SolarDB: Toward a Shared-Everything Database on Distributed Log-Structured Storage

TAO ZHU, East China Normal University
ZHUOYUE ZHAO and FEIFEI LI, University of Utah
WEINING QIAN and AOYING ZHOU, East China Normal University
DONG XIE and RYAN STUTSMAN, University of Utah
HAINING LI and HUIQI HU, Bank of Communications

Efficient transaction processing over large databases is a key requirement for many mission-critical applications. Although modern databases have achieved good performance through horizontal partitioning, their performance deteriorates when cross-partition distributed transactions have to be executed. This article presents SolarDB, a distributed relational database system that has been successfully tested at a large commercial bank. The key features of SolarDB include (1) a shared-everything architecture based on a two-layer log-structured merge-tree; (2) a new concurrency control algorithm that works with the log-structured storage, which ensures efficient and non-blocking transaction processing even when the storage layer is compacting data among nodes in the background; and (3) find-grained data access to effectively minimize and balance network communication within the cluster. According to our empirical evaluations on TPC-C, Smallbank, and a real-world workload, SolarDB outperforms the existing shared-nothing systems by up to 50x when there are close to or more than 5% distributed transactions.

11

CCS Concepts: • **Information systems → Database transaction processing**; **Relational parallel and distributed DBMSs**;

Additional Key Words and Phrases: Shared-everything architecture, log-structured storage, concurrency control

ACM Reference format:
Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, and Huiqi Hu. 2019. SolarDB: Toward a Shared-Everything Database on Distributed Log-Structured Storage. *ACM Trans. Storage* 15, 2, Article 11 (June 2019), 26 pages.
https://doi.org/10.1145/3318158

This article is an extended version of the paper entitled Solar: Towards a Shared-Everything Database on Distributed Log-Structured Storage, published in USENIX ATC'18.
T. Zhu, W. Qian, and A. Zhou were supported by the 863 Program (2015AA015307), National Key R&D Plan Project (2018YFB1003303), and NSFC (61432006 and 61332006). F. Li, Z. Zhao, and D. Xie were supported in part by NSF grants 1619287 and 1443046. F. Li was also supported in part by NSFC grant 61729202. R. Stutsman was supported in part by NSF grant CNS-1750558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.
Authors' addresses: T. Zhu, W. Qian, and A. Zhou, East China Normal University, 3663 Zhongshan N. Rd, Shanghai, China 200241; emails: zhutao@stu.ecnu.edu.cn, {wnqian, ayzhou}@sei.ecnu.edu.cn; Z. Zhao, F. Li, D. Xie, and R. Stutsman, University of Utah, 50 Central Campus Dr, Salt Lake City, UT, USA, 84112; emails: {zyzhao, lifeifei, dongx, stutsman}@cs.utah.edu; H. Li and H. Hu, Bank of Communications, 188 Yincheng Middle Rd, Shanghai, China, 200090; emails: {lihn, hqhu}@bankcomm.com.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1553-3077/2019/06-ART11 $15.00
https://doi.org/10.1145/3318158

1 INTRODUCTION

The success of NoSQL systems has shown the advantage of the scale-out architecture for achieving near-linear scalability. However, it is hard to support transaction in those systems, an essential requirement for large databases, due to the distributed data storage. For example, Bigtable [5] only supports single-row transactions, whereas others like Dynamo [6] do not support transactions at all. In response to the need for transaction support, NewSQL systems are designed for efficient OnLine Transaction Processing (OLTP) on a cluster with distributed data storage.

Distributed transaction processing is hard because of the need for efficient synchronization among nodes to ensure ACID properties and maintain good performance. Despite the significant progress and success achieved by many recently proposed systems [8, 9, 12, 19, 23, 27, 29, 34, 38], they still have various limitations. For example, the systems relying on the shared-nothing architecture and two-phase commit (2PC) heavily suffer from cross-partition distributed transactions and thus require careful data partitioning with respect to given workloads. However, distributed shared-data systems like Tell [19] require specific hardware supports that are not commonly available yet at large scale.

That said, when no prior assumption can be made regarding the transaction workloads, and with no special hardware support, achieving high performance transaction processing on a commodity cluster is still a challenging problem. Meanwhile, prior studies have also shown that it is possible to design high performance transaction engines on a single node by exploring the multi-core and multi-socket (e.g., NUMA) architecture. Both Silo [30] and Hekaton [7] have used a single server for transaction processing and demonstrated high throughput. However, such systems may not meet the needs of big data applications whose data cannot fit on a single node, hence requiring the support for distributed data storage.

Inspired by these observations, our objective is to design a transactional database engine that combines the benefits of scalable data storage provided by a cluster of nodes and the simplicity for achieving efficient transaction processing on a single server node, without making any *a priori* assumptions on the transactional workloads and without requiring any special hardware support.

Bank of Communications, one of the largest banks in China, has faced these challenges. On one hand, new e-commerce applications from its own and its partners' mobile and online apps have driven the need for the support of ad hoc transactions over large data, where little or no knowledge/assumptions can be made toward the underlying workloads, as new apps emerge constantly. On the other hand, the bank has a strong interest toward better utilization of its existing hardware infrastructures to avoid costly new hardware investment, if possible.

With that in mind, SolarDB is designed using a *shared-everything* architecture, where a server node (called *T-node*) is reserved for in-memory transaction processing and many storage nodes (called *S-nodes*) are used for data storage and read access. In essence, the S-nodes in SolarDB form a *distributed storage engine* and the T-node acts as *a main-memory transaction engine*. The distributed storage engine takes advantage of a cluster of nodes to achieve scalability in terms of the database capacity and the ability to service concurrent reads. The transaction engine provides efficient transaction processing and temporarily stores committed updates through its *in-memory committed list*. Periodically, *recently committed data items* on the T-node are merged back into the S-nodes through a *data compaction* procedure running in the background, without interrupting ongoing transactions. Overall, SolarDB is designed to achieve high performance transaction processing and scalable data storage.

To speed up update operations in the system, the in-memory committed list on the T-node and the disk storage from all S-nodes collectively form a distributed two-layer log-structured merge tree (LSM-tree) design [22]. Furthermore, a processing layer called *P-units* is introduced to carry out both data access from the S-nodes and any computation needed in a transaction so that the T-node can be freed from the burden of coordinating data access and performing business logic computation. This separation of storage and computation also enables the system to leverage all CPU resources for transaction scheduling and validation. Toward realizing the preceding design principle, we also design and implement a number of optimizations and algorithms to minimize the overhead in the system. Our contributions are summarized as follows:

- A distributed shared-everything architecture with a T-node, a set of S-nodes, and P-units is proposed for achieving high performance transaction processing.
- A hybrid concurrency control scheme called *MVOCC* is explored that combines the optimistic concurrency control (OCC) and the multi-version concurrency control (MVCC) schemes.
- A data compaction algorithm, as part of MVOCC, is designed to efficiently merge the *committed list* on the T-node back to the S-nodes periodically, without interrupting transaction processing on the T-node.
- Several optimizations are investigated to improve the overall performance, such as separation of computation and storage through the P-units, grouping multiple data access operations in one transaction, and maintaining a bitmap for avoiding unnecessary data access to the distributed storage engine.

In our empirical evaluation on TPC-C, Smallbank, and a real-world workload, SolarDB outperforms existing shared-nothing systems by 50x when the transactions requiring distributed commits are close to or more than 5%.

2 SOLARDB ARCHITECTURE

SolarDB is a distributed shared-everything relational database that runs concurrent transactions on a cluster of commodity servers. Figure 1 shows its architecture.

2.1 Design Considerations

Shared-everything architecture. Shared-nothing architecture [12, 27] places data in non-overlapping partitions on different nodes in the hope that it can avoid expensive 2PC among nodes when almost all of the transactions only need to touch data on one partition and thus can run independently. For distributed transactions, multiple partitions with data involved need to be locked, blocking all other transactions that need to touch those partitions, which greatly increases system latency. Even worse, it only takes merely a handful of distributed transactions to always have locks on all the partitions, and as a result, system throughput can be reduced to nearly zero.

Instead, SolarDB employs a shared-everything architecture, where a transaction processing unit can access any data. ACID can be enforced at a finer granularity of individual records rather than at partitions. It also avoids expensive 2PC by storing updates on a single high-end server, enabling a higher transaction throughput.

In-memory transaction processing and scalable storage. Traditional disk-based databases rely on buffering mechanisms to reduce the latency of frequent random access to the data. However, this is several magnitudes slower than accessing in-memory data due to the limited size of the buffers and complication added to recovery.

In-memory transaction processing proves to be much more efficient than disk-based designs [7, 12]. Limited memory is always a key issue with in-memory transaction processing. Databases must have mechanisms to offload data to stable storage to free up memory for an unbounded stream of transactions. A key observation is that transactions typically only touch a very small subset

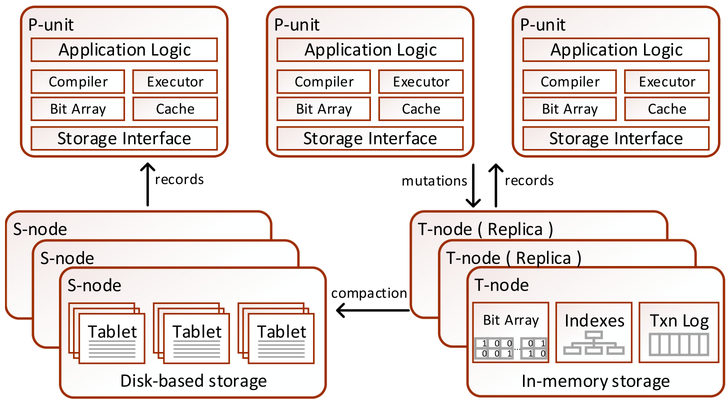

Fig. 1. Architecture of SolarDB.

of the whole database, writing a few records at a time in a database of terabytes of data. Thus, SolarDB reduces transaction processing latency by writing completely in memory while having an unbounded capacity by storing a consistent snapshot on a distributed disk-based storage, which can be scaled out to more nodes if needed.

Fine-grain data access control. In SolarDB, processing nodes directly access data stored in remote nodes via a network, which can lead to overheads. Existing studies have shown that it is advantageous to use networks such as InfiniBand and Myrinet [19]. However, they are far from widely available. They require special software and hardware configurations. It is still unclear how to do that on a cluster of hundreds of off-the-shelf machines.

SolarDB is designed to work on a cluster of commodity servers and thus uses a standard networking infrastructure based on Ethernet/IP/TCP. But network latency is significant because of the transition and data copying into and out of the kernel. It also consumes more CPU than InfiniBand, where data transport is offloaded onto NIC. To address the issue, we designed fine-grain data access to reduce network overhead, including caching, avoiding unnecessary reads, and optimizing inter-node communication via transaction compilation. Fine-grain data access brings the transaction latency on par with the state-of-the-art systems and improves throughput by 3x.

2.2 Architecture Overview

Figure 1 provides an overview of SolarDB's architecture. SolarDB separates transaction processing into *computation, validation*, and*commit* phases using a *multi-version OCC protocol.* A transaction can be initiated on any one of the P-units, which do not store any data except several small data structures for data access optimization (Section 4). The P-unit handles all the data fetches from either the T-node or the S-nodes, as well as transaction processing. The writes are buffered at the P-unit until the transaction commits or aborts. When the transaction is ready to commit, the P-unit sends the write set to the T-node for validation and commit. Once the T-node completes the validation, it writes the updates to its in-memory storage, and also a Write-Ahead Log to ensure durability. Finally, the T-node notifies the P-unit if the transaction is successfully committed. P-units can be instantiated on any machine in or outside the cluster (typically on the S-nodes or at the client side). They offload most of the computation burden from the T-node so that the T-node can be dedicated to transaction management. Cluster information (e.g., states of all nodes, data distribution) are maintained by a manager node and cached by other nodes.

SolarDB adopts a two-layer distributed storage that mimics the LSM-tree [22]. The storage layer consists of (1) a consistent database snapshot and (2) all committed updates since the last snapshot.

The size of the snapshot can be arbitrarily large and thus is stored in a distributed structure called SSTable across the disks of the S-nodes. Records in a table are dynamically partitioned into disjoint ranges according to their primary keys. Each range of records is stored in a structure called tablet (256MB in size by default), which is essentially a B-tree index. The committed updates are stored in Memtable on the T-node, which are from recent transactions and are typically small enough to fit entirely in memory. Memtable contains both a hash index and a B-tree index on the primary keys. The data entry points to all the updates (updated columns only) since the last snapshot, sorted by their commit timestamp. To access a specific record, a P-unit first queries Memtable. If there is no visible version in Memtable, it then queries SSTable for the version from the last snapshot.

The size of Memtable increases as transactions are committed. When it reaches a certain memory threshold or some scheduled off-peak time (e.g., 12:00 am to 4:00 am for Bank of Communications), SolarDB performs a data compaction operation to merge the updates in Memtable into SSTable to free up the memory on the T-node. At the end of data compaction, a new consistent snapshot is created in SSTable and Memtable drops all the committed updates prior to the start of the data compaction.

During data compaction, a new Memtable is created to handle new transactions arriving after the start of the data compaction. Then the old Memtable is merged into SSTable in a way similar to LSM-tree, namely merging two sorted lists from the leaf level of the B-tree index. Instead of overwriting the data blocks with new contents, we make new copies and apply updates on the copies. As we will explain in Section 3, transactions that have already started at the start of data compaction might still need to access the old SSTable. Thus, this approach minimizes the interruption to ongoing transactions.

Note that the function of the T-node is twofold: it works as a transaction manager that performs timestamp assignment, transaction validation, and committing updates; however, it serves as the *in-memory portion of the log-structured storage layer*. This architecture allows low-latency and high-throughput insertion, deletion, and update through the in-memory portion. The log-structured storage also enables fast data compaction, which has a very small impact on the system performance because it mainly consumes network bandwidth instead of the T-node's CPU resource.

Finally, SolarDB uses data replication to provide high availability and resistance to node failures. In SSTable, each tablet has at least three replicas, and they are assigned to different S-nodes. Replication also contributes to achieve better load balancing among multiple S-nodes: a read request can access any one of the replicas. Memtable is replicated on two backup T-nodes. Details of data replication and node failures are discussed in Section 3.2.

3 TRANSACTION MANAGEMENT

SolarDB utilizes both OCC and MVCC to provide *snapshot isolation* [2]. Snapshot isolation is widely adopted in real-world applications, and many database systems (e.g., PostgreSQL prior to 9.1, Tell [19]) primarily support snapshot isolation, although it admits the write-skew anomaly that is prevented by serializable isolation. This article focuses on SolarDB's support for snapshot isolation, and we leave the discussion of serializable isolation to a future work. To ensure durability and support system recovery, redo log entries are persisted into the durable storage on the T-node before transaction commits (i.e., write-ahead logging).

3.1 Supporting Snapshot Isolation

SolarDB implements snapshot isolation through combining OCC with MVCC [2, 15]. More specifically, MVOCC is used by the T-node over Memtable. Recall that each record in Memtable maintains multiple versions. A transaction t_x is allowed to access versions created before its start time, which is called the *read-timestamp* and can be any timestamp before its first read. At the commit time, a transaction obtains a *commit-timestamp*, which should be larger than any existing read-timestamp or commit-timestamp of other transactions. Transaction t_x should also verify that no other transactions ever write any data, between t_x's read-timestamp and commit-timestamp, that t_x has also written. Otherwise, t_x should be aborted to avoid a lost update anomaly [2]. When a transaction is allowed to commit, it updates a record by creating a new version tagged with its commit-timestamp.

With MVOCC, SSTable contains, for all records in the database, the latest versions created by transactions with commit-timestamps are smaller than the *last data compaction time* (*compaction-timestamp*). Memtable contains newer versions created by transactions with commit-timestamps larger than the compaction-timestamp.

The T-node uses a global, monotonically increasing counter to allocate timestamps for transactions. Transaction processing in SolarDB is decomposed into three phases: processing, validating, and writing/committing.

Processing. In the processing phase, a worker thread of a P-unit executes the user-defined logic in a transaction t_x and reads records involved in t_x from both the T-node and the S-nodes. A transaction t_x obtains its read-timestamp (rt_x for short) when it first communicates with the T-node. The P-unit for processing t_x reads the latest version of each record involved in t_x, whose timestamp is smaller than rt_x. In particular, it first retrieves the latest version from Memtable. If a proper version (i.e., timestamp less than rt_x) is not fetched, it continues to access the corresponding tablet of SSTable to read the record. During this process, t_x buffers its writes in a local memory space on the P-unit. When t_x has completed all of its business logic code, it enters the second phase. The P-unit sends a commit request for t_x containing t_x's write-set to the T-node. The T-node would then validate and commit the transaction.

Validating. The validation phase is conducted on the T-node, which aims to identify potential write-write conflicts between t_x and other transactions. During the validation phase, the T-node attempts to lock all records in t_x's write-set (denoted as w_x) on Memtable and checks, for any record $r \in w_x$, whether there is any newer version of r in Memtable whose timestamp is larger than rt_x. When all locks are successfully held by t_x and no newer version for any record in w_x is found, the T-node guarantees that t_x has no write-write conflict and can continue to commit. Otherwise, the T-node will abort t_x due to the lost update anomaly. Hence, after validation, the T-node determines whether to commit or abort a transaction t_x. If it decides to abort t_x, the T-node sends the abort decision back to the P-unit that sent in the commit request for t_x. The P-unit will simply remove the write-set w_x. Otherwise, the transaction t_x continues to the third phase.

Writing/Committing. In this phase, a transaction t_x first creates a new version for each record from its write-set w_x in Memtable and temporarily writes its transaction ID x into the header field of each such record. Next, the T-node obtains a commit-timestamp for t_x by incrementing the global counter. The T-node then replaces the transaction identifier with t_x's commit-timestamp for each record with transaction ID x in Memtable (i.e., those from w_x). Last, the T-node releases all locks held by t_x.

Correctness. Given a transaction t_x with read-timestamp (rt_x) and commit-timestamp (ct_x), SolarDB guarantees that t_x reads a consistent snapshot of the database and there is no lost update anomaly.

Consistent snapshot read. First, t_x sees the versions written by all transactions committed before rt_x because those transactions have finished creating new versions for their write-sets and obtained their commit-timestamps before t_x is assigned rt_x as its read-timestamp. Second, the remaining transactions in the system always write a new data version using a commit-timestamp that is larger than rt_x. Hence, their updates will not be read by t_x. Hence, t_x always operates on a consistent snapshot.

Prevention of lost update. A lost update anomaly happens when a new version of record r is created by another transaction for $r \in w_x$ and the version's timestamp is in the range of (rt_x, ct_x). Assume that the version is created by t_y. There are two cases:

(1) t_y acquired the lock on record r prior to t_x's attempt to lock r. Thus, t_x only gets the lock after t_y has committed and created a new version of r. Hence, t_x will see the newer version of r during validation and be aborted.
(2) t_y acquires the lock on r after t_x has secured the lock. In this case, t_y will not be able to obtain a commit timestamp until it has acquired the lock released by t_x, which means $ct_y > ct_x$. This contradicts with the assumption that the new version of r has a timestamp within (rt_x, ct_x). Recall that the timestamp of a new version for a record $r \in w_y$ is assigned the commit-timestamp of t_y.

3.2 System Recovery

Failure of a P-unit. When a P-unit fails, a transaction may still be in the processing phase if it has not issued the commit request. Such a transaction is treated as being aborted. For transactions in either the validation or the committing phase, they can be terminated by the T-node without communicating with the failed P-unit. The T-node will properly validate a transaction in this category and decide whether to commit or to abort. Both the snapshot isolation and durability are guaranteed, and all affected transactions are properly ended after a P-unit fails.

Failure of the T-node. The T-node keeps its Memtable in main memory. To avoid data loss, it uses WAL and forces redo log records to its disk storage for all committed transactions. When the T-node fails, it is able to recover committed data by replaying the redo log. Moreover, to avoid being the single point of failure, SolarDB also synchronizes all redo log records to two replicas of the T-node using a primary-backup scheme. Each replica catches up the content of the T-node by replaying the log. When the primary T-node crashes, all actively running transactions are terminated, and further transaction commit requests are redirected to a secondary T-node quickly. As a result, SolarDB is able to recover from a T-node failure and resume services in just a few seconds.

Failure of an S-node. An S-node failure does not lead to loss of data as an S-node keeps all tablets on disk. The failure of a single S-node does not negatively impact the availability of the system because all tablets have at least three replicas on different S-nodes. When one S-node has crashed, a P-unit can still access records of a tablet from the copy on another S-node.

3.3 Snapshot Isolation in Data Compaction

Data compaction recycles memory used for Memtable. It produces a new SSTable by merging the current Memtable from the T-node into SSTable on S-nodes.

Data compaction. Let m_0 and s_0 be the current Memtable and SSTable, respectively. Data compaction creates a new SSTable s_1 by merging m_0 and s_0. An empty Memtable m_1 replaces m_0 on the T-node to service future transaction writes. Note that s_1 contains the latest version of each record originally stored in either m_0 or s_0 and is a consistent snapshot of the database. It indicates that there is a timestamp t_{dc} for the start of compaction such that transactions committed before t_{dc} store their updates in s_1 and transactions committed after t_{dc} keep new versions in m_1.

When data compaction starts, the T-node creates m_1 for servicing new write requests. A transaction is allowed to write data into m_0 if and only if its validation phase occurred before data compaction started. The T-node waits until all such transactions have committed (i.e., no more transaction will update m_0 any more). At this point, the S-nodes start to merge m_0 with their local tablets. An S-node does not overwrite an existing tablet directly. Rather, it writes the new tablet using the copy-on-write strategy. Thus, ongoing transactions can still read s_0 as usual. An S-node

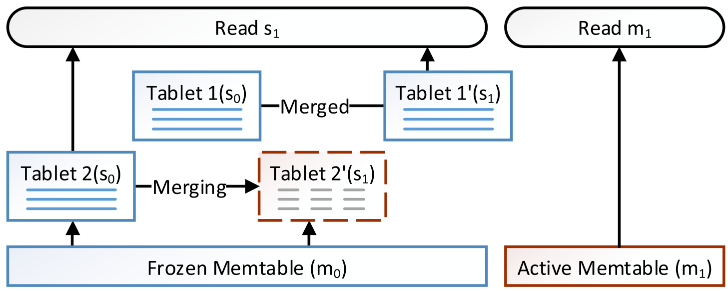

Fig. 2. Data access during data compaction.

acknowledges the T-node when a tablet on that S-node involving some records in m_0 is completely merged with the new versions of those records from m_0. Data compaction completes when all new tablets have been created. The T-node is now allowed to discard m_0 and truncate the associated log records.

Figure 2 illustrates how to serve read access during data compaction. A read request for any newly committed record versions (after t_{dc}) is served by m_1; otherwise, it is served by s_1. There are two cases when accessing s_1: if the requested record is in a tablet that has completed the merging process, only the new tablet in s_1 needs to be accessed (e.g., Tablet $1'$ in Figure 2); if the requested record is in a tablet that is still in the merging process (e.g., Tablet 2 in Figure 2), we need to access that tablet from both s_0 and m_0.

Concurrency control. Snapshot isolation needs to be upheld during data compaction. The following concurrency control scheme is enforced. First, if a transaction starts its validation phase before a data compaction operation is initiated, it validates and writes on m_0 as described in Section 3.1. Second, a data compaction operation can acquire a timestamp t_{dc} only when each transaction that started validation before the data compaction operation is initiated either aborts or acquires a commit-timestamp. Third, the data compaction can actually be started once all transactions with a commit-timestamp smaller than t_{dc} finish. Fourth, if a transaction t_x starts its validation phase after a data compaction operation is initiated, it can start validation only after the data compaction operation obtains its timestamp t_{dc}. The transaction t_x validates against both m_0 and m_1 but only writes to m_1. During validation, t_x acquires locks on both m_0 and m_1 for each record in its write set w_x and verifies that no newer version is created relative to t_x's read-timestamp. Once passing validation, t_x writes its updates into m_1, after which t_x is allowed to acquire its commit-timestamp. Fifth, if a transaction acquires a read-timestamp that is larger than t_{dc}, it validates against and writes to m_1 only.

Correctness. Consistent snapshot read is guaranteed by assigning a read-timestamp to each transaction. Its correctness follows the same analysis as discussed for the normal transaction processing. The preceding procedure also prevents a lost update during data compaction. Consider a transaction t_x with read-timestamp rt_x and commit-timestamp ct_x. Assume that another transaction t_y exists, which has committed between rt_x and ct_x (i.e., $rt_x < ct_y < ct_x$), and t_y has written some records that t_x will also write later after t_y has committed. We only need to consider the case where $ct_y < t_{dc} < ct_x$, since, otherwise, a lost update anomaly is guaranteed not to happen because both t_x and t_y will validate against the same set of Memtables (m_0 and/or m_1). This leads to the situation where $rt_x < ct_y < t_{dc} < ct_x$. Thus, t_x will be validated against both m_0 and m_1, and it will guarantee to see the committed updates made by t_y. As a result, t_x will be aborted since it will find at least one record with a timestamp greater than its read-timestamp rt_x. Hence, a lost update anomaly still never happens even when data compaction runs concurrently with other transactions.

Recovery. The recovery mechanism is required to correctly restore both m_0 and m_1 when a node fails during an active data compaction. Data compaction acts as a boundary for recovery. Transactions committed before the start of the latest data compaction (that was actively running when a crash happened) should be replayed into m_0, whereas those committed after that should be replayed into m_1. Furthermore, we do not need to replay any transactions committed before the completion of the latest completed data compaction, as they have already been successfully persisted to SSTable through the merging operation of that completed data compaction. To achieve that, a compaction start log entry (CSLE) is persisted into the log on disk storage, when a data compaction starts, to document its t_{dc}. A compaction end log entry (CELE) is persisted when a data compaction ends with its t_{dc} serving as a unique identifier to identify this data compaction.

That said, failure of any P-unit does not lead to data loss or impact data compaction. When the T-node fails, the recovery procedure replays the log from the CSLE with timestamp t_{dc}, which can be found in the last CELE. Initially, it replays the log into the Memtable m_0. When the next CSLE is encountered, it creates a new Memtable m_1 and replays subsequent log entries into m_1. The merge process of m_0 into the S-nodes continues after m_0 is restored from the recovery.

If an S-node fails during a data compaction, no data is lost since S-nodes use disk storage. But an S-node β may still be in the process of creating new tablets when it fails. Thus, when β recovers and rejoins the cluster, it contains the tablets of old SSTable and incomplete tablets produced during merging. If the system has already completed the data compaction (using other replicas for the failed node), there is at least one replica for each tablet in the new SSTable. The recovered node β simply copies the necessary tablets from a remote S-node. If data compaction has not completed, β would continue merging by reading records in m_0 from the T-node.

Storage management. During data compaction, m_0 and s_0 (the existing SSTable before the current compaction starts) remain read-only while s_1 and m_1 are being updated. When compaction completes, m_0 and s_0 are to be truncated. But they can only be truncated when no longer needed for any read access. To that end, m_0 may be accessed by some long-running transactions whose read-timestamps are smaller than t_{dc} even when compaction has completed. The T-node remembers the last time m_0 is used by any transaction. A time-out mechanism is used to avoid any transactions idling for too long. The T-node truncates m_0 when all such transactions have either ended or timed out. However, m_0 and s_0 may also be used by transactions whose read-timestamps are larger than t_{dc}. The snapshots they operate on could be provided by either s_0, m_0 and m_1, or s_1 and m_1. Before compaction is finished, s_1 does not physically exist. Hence, a transaction needs to get the latest data version from m_0, s_0, and m_1. After compaction is finished, the s_1 is available for reading. Thus, those transactions with read-timestamps larger than t_{dc} can immediately switch its access to m_1 and s_1 at this point and no longer needs to access m_0 and s_0. In summary, m_0 and s_0 can be truncated when the data compaction has completed and no transaction has a read-timestamp smaller than t_{dc}.

4 OPTIMIZATION

It is important for SolarDB to reduce the network communication overhead among the P-units, the S-nodes, and the T-node. To achieve better performance, we design fine-grain data access methods between the P-units and the storage nodes.

4.1 Optimizing Data Access

The correct data version that a transaction needs to read is defined by the transaction's read-timestamp, which could be stored either in SSTable on the S-nodes or in Memtable on the T-node. Thus, SolarDB does not know where a record (or columns of a record) should be read from, and the P-units have to access both SSTable on the S-nodes and Memtable on the T-node to ensure read consistency (although one of which will turn out to be an incorrect version).

Here, we first present an SSTable cache on the P-units to reduce data access between the P-units and the S-nodes. Then, an asynchronous bit array is designed to help the P-units identify potentially useless data accesses to the T-node.

4.1.1 SSTable Cache. A P-unit needs to pull records from SSTable. These remote data accesses can be served efficiently using a data cache. The immutability of SSTable makes it easy to build a cache pool on a P-unit. The cache pool holds records fetched from SSTable and serves data accesses to the same records.

The cache pool is a simple *key-value store*. The *key* stores the primary key, and the *value* holds the corresponding record. All entries are indexed by a hash map. A read request on a P-unit first looks for the record from its cache pool. Only if there is a cache miss doe the P-unit pull the record from an S-node and add it to its cache pool. The cache pool uses a standard buffer replacement algorithm to satisfy a given memory budget constraint.

Since SSTable is immutable and persisted on disk, SolarDB does not persist the cache pools. Entries in a cache pool do expire when the SSTable from they were fetched is obsolete after a data compaction operation. A P-unit builds a new cache pool when that happens.

4.1.2 Asynchronous Bit Array. SSTable is a consistent snapshot of the whole database. In comparison, Memtable only stores the newly created data versions after the last data compaction, which must be a small portion of the database. As a result, most likely a read request sent to the T-node would fetch nothing from the T-node. We call this phenomenon *empty read*. These requests are useless and have negative effects. They increase latency and consume the T-node's resources.

To avoid making many empty reads, the T-node uses a succinct structure called *memo structure* to encode the existence of items in Memtable. The structure is periodically synchronized to all P-units. Each P-unit examines its local memo to identify potential empty reads.

The memo structure is a bit array. In the bit array, each bit is used to represent whether a column of a tablet has been modified or not. In other words, if any record of a tablet T has its column C modified, the bit corresponding to (T, C) is turned on. Otherwise, the bit is turned off. Other design choices are possible, such as to encode the record-level information, but that would increase the size of the bit-array dramatically.

SolarDB keeps two types of bit arrays. The first type is a real-time bit array on the T-node, denoted as b. The second type is an asynchronous bit array on each P-unit, which is a copy of b at some timestamp t, denoted as $b' = b_t$, where b_t is the version of b at time t. A P-unit queries b' to find potential empty reads without contacting the T-node.

On the T-node, b is updated when a new version is created for any column of a record for the first time. Note that when a version is created for a data item (a column value) that already exists in Memtable, it is not necessary to update b, as that has already been encoded in b. Each P-unit pulls b from the T-node periodically to refresh and synchronize its local copy b'.

During query processing for a transaction t_x on a P-unit p, p examines its local b' to determine whether the T-node contains newer versions for the columns of interest of any record in t_x's read set. If (T, C) is 0 in b' for such a column C, p treats the request as an empty read and does not contact the T-node; otherwise, p will send a request to pull data from the T-node.

Clearly, querying b' leads to false positives due to the granularity of the encoding, and such false positives will lead to empty reads to the T-node. Consider in tablet T that row r_1 has its column C updated and row r_2 has not updated its column C. When reading column C of r_2, a P-unit may find the bit (T, C) in b' is set while there is no version for $r_2.C$ on the T-node. In fact, the preceding

Table 1. Possible Operations in a Physical Plan

Read	Memtable read
	SSTable read
Write	Update local buffer on P-unit (local operation)
Process	Expression, project, sort, join ... (local operation)
Compound	Loop, branch

method is most effective for read-intensive or read-only columns. They seldom have their bits turned on in the bit array.

Querying b' may also return false negatives because it is not synchronized with the latest version of b on the T-node. Once a false negative is present, a P-unit may miss the latest version of some values it needs to read and end up using an inconsistent snapshot. To address this issue, a transaction must check all potential empty reads during its validation phase. If a transaction sees that the bit for (T, C) is 0 in b' during processing, it needs to check whether the bit is also 0 in b during validation. If any empty read previously identified by b' cannot be confirmed by b, a transaction has to be re-processed by reading the latest versions in Memtable. False negatives are rare because b does not see frequent update: it is only updated at the first time any row in tablet T has its column C modified.

4.2 Transaction Compilation

SolarDB supports JDBC/ODBC connections and stored procedures. The latter takes the one-shot execution model [26] and avoids client-server interaction. This poses more processing burden on the DBMS but enables server-side optimizations [34, 40, 41]. SolarDB leverages the server-side optimization opportunity and designs a compilation technique to reduce inter-node communication during transaction processing by generating an optimized physical plan.

Execution graph and dependency constraints. A stored procedure may be compiled into different instances of physical plans when provided with different input parameters and database snapshots. The physical plan, to be executed by a P-unit, of a stored procedure is represented as a sequence of operations in Table 1 (nested structures, e.g., branch or loop, can be viewed as a compound operation). Reads are implemented via remote procedure calls, whereas write and process/computation are *local function calls* on the P-unit. Hence, reads are the key to optimizing network communication.

Two operations have to be executed in order if they satisfy one of the following:

(1) *Procedure constraint*: Two operations have data/control dependency [21]. This ensures that the values of the variables read or written are correct and any control flow is executed correctly.
(2) *Access constraint*: Two operations access the same database record, and one operation is a write. This case can be interpreted as a special data dependency over records from the database, which is not covered by procedure constraint.

If two operations do not satisfy any of the two constraints, they can be arbitrarily reordered. In the appendix, we will show that a reordered physical plan has the same semantics as the original plan if it does not violate the preceding constraints.

Constraints between a pair of operations are determined by their variable and record read/write sets. Variables used by an operation can be easily found during compilation. It is, however, not always the case for database records because the record id may not be determined until runtime.

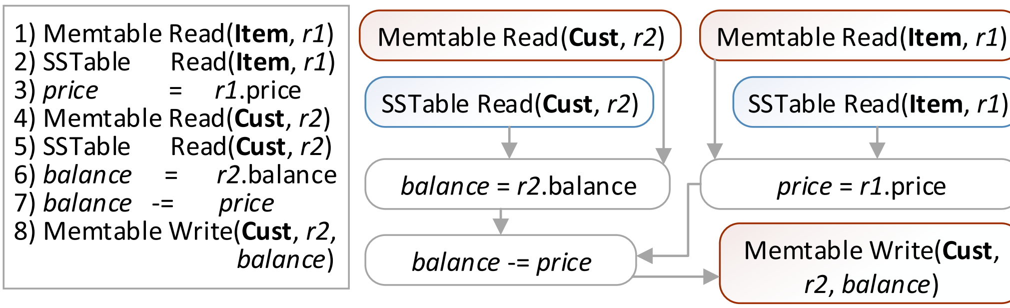

Fig. 3. Example of operation sequence and execution graph.

In practice, we treat it as a potential access constraint if two operations are accessing the same table and one of them is a write.

Then we can represent an operation sequence as an *execution graph*, where the nodes are operations and edges are the constraints and represent the execution order (Figure 3).

We also support branches and loops as compound operations. A compound operation is a complex operation if it contains multiple reads. If it only contains one read, the compound operation is viewed as the same type of read (defined in Table 1) as that single read. Otherwise, a compound operation is viewed as a local operation. We adopt loop distribution [14] to split a large loop into multiple smaller loops so that they can be categorized more specifically. For a read operation in a branch block, it can be moved out for speculative execution because reads do not have side effect and thus are safe to execute even if the corresponding branch is not taken.

Grouping Memtable **reads.** To reduce the number of RPCs to the T-node, we can group multiple Memtable reads together in one RPC to the T-node if they do not have constraints between them. It saves multiple round trips between the P-unit and the T-node and thus reduces transaction latency.

Finding the grouped Memtable reads can be done in two passes over the physical plan (see Algorithm 1). The first pass finds all the unconstrained Memtable reads by marking all operations that are constrained by some Memtable reads as `block`, via a BFS over the execution graph. Unconstrained Memtable reads are marked as `group` instead. Complex operations are not grouped because there may be constraints among the Memtable reads within the nested structure themselves. The second pass starts from the unconstrained Memtable reads and marks all *local operations* that precede them as `group`.

Before executing transaction logics, all of those local operations marked in pass 2 get executed first. Then the Memtable reads marked in pass 1 are sent in a single RPC request to the T-node.

Pre-executing SSTable **reads.** SSTable reads can be issued even before a transaction obtains its read-timestamp from the T-node, because there is only one valid snapshot in SSTable at a time. Note that even during data compaction, at which time we have two snapshots s_0 and s_1, whether to read s_0 and m_0 or s_1 for a read request on a tablet can be solely determined by whether the tablet has finished merging or not, which is irrelevant to the read-timestamp. Hence, we can concurrently execute them while executing other operations, as long as the pre-executed SSTable reads are not constrained by other operations.

During execution, the result of a SSTable read might or might not be used depending on if there is an update to the same record in Memtable. Although this optimization might introduce unused SSTable reads, the problem can be mitigated by the SSTable cache pool. The main benefit of pre-executing SSTable reads is to reduce wait time and thus reduce latency.

The SSTable reads that can be pre-executed can be found using Algorithm 2 in a similar fashion to the one for grouping Memtable reads. In the first pass of the algorithm, it marks all the

ALGORITHM 1: Grouping Memtable reads

```
Input: operation sequence: seq = (o_1, o_2, ..., o_n)
Output: grouped operation sequence: group_op
Initialize label[1 : n] = normal, group_op = ();
for i = 1 to n do
    if o_i is complex operation then
        label[i] = block;
        continue;
    forall edge e ends with o_i do
        let e starts with o_j;
        if label[j] == (group or block) then
            label[i] = block;
            break;
    if o_i is Memtable read and label[i] ≠ block then
        label[i] = group;
for i = n to 1 do
    if label[i] == group then
        forall edge e ends with o_i do
            let e starts with o_j;
            label[j] = group;
        add o_i to the front of group_op;
```

operations constrained by some Memtable reads or complex operations as `block`. All the unmarked SSTable reads that can be pre-executed are marked as `preexec`. In the second pass, the algorithm marks all the local operations preceding the SSTable reads to be pre-executed as `preexec` as well. During query execution, the operations marked as `preexec` can be executed before the transaction obtains a read-timestamp and may run concurrently along with other operations.

4.3 Bulk Loading

Bulk loading is of great importance for many applications, which imports a large number of records into a database. A simple implementation of bulk loading is to start many connections and insert records into the database in parallel using the SQL interface. However, it is inefficient because it takes time to parse SQL requests and do network communications between the P-units and the other nodes. In this section, we introduce two methods to boost the performance of bulk loading.

Avoid network communications during bulk loading. In normal transaction processing, a P-unit inserts a record r in the following way. It first tries to read records from the S-nodes and the T-node, and only writes r into the T-node if the record does not already exist. In fact, most bulking loading tasks are used to move records from one source database system to another. The source system ensures that each record is unique. Therefore, it is unnecessary to check whether a record exists or not during bulk loading. With such an assumption, it is possible to directly write records into the T-node. The data loading procedure could read records from a large data file and append them into a write request, then send the write request to the T-node. With such schema, we can avoid many unnecessary network communications between a P-unit and the other storage nodes.

A problem with such design is that the data compaction could become the performance bottleneck. Since bulk loading is a write-only task, and usually inserts a large number of records into

ALGORITHM 2: Pre-executing SSTablereads

```
Input: operation sequence: seq = (o_1, o_2, ..., o_n)
Output: the pre-execution sequence: pre_op
Initialize label[1 : n] = normal, pre_op = ();
for i = 1 to n do
    if o_i is complex operation or Memtable -read then
        label[i] = block;
        continue;
    forall edge e ends with o_i do
        let edge e starts with o_j;
        if label[j] == block then
            label[i] = block;
            break;
    if o_i is SSTable-read and label[i] ≠ block then
        label[i] = preexec;
for i = n to 1 do
    if label[i] == preexec then
        forall edge e ends with o_i do
            let edge e starts with o_j;
            label[j] = preexec;
        add o_i to the front of pre_op;
```

the system, the memory capacity of the T-node can be easily exhausted by a single loading task. In this case, SolarDB has to start a data compaction procedure to move records from the T-node to the S-nodes. In this situation, the data compaction procedure can be problematic. In bulk loading, records are usually sorted in the primary key order as they are exported from another database. As a result, the compaction procedure is likely to always append records into the tail of SSTable (i.e., the tablet storing the record with the largest key). This means that there is only one tablet pulling and merging records from the T-node, which greatly limits the parallelism of compaction operation. In addition, since all records are inserted into the tail tablet, the tablet will have to be partitioned into two smaller ones each time its size exceeds the limit. As a result, it takes much longer time to merge records into SSTable.

Avoid data compaction during bulk loading. To tackle the problem described earlier, one possible way is to directly write records into the S-nodes, which can be done in the following steps. The first step is to sort all records to make sure they are in the primary key order. Then we partition these records into several disjoint ranges, each of which has fewer than 64MB of records. We create a tablet structure for each of these ranges of records. Finally, each tablet is copied to three S-nodes. These operations can be efficiently done with a Spark cluster [42]. Let T_e be the set of existing tablets before the bulk loading, and let T_c be the set of newly created tablets. During bulk loading, the range of any tablet $a \in T_c$ should not overlap with that of any existing tablet $b \in T_e$, because all tablets in SSTable must have disjoint ranges as described in Section 2. If some of the records in bulk loading have keys that fall into the ranges of existing tablets, we can use the range information of existing tablets to filter out these records and write them to the T-node instead. These records will be merged into an S-node during a normal data compaction later.

Remarks. The optimizations described in this section are designed for transactions. Other workloads, such as OLAP, require additional optimizations. For OLAP queries, they can be executed on a consistent database snapshot, and some relational operators can be pushed down into storage nodes to reduce inter-node data exchange.

5 SYSTEM MANAGEMENT

SolarDB relies on several R-nodes to handle cluster and schema management tasks. Among these nodes, one is elected as the primary R-node while the rest become backups. These nodes together provide a highly available system management service. To simplify the discussion, we would assume that there is only one R-node at first and discuss how to set up backup R-nodes in Section 5.3.

5.1 Cluster Management

The R-node keeps a living node list to remember all nodes in the cluster. All nodes join into the cluster by registering on the R-node. In their lifetime, they are required to send heartbeat messages to the R-node to remain alive. One is considered to be disconnected if its heartbeat messages are not received by the R-node for a specified period of time. In this case, the R-node would remove that from the living node list.

Each time a P-unit or the T-node connects into the cluster, it simply reports its address to the R-node. When an S-node connects into the cluster, it is required to report its local tablet information to the R-node in additional to its address. The R-node collects reports from all the S-nodes and constructs the complete SSTable distribution. An exception is the database initialization. From the first time SolarDB is started, the S-nodes connect with the R-node without reporting any information. In this case, the R-node starts a bootstrap procedure to initiate necessary data structures on the S-nodes and the T-node. When an S-node s_x is disconnected, the R-node removes tablet information of s_x from the SSTable distribution and broadcasts these changes to all P-units.

5.2 Schema Management

SolarDB has two kinds of schema information. The first kind includes definitions of all tables and indexes. The second kind is the tablet distribution of the SSTable.

Table schema. In SolarDB, DML operations (e.g., *select*,*insert*,*delete*,*update*) are serviced by the P-units, the S-nodes, and the T-node. However, DDL operations (e.g., *create/drop table*) are mainly processed by the R-node. To create a table, the P-unit would generate an execution plan for the operation and send the plan to the R-node. The R-node will check validity of the request (whether the new table uses the same name with any created one), then ask the S-nodes and the T-node to initiate necessary resources for the new table, and finally insert the table schema information into the system table. After the table is created, the R-node broadcasts the change of the schema to all P-units.

SSTable distribution. The SSTable distribution is also viewed as a kind of schema information. Hence, once a data compaction operation is finished, a new SSTable is created and comes into service. Each S-node would report its tablet information to the R-node. After that, the R-node would send these schema changes to all P-units. In addition to the data compaction operation, the distribution of SSTable could also be changed when a tablet is migrated or copied to another S-node. The R-node would monitor the replica number of each tablet and the total number of tablets kept by each S-node. If a tablet has too few replicas due to failure of the S-nodes, the R-node would ask another to keep copies of the tablet. If an S-node $s - x$ stores a large number of tablets because the data compaction merges too many records into s_x, the R-node would try to balance the number of tablets kept over each S-node and migrate some tablets from s_x to another node s_y. After tablets have been moved, the R-node would synchronize the change of data distribution to all P-units.

Schema synchronization. Each P-unit caches a copy of all schema information in the local cache. Although the R-node would send schema updates to all nodes, a P-unit is still required to synchronize the cached version with the R-node periodically. This is because messages sent by an R-node can be lost due to network problems. A P-unit finds its cached schema to be expired in two situations. First, the schema information is marked as expired if a fixed time has passed since the last synchronization (e.g., 10 minutes). Second, a P-unit uses the cached information to generate execution plans for DML requests. When reading records from the storage layer, the S-nodes and the T-node would check whether the schema used by read requests is stale. A data access request is not processed if it uses an out-of-date schema. In both cases, the P-unit is required to refresh the local schema information by contacting the R-node.

5.3 R-node Failure

Next, we introduce how to restart a failed R-node and how to set up multiple R-nodes to provide a highly available service.

First, we consider that there is only one R-node deployed in SolarDB. After the R-node goes down, DDL operations cannot be processed anymore. But the system can still process DML operations until the cached schema information on the P-units expires. When another R-node instance is restarted, it can recover all schema information with the following steps. First, it recovers the SSTable distribution by connecting with the S-nodes. Each S-node would report its local tablets to the new R-node. After the data distribution information is recovered, the R-node can access all database tables. Then it sends inner SQLs to read all table schemas from the system table. After that, the R-node comes into service again.

As we can see, the R-node only collects schema information from the S-nodes and the T-node during the recovery phase rather than persisting these data by itself. Actually, the kind of nodes are stateless, which makes it easy to maintain backups. Typically, SolarDB sets up three R-nodes, one of which is selected as the primary. When the primary r_0 is crashed or disconnected, the rest of the R-nodes vote for a new primary among themselves. After the new primary r_1 is elected, other nodes try to connect with r_1. Here, each node in SolarDB keeps a list of possible R-node addresses. When a node finds that r_0 is not reachable or not the primary, it will try to connect with the rest of the R-nodes and establish a connection with r_1.

6 EXPERIMENT

We implemented SolarDB by extending the open source version of Oceanbase [1]. In total, 58,281 lines were added or modified on its code base. Hence, SolarDB is a full-fledged database system, implemented in 457,206 lines of C++ code. To compare it to other systems that require advanced networking infrastructures, we conducted all experiments using 11 servers on Emulab [39], which allows configuring different network topologies and infrastructures. Each server has two 2.4GHz eight-core E5-2630 processors (32 threads in total when hyper-threading is enabled) and 64GB DRAM, connected through a 1GB Ethernet by default. By default, 10 servers are used to deploy the database system. One server is used to simulate clients. We compared SolarDB to MySQL-Cluster 5.6, Tell (shared-everything) [19], and VoltDB Enterprise 6.6 (shared-nothing) [27]. Although Tell is designed for InfiniBand, we used a simulated InfiniBand over Ethernet to have a fair comparison. We use Tell-1G (Tell-10G) to represent the Tell system using a 1GB (10GB) network, respectively.

SolarDB is not compared to lightweight prototype systems that aim at verifying the performance of new concurrency control scheme, such as Silo [30]. These systems achieve impressive throughput, but their implementations lack many important features, such as fully implemented logging, disaster recovery, and an SQL engine. These features often introduce significant performance overhead but are ignored by these lightweight system prototypes.

SolarDB deploys the T-node on a single server. It deploys both an S-node and a P-unit on each of the remaining nodes. Tell deploys a commit manager on a single server. It uses two servers for storage node deployment and the rest for processing nodes. We tested different combinations

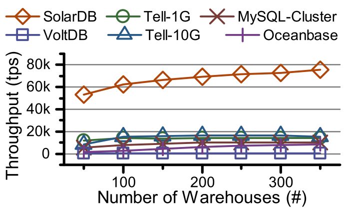

Fig. 4. TPC-C: vary number of warehouses.

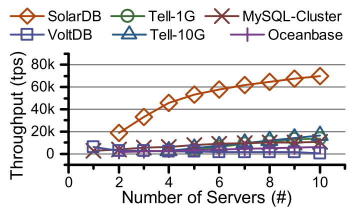

Fig. 5. TPC-C: vary number of servers.

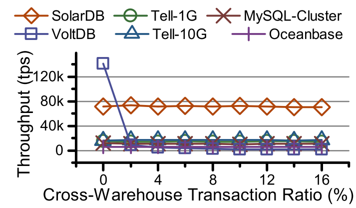

Fig. 6. TPC-C: vary ratio of cross-warehouse transactions.

of processing node and storage node instances and chose the best configuration. Tell uses more processing node instances and fewer storage nodes. MySQL-Cluster deploys both a mysqld and a ndbmtd instance (the multi-threaded process to handle all the data stored using the NDB cluster storage engine of MySQL) on each server. VoltDB creates 27 partitions on each server, which is based on the officially recommended strategy [32]. This was determined by adjusting partition numbers to achieve the best performance on a single server.

We used three different benchmarks. Performance of different systems are evaluated by transactions processed per second (TPS). In each test instance, we adjusted the number of clients to get the best throughput.

6.1 TPC-C Benchmark

We use a standard TPC-C workload with 45% NewOrder, 43% Payment, 4% OrderStatus, 4% Delivery, and 4% StockLevel requests. Request parameters are generated according to the TPC-C specification. By default, 200 warehouses are populated in the database. Warehouse keys are used for horizontal partitioning. Initially, SolarDB stores 1.6 million records (2.5GB) in Memtable and 100 million records (42GB) in SSTable (with 3x replication enabled). After the benchmark finishes, there are 11GB of data in Memtable and the size of SSTable is about 655GB.

Figure 4 shows the performance of different systems when we vary the number of warehouses. SolarDB achieves about 53k TPS on 50 warehouses and increases to about 75k TPS with 350 warehouses. When more warehouses are populated, there is less access contention in the workload, leading to fewer conflicts and higher concurrency. SolarDB clearly outperforms the other systems. Its throughput is 4.8x of that of Tell-10G (about 15.6k TPS) with 350 warehouses. Note that Tell-1G, which uses the same network infrastructure as SolarDB, performs even worse than Tell-10G. VoltDB exhibits the worst performance due to distributed transactions. Last, Oceanbase is primarily designed for processing very short transactions and thus is inefficient on general transaction workloads. SolarDB always achieves at least 10x throughput improvement over Oceanbase. Therefore, we skip Oceanbase in other benchmarks.

Figure 5 evaluates the scalability when using different numbers of nodes. The throughputs of SolarDB, Tell, and MySQL-Cluster increase with more nodes. In contrast, the throughput of VoltDB deteriorates for the following reason. Distributed transactions are processed by a single thread in VoltDB. They block all working threads of the system. With more servers being used, it becomes more expensive for such requests to be processed. The throughput growth in SolarDB slows down with more than seven servers. As there are more access conflicts with a higher number of client requests, more transactions fail in the validation phase. Another reason is that the T-node receives more loads when working with more P-units, and in our experimental setting, the T-node uses the same type of machine as that used for the P-units. Hence, the overall performance increases sub-linearly with the number of P-units. However, in the real deployment of SolarDB, a high-end

Table 2. 90th Latency, TPC-C Workload

Latency(ms)	SolarDB	Tell-1G	Tell-10G	MySQL-Cluster	VoltDB	OB
Payment	6	17	7	17	15,619	38
NewOrder	15	28	12	103	30	60
OrderStatus	6	20	8	23	14	30
Delivery	40	160	53	427	14	174
StockLevel	9	14	7	17	14	60
Overall	12	30	12	95	2,751	54

server is recommended for the T-node, whereas the P-units (and the S-nodes) can be served with much less powerful machines.

Figure 6 shows the results when we vary the ratio of cross-warehouse transactions. If a transaction accesses records from multiple warehouses, it is a distributed transaction. VoltDB achieves the best performance (141k TPS) when there are no distributed transactions, which is about 2.0x that of SolarDB (about 70k TPS). But as the ratio of distributed transactions increase, VoltDB's performance drops drastically as it uses horizontal partitioning to scale out. The other systems are not sensitive to this ratio.

Table 2 lists the 90th latency. SolarDB has a short latency for each transaction. Tell benefits from the better network. It gets better latency with the 10GB network than the 1GB network. The long latency of MySQL-Cluster comes from the network interaction between the database servers and clients because it uses JDBC instead of stored procedures. VoltDB is slow on distributed transactions. Under the standard TPC-C mix, about 15.0% Payment and 9.5% NewOrder requests are distributed transactions. Hence, the 90th latency of Payment is long. Although the 90th latency of NewOrder is small, its 95th latency reaches 15,819ms.

6.2 Smallbank Benchmark

Smallbank simulates a banking application. It contains three tables and six types of transactions. The *user* table contains users' personal information, the *savings* table contains the balances, and the *checking* table contains the checking balances. Each table takes the *account_id* as the primary key. The workload contains 15% Amalgamate transactions, 15% Balance transactions, 15% DepositChecking transactions, 25% SendPayment transactions, 15% TransactSavings transactions, and 15% WriteCheck transactions. Amalgamate and SendPayment operate on two accounts at a time. The other transactions access only a single account. We populated 10 million users into the database. Initially, there are 8M records (3GB) in Memtable and 30M records (1.1TB) in SSTable. After execution, Memtable has 5.2GB of data, and SSTable has about 1.1TB of data.

Figure 7 evaluates different systems by populating different numbers of accounts in the database. Note that the x-axis is shown in log-scale. SolarDB has the best overall performance. Its throughput initially increases as the number of accounts increases, because there is less contention when there are more accounts. Due to the drop of SSTable's cache hit ratio as the number accounts further increases to 10M, the P-units need to issue remote data access to the S-nodes. As a result, its throughput slightly drops.

Tell shows a fairly stable performance, but 10G Ethernet only improves its throughput slightly. MySQL-Cluster also has better performance initially with more accounts but stabilizes once it has maxed out all hardware resources. The performance of VoltDB is limited by cross-partition transactions. Table 3 lists the 90th latency number. It takes VoltDB much longer time than others to process *Amalgamate* and *SendPayment*, and there are 40% such transactions in this workload.

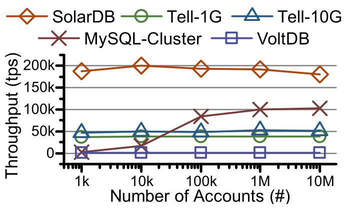

Fig. 7. Smallbank: vary number of accounts.

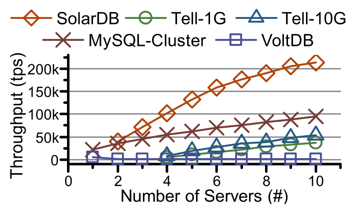

Fig. 8. Smallbank: vary number of servers.

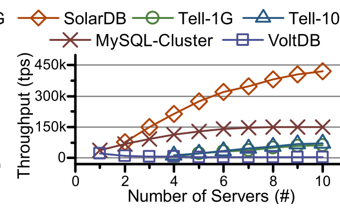

Fig. 9. E-commerce: vary number of servers.

Table 3. 90th Latency, Smallbank Workload

Latency(ms)	SolarDB	Tell-1G	Tell-10G	MySQL-Cluster	VoltDB
Amalgamate	5	5	4	8	100
Balance	3	3	3	4	5
Deposit	4	4	4	3	5
SendPayment	7	4	4	12	102
Xact Savings	3	4	4	6	5
WriteCheck	5	4	4	5	6
Overall	5	4	4	8	92

Figure 8 evaluates each system with different numbers of servers. Here, we populated 1M accounts in the database. SolarDB shows the best performance and scalability with respect to the number of servers. The throughputs of SolarDB, Tell, and MySQL-Cluster scale linearly with the number of servers. The throughput of VoltDB is still quite limited by distributed transaction processing.

6.3 E-commerce Benchmark

E-commerce is a workload from an e-business client of Bank of Communications. It includes seven tables and five transaction types. There are two user roles in this application: buyer and seller. There are four tables for buyers, *User*, *Cart*, *Favorite*, and *Order*, and three tables for sellers, *Seller*, *Item*, and *Stock*. These tables are partitioned by *user_id* and *seller_id*, respectively. At the start of the experiment, SolarDB has 11M records (5GB) in Memtable and 25M records (815GB) in SSTable. When all experiments are completed, Memtable has 8.6GB of data and the size of SSTable is 881GB.

The workload has 88% OnClick transactions, 1% AddCart transactions, 6% Purchase transactions, and 5% AddFavorite transactions. The OnClick request is a read-only transaction, whereas the others are read-write transactions. OnClick reads an item and accesses *Item* and *Stock*. AddCart inserts an item into a buyer's cart and accesses *User* and *Cart*. AddFavorite inserts an item into a buyer's favorite list and updates the item's popular level. It accesses *User*, *Favorite*, and *Item*. Purchase creates an order for a buyer and decrements the item's quantity. It accesses *User*, *Order*, *Item*, and *Stock*.

Figure 9 shows the performance of each system using different numbers of servers. The throughput of SolarDB increases with the number of servers used. It has achieved about 438k TPS when 10 servers are used and is at least 3x that of any other system. As shown in Table 4 for the 90th latency, most transactions are completed within 1ms by SolarDB. MySQL-Cluster and Tell also see performance improvement when more servers are used. However, they have higher latency as shown in Table 4. VoltDB is highly inefficient on AddFavorite and Purchase because tables accessed by these transactions use different partition keys. These transactions may visit multiple partitions

Table 4. 90th Latency, E-commerce Workload

Latency(ms)	SolarDB	Tell-1G	Tell-10G	MySQL-Cluster	VoltDB
OnClick	1	8	4	4	4
AddFavorite	2	12	5	6	47
AddCart	2	2	14	4	4
Purchase	4	12	4	6	49
Overall	1	8	4	4	19

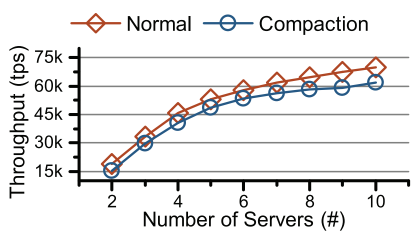

Fig. 10. TPC-C: data compaction.

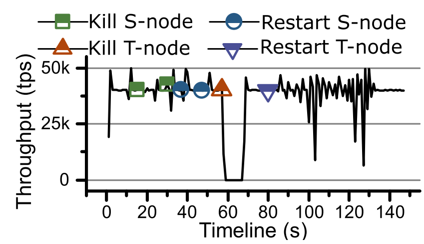

Fig. 11. SolarDB: throughput under node failures.

that block other single-partition transactions. As a result, OnClick and AddCart also have longer latency.

6.4 Data Compaction

During transaction processing, SolarDB may initiate a data compaction in the background. Figure 10 shows the impact of data compaction on the performance when SolarDB is processing the standard TPC-C workload. As shown in Figure 10, data compaction has little negative effect on the performance when five or fewer servers are used. This is because the performance is mainly limited by the number of P-units in these cases, and compaction would not influence the operation of the P-units. When more servers are used, there is about 10% throughput loss. This is because at this point, the T-node has more impact on the overall system performance when more servers are introduced. Data compaction consumes part of the network bandwidth and CPU resources, which are also required by transaction processing on the T-node.

6.5 Node Failures

We next investigate the impact of node failures in SolarDB. In this experiment, three servers were used to deploy the T-nodes, and seven servers were used to deploy the S-nodes and the P-units. One T-node acts as the primary T-node, and the other two are secondary T-nodes. The TPC-C benchmark was used with 200 warehouses populated, and we terminated some servers at some point during execution. Figure 11 plots the changes of throughput against the time.

Removing two S-nodes does not impact the performance, as the SSTable keeps three replicas for each tablet and each P-unit also caches data from SSTable. Thus, losing two S-nodes does not influence performance. We then terminate the primary T-node. Immediately after it goes down, the throughput drops to 0 because no T-node can service write requests now. After about 7 seconds, a secondary T-node becomes the primary and the system continues to function. After the failed T-node re-joins the cluster, the new primary T-node has to read redo log entries from the disk and send them to the T-node in recovery. Thus, the performance fluctuates and drops a little bit due to

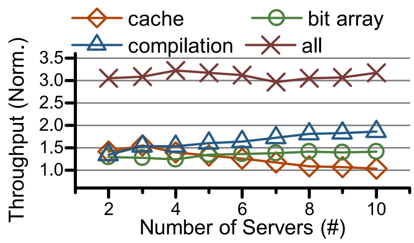

Fig. 12. Improvements under different optimizations.

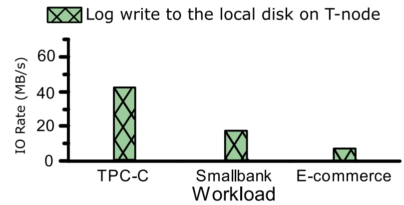

Fig. 13. Log writes to disk on the T-node.

Table 5. S-node Disk Access

Workload	Disk Read Frequency	Disk Read Bytes
TPC-C	4 reads/s	60KB/s
Smallbank	1,700 reads/s	34MB/s
E-commerce	90 reads/s	2MB/s

this overhead. It takes about 40 seconds for the failed T-node to catch up with the new primary, after which the system throughput returns to the normal level.

6.6 Access Optimizations

Figure 12 evaluates the performance improvement brought by different access optimizations. The y-axis shows the normalized performance to a baseline system without using any optimization. The figure shows the improvement brought by enabling each individual optimization, as well as all of them, using the TPC-C workload. Other workloads share similar performance trends. With more P-units and S-nodes deployed in system, the individual optimizations show different trends in improvement. The effectiveness of the SSTable cache drops because the overall data access throughput increases when more S-nodes are deployed. However, the accesses to the T-node are more contentious as more P-units communicate with the single T-node. With transaction compilation enabled, small data accesses to the T-node are combined, which improves the overall throughput when there are more P-units. The bit array shows a relatively stable impact to the throughput because it prunes data access to the T-node at the column level, which is related to the workload rather than the number of servers. As long as a column is not read-only in any row in a tablet, it cannot prune the data access to the T-node. When all optimizations are used together, they bring about 3x throughput improvement regardless of the number of servers used.

6.7 Read and Write Characterization

We investigated the read and write characteristics of SolarDB with respect to the three different workloads used in our experiments. Figure 13 shows the write characteristics on the T-node of the three benchmarks. In particular, we focus on writes to disk on the T-node, which are caused by WAL logging activities (note that all other writes to data records are done entirely in memory on the T-node, hence causing very little overhead). TPC-C is a write-heavy workload, which writes more than 40MB of redo entries to the disk per second. The Smallbank and E-commerce workloads write much fewer redo entries to the disk. However, Table 5 shows that the three workloads have different read characteristics in terms of disk reads on the S-nodes. The TPC-C workload has a small read set and issues few reads to the S-nodes per second. The E-commerce workload has a large read set, but most of its read accesses are effectively served by SolarDB's caching mechanism.

Table 6. Performance of Bulk Loading

Warehouses (#)	Raw File Size (GB)	Approach 1		Approach 2
		Load Into T-node (s)	Data Compaction (s)	Directly Load Into S-node (s)
50	3.4	91	738	96
100	6.8	179	768	147
200	14	349	900	248
400	28	696	2,042	448
800	55	1,369	3,386	883

The Smallbank workload has a more diverse read set and thus ends up using much more disk reads. These different characterizations of reads and writes show that SolarDB is able to adapt to, and achieve efficient and scalable transaction processing on, different real-world workloads.

6.8 Bulk Loading

Table 6 evaluates the performance of different bulk loading methods in SolarDB. This experiment uses the TPC-C benchmark and varies the number of warehouses imported. The size of the database increases from 50 to 800 warehouses, with the size of the raw data file increasing from 3.4 to 55 GB. We mainly compare two different loading methods here. The first approach writes all records into the T-node and then starts a data compaction to merge all data into the S-nodes. Here we show the time used by importing data into the T-node and the time spent on merging data into the S-nodes. The second approach directly writes all records into the S-nodes. Initially, records are organized out of order in the raw data file. A Spark cluster is deployed over 10 nodes to sort all records in their primary key order, partition them into disjoint ranges, and create tablets. Then each tablet is copied to three S-nodes. We show the total time used by writing records into the S-nodes. As shown in Table 6, the first approach spends most of the time on the data compaction. Here the T-node importing time is primarily spent on sending data to the T-node. The performance is mainly determined by the network bandwidth. Thus, the time increases linearly with the size of the raw data file. Data compaction takes more time because there are three replicas for each tablet, and each S-node is only updating one tablet at a time. This means that fewer numbers of tablets can concurrently pull data from the T-node in parallel, and it takes longer to move all data from the T-node to the S-nodes. In comparison, it is much more efficient to load data directly into the S-nodes. The second approach mainly spends time on exchanging data among the nodes. Hence, the time used increases linearly with the total size of the data file. Overall, the second approach achieves 5x to 9x speedup compared to the first approach.

7 RELATED WORK

Single-node system. Single-node in-memory systems have exploited NUMA architectures, RTM, latch-free data structures, and other novel techniques to achieve high performance transaction processing, such as Silo [30], Hekaton [7], Hyper [13, 24], and DBX [35]. The usage of these systems are subject to the main memory capacity on a single node as they require all data stored in the memory. Deuteronomy's [17] transaction component (TC) uses pessimistic, timestamp-based MVCC with decoupled atomic record stores. It can manage data sharded over multiple record stores, but Deuteronomy is not itself networked or distributed; instead, stores are on different CPU sockets. It ships updates to the data storage via log replaying, and all reads have to go through TC. In contrast, SolarDB uses MVOCC and a cluster of data storage, and it can potentially skip the T-node access using its asynchronous bit arrays.

Shared-nothing systems. Horizontal partitioning is widely used to scale out. Examples include HStore [12, 26], VoltDB [27], Accordion [25], and E-Store [28]. We discussed their limitations in Section 2.1. Calvin [29] takes advantage of deterministic execution to maintain high throughput even with distributed transactions. However, it requires a separate reconnaissance query to predict unknown read/write sets. Oceanbase [1] is Alibaba's distributed shared-nothing database designed for short transactions. In shared-nothing systems, locking happens at the partition level. To get sub-partition locking, distributed locks or a central lock manager must be implemented, which goes against the principle for strict partitioning (i.e., get rid of distributed locking/latching), and reintroduces (distributed) locking and latching coordination overheads and defeats the gains of shared nothing. That said, new concurrency control schemes can improve the performance of distributed transactions (e.g., [20]) when certain assumptions are made (e.g., knowing the workload *a priori*, using offline checking, deterministic ordering, and dependency tracking).

Shared-everything systems. The shared-everything architecture is an alternative choice to enable high scalability and high performance, where any node can access and modify any record in the system. Traditional shared-everything databases, like IBM DB2 Data Sharing [11] and Oracle RAC [4], suffer from expensive distributed lock management. Modern shared-everything designs exploit advanced hardware to improve performance, such as Tell [19], DrTM [38] and DrTM+B [37] (with live reconfiguration and data repartitioning), and HANA SOE [10]. SolarDB, however, uses commodity servers and does not rely on special hardware.

Log-structured storage. The LSM-tree [22] is optimized for insertion, update, and deletion. It is widely adopted by many NoSQL vendors, such as LevelDB [18], BigTable [5], and Cassandra [16]. However, none of these supports multi-row transactions. LogBase [31] is a scalable log-structured database with a *log file only storage* where the objective is to remove the write bottleneck and support fast system recovery rather than optimizing OLTP workloads. Hyder II optimizes OCC for tree-structured, log-structured databases [3], which SolarDB may leverage for further improving its concurrency control scheme. vCorfu [36] implements materialized streams on a shared log to support fast random reads. However, it increases transaction latency because committing requires at least four network round trips.

8 CONCLUSION

This work presents SolarDB, a high performance and scalable relational database system that supports OLTP over a distributed log-structured storage. Extensive empirical evaluations have demonstrated the advantages of SolarDB compared to other systems on different workloads. SolarDB has been deployed at Bank of Communications to handle its e-commerce OLTP workloads. We plan to open source SolarDB on GitHub. Current and future works include designing a more effective query optimizer and task processing module by leveraging the NUMA architecture, improving its concurrency control scheme, and designing an efficient and scalable OLAP layer.

APPENDIX

A CORRECTNESS OF TRANSACTION COMPILATION

Definition 1 (Transaction Semantics). Let D be the database. A transaction t is a partial order of actions of the from read(x) or write(x), where $x \in D$; read and write (and multiple writes) applied to the same data item are ordered.

A transaction can be formalized with the preceding definition [33]. Under this model, two transactions are considered to be the same if they share the identical set of read/write actions and have the same partial order over these actions. That said, two physical plans are considered to be the same using the following definition.

Definition 2 (Equivalence). Two plans are considered the same if they generate the same transaction instances whenever provided the same parameters and database state.

Consider a physical plan, $p = o_1, o_2, \ldots, o_n$, and a legally adjusted plan, $p' = o'_1, o'_2, \ldots, o'_n$. Following the steps as described in Section 4.2, we will show that provided with the same input parameters and database snapshot, they must generate the same transaction instance.

If all operations in p' get the same input as those in p, then both p and p' generate the identical set of database accesses. Given any two operations o'_u and o'_v accessing the same record and one of them is a write, their relative order in p' must be consistent with that in p. Otherwise, the access constraint is violated.

If one operation in p' does not get the same input with that in p, assuming o'_i is the first operation that gets different input. It must be resulted from one of the following cases:

(1) *Some o'_j writes a variable that o'_i reads, and o'_j is expected to be executed before(after) o'_i but it is actually executed after(before) o'_i.* This contradicts with the fact o'_i and o'_j have data dependence.
(2) *Two operations o'_u and o'_v write a variable that o'_i reads, and both are expected to execute before o'_i but the relative order is changed, leaving the variable with an incorrect value.* This contradicts with the fact o'_u and o'_v have data dependence.
(3) *o'_i reads a different value of a database record.* If o'_i reads from a write operation in p', then the same discussion presented earlier for variables applies. If o'_i reads from another transaction's write, o'_i must read the same record as p does because p' operates on the same database snapshot.

Hence, p and p' must be equivalent.

ACKNOWLEDGMENTS

The authors greatly appreciate the valuable feedback provided by USENIX ATC reviewers.

REFERENCES

[1] Alibaba Oceanbase. 2015. Oceanbase. Retrieved April 4, 2019 from https://github.com/alibaba/oceanbase.
[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of SIGMOD*, Vol. 24. ACM, New York, NY, 1–10.
[3] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. 2015. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *Proceedings of SIGMOD*. 1295–1309.
[4] Sashikanth Chandrasekaran and Roger Bamford. 2003. Shared cache-the future of parallel databases. In *Proceedings of ICDE*. IEEE, Los Alamitos, CA, 840–850.
[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, et al. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* 26, 2 (2008), 4.
[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, et al. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of SOSP*, Vol. 41. ACM, New York, NY, 205–220.
[7] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, et al. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of SIGMOD*. ACM, New York, NY, 1243–1254.
[8] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of NSDI*. 401–414.
[9] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of SOSP*. 54–70.

[10] Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, et al. 2015. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1716–1727.

[11] J. W. Josten, C. Mohan, I. Narang, and J. Z. Teng. 1997. DB2's use of the coupling facility for data sharing. *IBM Systems Journal* 36, 2 (1997), 327–351.

[12] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, et al. 2008. H-store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[13] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of ICDE*. IEEE, Los Alamitos, CA, 195–206.

[14] Ken Kennedy and Kathryn S. McKinley. 1993. *Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution*. Springer.

[15] Hsiang-Tsung Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems* 2 (1981), 213–226.

[16] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[17] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High performance transactions in Deuteronomy. In *Proceedings of CIDR*. https://www.microsoft.com/en-us/research/publication/high-performance-transactions-in-deuteronomy/.

[18] LevelDB. 2017. Home Page. Retrieved April 4, 2019 from http://leveldb.org/.

[19] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the design and scalability of distributed shared-data databases. In *Proceedings of SIGMOD*. ACM, New York, NY, 663–676.

[20] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting more concurrency from distributed transactions. In *Proceedings of OSDI*. 479–494.

[21] Steven S. Muchnick. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufmann.

[22] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[23] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, et al. 2010. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 92–105.

[24] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-speed query processing over high-speed networks. *Proceedings of the VLDB Endowment* 9, 4 (2015), 228–239.

[25] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. 2014. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1035–1046.

[26] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The end of an architectural era: (It's time for a complete rewrite). In *Proceedings of VLDB*. 1150–1160.

[27] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB main memory DBMS. *IEEE Data Engineering Bulletin* 36, 2 (2013), 21–27.

[28] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, et al. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. In *Proceedings of VLDB*. 245–256.

[29] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of SIGMOD*. 1–12.

[30] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of SOSP*. 18–32.

[31] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. 2012. LogBase: A scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment* 5, 10 (2012), 1004–1015.

[32] VoltDB Inc. 2017. VoltDB. Retrieved April 4, 2019 from https://www.voltdb.com/.

[33] Gottfried Vossen. 1995. Database transaction models. In *Computer Science Today*. Springer, 560–574.

[34] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling multicore databases via constrained parallel execution. In *Proceedings of SIGMOD*. ACM, New York, NY, 1643–1658.

[35] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of EuroSys*. 26:1–26:15.

[36] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, et al. 2017. vCorfu: A cloud-scale object store on a shared log. In *Proceedings of USENIX NSDI*. 35–49.

[37] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. 2017. Replication-driven live reconfiguration for fast distributed transaction processing. In *Proceedings of USENIX ATC*. 335–347.

[38] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of SOSP*. ACM, New York, NY, 87–104.

[39] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, et al. 2002. An integrated experimental environment for distributed systems and networks. In *Proceedings of OSDI*. 255–270.

[40] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. 2016. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of SIGMOD*. ACM, New York, NY, 1689–1704.

[41] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. In *Proceedings of VLDB*. 444–455.

[42] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of USENIX HotCloud.*

Received January 2019; accepted March 2019