A. Weisberg, The voltDB main memory database system, IEEE Data Engineering Bulletin (2013).
[10] Y. Lin, B. Kemme, M. Patiño-Martınez, R. Jimenez-Peris, Middleware-based data replication providing snapshot isolation, in: SIGMOD Conference, 2005, pp. 419–430.
[11] M. Patiño-Martinez, R. Jimenez-Peris, B. Kemme, G. Alonso, MIDDLE-R: Consistent database replication at the middleware level, ACM Transactions on Computer Systems (TOCS) (2005) 375–423.
[12] Y. Lin, B. Kemme, M. Patiño-Martınez, R. Jimenez-Peris, Snapshot isolation and integrity constraints in replicated databases, ACM Transactions on Database Systems (TODS) (2009) 1–49.
[13] P. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.
[14] R. Jimenez-Peris, M. Patiño-Martınez, G. Alonso, B. Kemme, Are quorums an alternative for data replication? ACM Transactions on Database Systems (TODS) (2003) 257–294.
[15] A. Dragojevic, D. Narayanan, E.B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, M. Castro, No compromises: distributed transactions with consistency, availability, and performance, in: USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2015, pp. 54–70.
[16] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojevic, D. Narayanan, M. Castro, Fast general distributed transactions with opacity, in: SIGMOD Conference, 2019, pp. 433–448.
[17] A. Kalia, M. Kaminsky, D.G. Andersen, FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs, in: USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2016, pp. 185–201.
[18] E. Zamanian, C. Binnig, T. Harris, T. Kraska, The end of a myth: Distributed transactions can scale, Proceedings of the VLDB Endowment (PVLDB) (2017) 685–696.
[19] J.C. Corbett, et al., Spanner: Google's globally distributed database, in: USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2012, pp. 251–264.
[20] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S.B. Zdonik, E.P.C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, D.J. Abadi, H-Store: A high-performance, distributed main memory transaction processing system, Proceedings of the VLDB Endowment (PVLDB) (2008) 1496–1499.

[21] S. Machine, Splice Machine: The Only Hadoop RDBMS, Tech. Rep., 2014, http://www.splicemachine.com/wp-content/uploads/sp.WhitePaper_141015.pdf.

[22] EsgynDB, EsgynDB: Enterprise Class Operational SQL-on-Hadoop, Tech. Rep., 2016, https://esgyn.com/wp-content/uploads/EsgynDB-Technical-Whitepaper-v2.1.pdf.

[23] Apache, Hbase, 2021, https://hbase.apache.org/.

[24] P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil, The log-structured merge-tree (LSM-tree), Acta Informatica (1996) 351–385.

[25] R. Jimenez-Peris, M. Patiño-Martinez, System and method for highly scalable decentralized and low contention transaction processing, 2011, European Patent #EP2780832, US Patent #US9, 760, 597.

[26] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, D. Shasha, Making snapshot isolation serializable, ACM Transactions on Database Systems (ACMTDS) (2005) 492–528.

[27] M.J. Cahill, U. Röhm, A.D. Fekete, Serializable isolation for snapshot databases, in: SIGMOD Conference, 2008, pp. 729–738.

[28] W. Vogels, Eventually consistent, Communications of the ACM (2008) 14–19.

[29] C. Chekuri, S. Khanna, On multidimensional packing problems, SIAM Journal on Computing (SICOMP) (2004) 837–851.

[30] Y. Jia, I. Brondino, R. Jimenez-Peris, M. Patiño-Martinez, D. Ma, A multi-resource load balancing algorithm for cloud cache systems, in: Proceedings of the ACM Symposium on Applied Computing, SAC, 2013, pp. 463–470.

[31] A. Adya, Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions (Ph.D. thesis), Massachusetts Institute of Technology, 1999.

[32] The Transaction Processing Performance Council, TPC-C Top Results, Tech. Rep., 2021, http://tpc.org/tpcc/results/tpcc_results5.asp.