Solar: Towards a Shared-Everything Database on Distributed Log-Structured Storage

Tao Zhu[1], Zhuoyue Zhao[2], Feifei Li[2], Weining Qian[1], Aoying Zhou[1], Dong Xie[2]
Ryan Stutsman[2], Haining Li[3], Huiqi Hu[1,3]

[1]*East China Normal University* [2]*University of Utah* [3]*Bank of Communications*

Abstract

Efficient transaction processing over large databases is a key requirement for many mission-critical applications. Though modern databases have achieved good performance through horizontal partitioning, their performance deteriorates when cross-partition distributed transactions have to be executed. This paper presents Solar, a distributed relational database system that has been successfully tested at a large commercial bank. The key features of Solar 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; 3) fine-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, Solar outperforms the existing shared-nothing systems by up to 50x when there are close to or more than 5% distributed transactions.

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 them, an essential requirement for large databases, due to the distributed data storage. For example, Bigtable [5] only supports single-row transactions, while 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 of efficient synchronization among nodes to ensure ACID properties and maintain good performance. Despite the significant progress and success achieved by many recently proposed systems [12, 27, 29, 19, 9, 23, 8, 37, 33], they still have various limitations. For example, the systems relying on shared-nothing architecture and 2PC (two-phase commit) heavily suffer from cross-partition distributed transactions, and thus require careful data partitioning with respect to given workloads. On the other hand, 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 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 a 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 apriori 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 towards the underlying workloads as new apps emerge constantly. On the other hand, the bank has a strong interest towards better utilization of its *existing* hardware infrastructures to avoid costly new hardware investment if possible.

With that in mind, Solar 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 Solar 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 process-

ing and temporarily stores committed updates through its *in-memory committed list*. Periodically, *recently committed data items* on T-node are merged back into S-nodes through a *data compaction* procedure running in the background, *without* interrupting ongoing transactions. Overall, Solar 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 T-node and the disk storage from all S-nodes *collectively form a distributed two-layer log-structured merge-tree* design [22]. Furthermore, a processing layer called *P-unit* is introduced to carry out *both* data access from 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. Towards realizing the above 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 OCC (optimistic concurrency control) and the MVCC (multi-version concurrency control) schemes.
- A data compaction algorithm, as part of MVOCC, is designed to efficiently merge the *committed list* on T-node back to S-nodes periodically, *without interrupting* transaction processing on T-node.
- Several optimizations are investigated to improve the overall performance, e.g., separation of computation and storage through P-units, grouping multiple data access operations in one transaction, 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, Solar outperforms existing shared-nothing systems by 50x when the transactions requiring distributed commits are close to or more than 5%.

2 Solar Architecture

Solar 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, Solar 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 of the whole database, writing a few records at a time in a database of terabytes of data. Thus, Solar 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-grained data access control. In Solar, processing nodes directly access data stored in remote nodes via 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.

Solar 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 kernel. It also consumes more CPU than Infiniband, where data transport is offloaded onto NIC. To address the issue, we designed fine-grained data access to reduce network overhead, including caching, avoiding unnecessary reads and optimizing inter-node communication via transaction compilation. Fine-grained data access brings the transaction latency on par with the state-of-the-art systems and improves throughput by 3x.

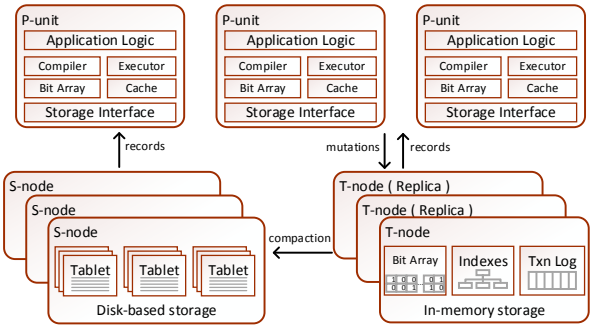

Figure 1: Architecture of Solar

2.2 Architecture overview

Figure 1 provides an overview of Solar's architecture. Solar separates transaction processing into *computation, validation and commit* phases using a *multi-version optimistic concurrency control 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 T-node or 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 T-node for validation and commit. Once T-node completes the validation, it writes the updates to its in-memory storage, and also a Write-Ahead Log to ensure durability. Finally, 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 S-nodes or at client side). They offload most of the computation burden from T-node so that 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.

Solar adopts a two-layer distributed storage that mimics the *log-structured merge tree (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 (256 MB in size by default), which is essentially a B-tree index. The committed updates are stored in Memtable on 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's 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 certain memory threshold or some scheduled off-peak time (e.g. 12:00 am - 4:00 am for Bank of Communications), Solar performs a data compaction operation to merge the updates in Memtable into SSTable to free up the memory on 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 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 T-node is twofold: it works as a transaction manager that performs timestamp assignment, transaction validation as well as committing updates; on the other hand, 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 T-node's CPU resource.

Finally, Solar uses data replication to provide high availability and resistance to node failures. In SSTable, each tablet has at least 3 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

Solar utilizes both Optimistic Concurrency Control (OCC) and Multi-Version Concurrency Control (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 paper focuses on Solar's support for snapshot isolation, and 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 T-node before transaction commits (i.e., write-ahead logging).

3.1 Supporting snapshot isolation

Solar implements snapshot isolation through combining OCC with MVCC [15, 2]. More specifically, MVOCC is used by 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 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 *compaction-timestamp*.

T-node uses a global, monotonically increasing, counter to allocate timestamps for transactions. Transaction processing in Solar 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 T-node and S-nodes. A transaction t_x obtains its *read-timestamp* (rt_x for short) when it first communicates with 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 T-node. T-node would then validate and commit the transaction.

Validating. The validation phase is conducted on T-node, which aims to identify potential write-write conflicts between t_x and other transactions. During the validation phase, 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, T-node guarantees that t_x has no write-write conflict and can continue to commit. Otherwise, T-node will abort t_x due to the lost update anomaly. Hence, after validation, T-node determines whether to commit or abort a transaction t_x. If it decides to abort t_x, T-node sends the abort decision back to the P-unit who 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, T-node obtains a *commit-timestamp* for t_x by incrementing the global counter. 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). Lastly, 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), Solar guarantees that t_x reads a consistent snapshot of the database and there is no lost update anomaly.

Consistent snapshot read: Firstly, 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. Secondly, 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: 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 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 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 T-node without communicating with the failed P-unit. 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 T-node. 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 T-node fails, it is able to recover committed data by replaying the redo log. Moreover, to avoid being the single point of failure, Solar also synchronizes all redo log records to two replicas of T-node using a primary-backup scheme. Each replica catches up the content of 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, Solar is able to recover from T-node failure and resume services in just a few seconds.

Failure of 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 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 T-node into the 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 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, 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. T-node waits till all such transactions have committed (i.e., no more transaction will update m_0 any more). At this point, 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 acknowledges 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. T-node is now allowed to discard m_0 and truncate the associated log records.

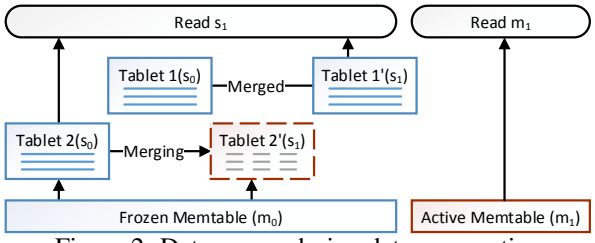

Figure 2: Data access during data compaction.

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 both that tablet from s_o and m_0.

Concurrency control. Snapshot isolation needs to be upheld during data compaction. The following concurrency control scheme is enforced. 1) 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. 2) 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. 3) The data compaction can actually be started once all transactions with a commit-timestamp smaller than t_{dc} finish. 4) 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. 5) If a transaction acquires a read-timestamp which 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 above procedure also prevents 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, 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 timestamp greater than its read timestamp rt_x. Hence, 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 while 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, since 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 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 a CSLE is encountered, it creates a new Memtable m_1 and replays subsequent log entries into m_1. The merging into 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 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. 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 Solar to reduce the network communication overhead among P-units, S-nodes and T-node. To achieve better performance, we design fine-grained data access methods between 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 S-nodes or in Memtable on T-node. Thus, Solar does not know where a record (or columns of a record) should be read from, and P-units have to access both SSTable on S-nodes and Memtable on T-node to ensure read consistency (though one of which will turn out to be an incorrect version).

Here, we first present an SSTable cache on P-units to reduce data access between P-units and S-nodes. Then, an asynchronous bit array is designed to help P-units identify potentially useless data accesses to 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, the P-unit pulls the record from an S-node and adds 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, Solar does not persist the cache pools. Entries in a cache pool do expire when the SSTable they were fetched from 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 a T-node would fetch nothing from T-node. We call this phenomenon *empty read*. These requests are useless and have negative effects. They increase latency and consume T-node's resources.

To avoid making many empty reads, 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. That is to say, 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, e.g., to encode the record-level information, but that would increase the size of the bit-array dramatically.

Solar keeps two types of bit arrays. The first type is a real-time bit array on 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 T-node.

On 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 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 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 T-node; otherwise, p will send a request to pull data from 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 T-node. Consider in tablet T, 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 T-node. In fact, the above 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 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 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

Solar supports JDBC/ODBC connections, as well as 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 [33, 39, 40]. Solar designs a compilation technique to optimize inter-nodes communication by generating an optimized 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

Table 1: Possible operations in a physical plan.

Execution graph. 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, such as branch or loop, can be viewed as a compound operation). Reads are implemented via RPC while write and process/computation are *local operations*. Hence, reads are the key to optimizing network communication.

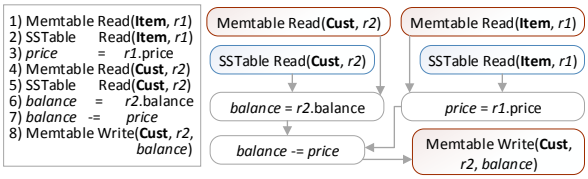

Figure 3: Example of operation sequence and execution graph.

Two operations have to be executed in order if they have 1) *procedure constraint*: two operations contain data/control dependence [21]; or 2) *access constraint*: two operations access the same record and one operation is a write. In practice, we cannot always determine two database records are the same during compilation, so we treat it as a potential access constraint if two operations are accessing the same table. 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 since 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 T-node, we can group multiple Memtable reads together in one RPC to T-node if they do not have constraints between them. This is done in two passes over the physical plan. The first pass finds all the Memtable reads not constrained by any other reads via a BFS over

the execution graph. The second pass starts from the unconstrained Memtable reads and marks all *local operations* that precede them. Before executing transaction logics, all 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 T-node.

Pre-executing SSTable **Reads.** SSTable reads can be issued even before a transaction obtains its read-timestamp from T-node and we concurrently execute them with other operations, since there is only one valid snapshot in SSTable at a time. This requires that the 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 update to the same record in Memtable. Though 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 reducing wait time and thus reducing latency. The SSTable reads that can be pre-executed can be found using a similar algorithm to the one that finds Memtable reads that can be grouped.

Remarks. The optimizations described in this section are desgined for short transactions. Other workloads, such as bulk loading, OLAP, require additional optimizations. For bulk loading, it is possible to skip the T-node and directly load data into S-nodes. For OLAP queries, they can be executed upon a consistent database snapshot, and some relational operators can be pushed down into storage nodes to reduce inter-node data exchange.

5 Experiment

We implemented Solar by extending the open-sourced version of Oceanbase (OB) [1]. In total, 58,281 lines were added or modified on its code base. Hence, Solar is a full-fledged database system, implemented in 457,206 lines of C++ code. In order to compare it to other systems that require advanced networking infrastructures, we conducted all experiments using 11 servers on Emulab [38], which allows configuring different network topologies and infrastructures. Each server has two 2.4 GHz 8-Core E5-2630 processors (32 threads in total when hyper-threading enabled) and 64GB DRAM, *connected through a 1 Gigabits Ethernet by default*. *By default ten servers* are used to deploy the database system. One server is used to simulate clients. We compared Solar with MySQL-Cluster 5.6, Tell (shared-everything) [19], and VoltDB Enterprise 6.6 (shared-nothing) [27]. Though 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 1-Gigabits (10-Gigabits) network respectively.

Solar is not compared with 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 durable logging, disaster recovery and a SQL engine. These features often introduce significant performance overhead, but are ignored by these lightweight system prototypes.

Solar 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 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 on each server. VoltDB creates 27 partitions on each server, which is based on the officially recommended strategy [32]. It 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 transaction processed per second (TPS). In each test instance, we adjusted the number of clients to get the best throughput.

5.1 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, Solar stores 1.6 million records (2.5 GB) in the Memtable and 100 million records (42GB) in the SSTable (with 3x replication enabled). After the benchmark finishes, there are 11 GB data in the Memtable and the size of SSTable is about 655 GB.

Figure 4 shows the performance of different systems when we vary the number of warehouses. Solar achieves about 53k TPS on 50 warehouse, and increases to about 75k TPS with 350 warehouses. When more warehouses are populated, there are less access contention in the workload, leading to fewer conflicts and higher concurrency. Solar 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 Solar, performs even worse than Tell-10G. VoltDB exhibits the worst performance due to distributed transactions. Lastly, Oceanbase is primarily designed for processing very short transactions and thus is inefficient on general transaction workloads. Solar always achieves at least 10x throughput improvement over Oceanbase. Therefore, we skip Oceanbase in other benchmarks.

Figure 5 evaluates the scalability when using different number of nodes. The throughputs of Solar, Tell and MySQL-Cluster increase with more nodes. In contrast, the throughput of VoltDB deteriorates for the following

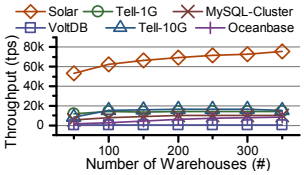

Figure 4: TPC-C: vary number of warehouses.

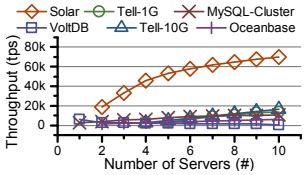

Figure 5: TPC-C: vary number of servers.

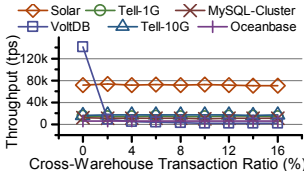

Figure 6: TPC-C: vary ratio of cross-warehouse transactions.

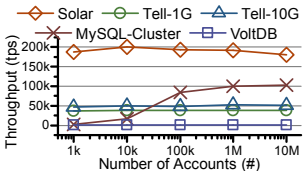

Figure 7: Smallbank: vary number of accounts.

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 request to be processed. The throughput growth in Solar slows down with more than 7 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 T-node receives more loads when working with more P-units, *and* in our experimental setting, T-node uses the same type of machine as that used for P-units. Hence, the overall performance increases sub-linear with the number of P-units. However, in the real deployment of Solar, a high-end server is recommended for T-node, whereas P-units (and 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 warehouse, it is a distributed transaction. VoltDB achieves the best performance (141k TPS) when there are no distributed transactions, which is about 2.0x of Solar (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.

Latency(ms)	Solar	Tell -1G	Tell -10G	MySQL-Cluster	VoltDB	OB
Payment	6	17	7	17	15619	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	2751	54

Table 2: 90th Latency, TPC-C workload.

Table 2 lists the 90th latency. Solar has a short latency for each transaction. Tell benefits from the better network. It gets better latency with the 10-Gbit network than the 1-Gbit one. 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. Though the 90th latency of NewOrder is small, its 95th latency reaches 15,819 ms.

5.2 Smallbank benchmark

Smallbank simulates a banking application. It contains 3 tables and 6 types of transactions. 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 (3 GB) in Memtable and 30M records (1.1 TB) in SSTable. After execution, Memtable has 5.2 GB data, and SSTable has about 1.1 TB data.

Figure 7 evaluates different systems by populating different number of accounts in the database. Note that *x-axis is shown in log-scale*. Solar has the best overall performance. Its throughput initially increases as the number of accounts increases, because 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, P-units need to issue remomte data access to S-nodes. As a result, its throughput slightly drops.

Latency(ms)	Solar	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

Table 3: 90th Latency, Smallbank workload.

Tell shows a fairly stable performance, but 10G Ethernet only improves its throughput slightly. MySQL Cluster also has a 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.

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

5.3 E-commerce benchmark

E-commerce is a workload from an e-business client of Bank of Communications. It includes 7 tables and 5 transaction types. There are two user roles in this application: buyer and seller. There are 4 tables for buyers:

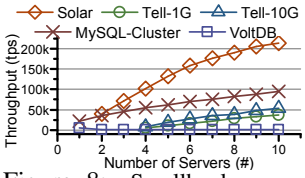

Figure 8: Smallbank: vary number of servers.

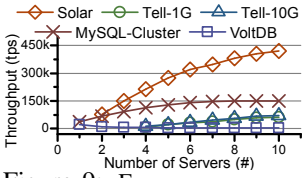

Figure 9: E-commerce: vary number of servers.

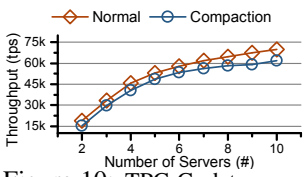

Figure 10: TPC-C: data compaction.

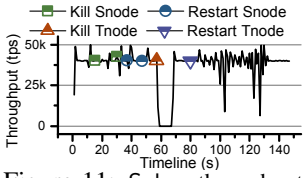

Figure 11: Solar: throughput under node failures.

User, *Cart*, *Favorite* and *Order* and 3 tables for sellers: *Seller*, *Item* and *Stock*. These tables are partitioned by *user_id* and *seller_id* respectively. At the start of the experiment, Solar has 11M records (5GB) in Memtable, and 25M records (815GB) in SSTable. When all experiments are completed, the Memtable has 8.6 GB 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 *r*ead-only transaction while the others are *r*ead-write ones. 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*.

Latency(ms)	Solar	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

Table 4: 90th Latency, E-commerce workload.

Figure 9 shows the performance of each system using different number of servers. The throughput of Solar 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 completed within 1ms by Solar. MySQL-Cluster and Tell also see performance improvement when more servers are used. But 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 which block other single-partition transactions. As a result, OnClick and AddCart also have longer latency.

5.4 Data compaction

During transaction processing, Solar may initiate a data compaction in the background. Figure 10 shows the impact of data compaction on the performance, when Solar is processing the standard TPC-C workload. As shown in Figure 10, data compaction has little negative effect on the performance when 5 or less servers are used. It is because the performance is mainly limited by the number of P-units in these cases, and compaction would not influence the operation of P-units. When more servers are used, there is about 10% throughput loss. This is because at this point 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 T-node.

5.5 Node failures

We next investigate the impact of node failures in Solar. In this experiment, 3 servers were used to deploy T-nodes, and 7 servers were used to deploy S-nodes and 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 2 S-nodes does not impact the performance, as the SSTable keeps 3 replicas for each tablet and each P-unit also caches data from SSTable. Thus, losing 2 S-nodes does not influence performance. We then terminated the primary T-node. Immediately after it went 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 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.

5.6 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 the similar performance trends. With more P-units and S-nodes deployed in system, the individual optimizations show different trends of improvement. The effectiveness of SSTable cache drops because the overall data access throughput increases when more S-nodes are deployed. However, the accesses to T-node are more contentious as more P-units communicate with the single T-node. With transaction compilation enabled, small data accesses to T-node are

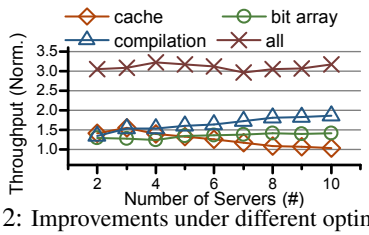

Figure 12: Improvements under different optimizations.

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 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 T-node. When all optimizations are used together, they bring about 3x throughput improvement regardless of the number of servers used.

6 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], DBX [34], and others. 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, Solar uses MVOCC and a cluster of data storage, and it can potentially skip 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], E-Store [28]. We have 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 set. Oceanbase [1] is Alibaba's distributed shared-nothing database designed for short transactions. In shared-nothing systems, locking happens at partition level. To get subpartition 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 apriori, 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 [37] and DrTM+B [36] (with live reconfiguration and data repartitioning), HANA SOE [10]. Solar, on the other hand, uses commodity servers and does not rely on special hardwares.

Log-structured storage. The log-structured merge tree [22] is optimized for insertion, update and deletion. It is widely adopted by many NoSQL vendors, such as LevelDB [18], BigTable [5], Cassandra [16] and etc. 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 to support fast system recovery, rather than optimizing OLTP workloads. Hyder II optimizes OCC for tree-structured, log-structured databases [3] which Solar may leverage for further improving its concurrency control scheme. vCorfu [35] implements materialized streams on a shared log to support fast random reads. But, it increases transaction latency as committing requires at least four network roundtrips.

7 Conclusion

This work presents Solar, 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 Solar compared to other systems on different workloads. Solar has been deployed at Bank of Communications to handle its e-commerce OLTP workloads. We plan to open source Solar 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.

Acknowledgments. Tao Zhu, Weining Qian and Aoying Zhou are supported by 863 Program (2015AA015307), National Key R&D Plan Project (2018YFB1003303), NSFC (61432006 and 61332006). Feifei Li, Zhuoyue Zhao and Dong Xie are supported in part by NSF grants 1619287 and 1443046. Feifei Li is also supported in part by NSFC grant 61729202. Ryan Stutsman is 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. The authors greatly appreciate the valueable feedback provided by USENIX ATC reviewers.

References

[1] ALIBABA OCEANBASE. Oceanbase. `https://github.com/alibaba/oceanbase`, 2015.

[2] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ANSI SQL isolation levels. In *SIGMOD* (1995), vol. 24, ACM, pp. 1–10.

[3] BERNSTEIN, P. A., DAS, S., DING, B., AND PILMAN, M. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *SIGMOD* (2015), pp. 1295–1309.

[4] CHANDRASEKARAN, S., AND BAMFORD, R. Shared cache-the future of parallel databases. In *ICDE* (2003), IEEE, pp. 840–850.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *TOCS 26*, 2 (2008), 4.

[6] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007), vol. 41, ACM, pp. 205–220.

[7] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD* (2013), ACM, pp. 1243–1254.

[8] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *NSDI* (2014), pp. 401–414.

[9] DRAGOJEVIC, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP* (2015), pp. 54–70.

[10] GOEL, A. K., POUND, J., AUCH, N., BUMBULIS, P., MACLEAN, S., FÄRBER, F., GROPENGIESSER, F., MATHIS, C., BODNER, T., AND LEHNER, W. Towards scalable real-time analytics: an architecture for scale-out of OLxP workloads. *PVLDB 8*, 12 (2015), 1716–1727.

[11] JOSTEN, J. W., MOHAN, C., NARANG, I., AND TENG, J. Z. DB2's use of the coupling facility for data sharing. *IBM Systems Journal 36*, 2 (1997), 327–351.

[12] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P., MADDEN, S., STONEBRAKER, M., ZHANG, Y., ET AL. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB 1*, 2 (2008), 1496–1499.

[13] KEMPER, A., AND NEUMANN, T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE* (2011), IEEE, pp. 195–206.

[14] KENNEDY, K., AND MCKINLEY, K. S. *Maximizing loop parallelism and improving data locality via loop fusion and distribution*. Springer, 1993.

[15] KUNG, H.-T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *TODS 6*, 2 (1981), 213–226.

[16] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *SIGOPS 44*, 2 (2010), 35–40.

[17] LEVANDOSKI, J., LOMET, D., SENGUPTA, S., STUTSMAN, R., AND WANG, R. High performance transactions in Deuteronomy. Conference on Innovative Data Systems Research (CIDR 2015).

[18] LEVELDB. LevelDB. `http://leveldb.org/`, 2017.

[19] LOESING, S., PILMAN, M., ETTER, T., AND KOSSMANN, D. On the design and scalability of distributed shared-data databases. In *SIGMOD* (2015), ACM, pp. 663–676.

[20] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *OSDI* (2014), pp. 479–494.

[21] MUCHNICK, S. S. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[22] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[23] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., ET AL. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS 43*, 4 (2010), 92–105.

[24] RÖDIGER, W., MÜHLBAUER, T., KEMPER, A., AND NEUMANN, T. High-speed query processing over high-speed networks. *PVLDB 9*, 4 (2015), 228–239.

[25] SERAFINI, M., MANSOUR, E., ABOULNAGA, A., SALEM, K., RAFIQ, T., AND MINHAS, U. F. Accordion: elastic scalability for database systems supporting distributed transactions. *PVLDB 7*, 12 (2014), 1035–1046.

[26] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era:(it's time for a complete rewrite). In *PVLDB* (2007), VLDB Endowment, pp. 1150–1160.

[27] STONEBRAKER, M., AND WEISBERG, A. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull. 36*, 2 (2013), 21–27.

[28] TAFT, R., MANSOUR, E., SERAFINI, M., DUGGAN, J., ELMORE, A. J., ABOULNAGA, A., PAVLO, A., AND STONEBRAKER, M. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *PVLDB* (2014), 245–256.

[29] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD* (2012), pp. 1–12.

[30] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *SOSP* (2013), pp. 18–32.

[31] VO, H. T., WANG, S., AGRAWAL, D., CHEN, G., AND OOI, B. C. Logbase: A scalable log-structured database system in the cloud. *PVLDB 5*, 10 (2012), 1004–1015.

[32] VOLTDB INC. VoltDB. `https://www.voltdb.com/`, 2017.

[33] WANG, Z., MU, S., CUI, Y., YI, H., CHEN, H., AND LI, J. Scaling multicore databases via constrained parallel execution. In *SIGMOD* (2016), ACM, pp. 1643–1658.

[34] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *EuroSys* (2014), pp. 26:1–26:15.

[35] WEI, M., TAI, A., ROSSBACH, C. J., ABRAHAM, I., MUNSHED, M., DHAWAN, M., STABILE, J., WIEDER, U., FRITCHIE, S., SWANSON, S., FREEDMAN, M. J., AND MALKHI, D. vCorfu: A cloud-scale object store on a shared log. In *USENIX NSDI* (2017), pp. 35–49.

[36] WEI, X., SHEN, S., CHEN, R., AND CHEN, H. Replication-driven live reconfiguration for fast distributed transaction processing. In *USENIX ATC* (2017), pp. 335–347.

[37] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *SOSP* (2015), ACM, pp. 87–104.

[38] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *OSDI* (2002), pp. 255–270.

[39] WU, Y., CHAN, C.-Y., AND TAN, K.-L. Transaction healing: Scaling optimistic concurrency control on multicores. In *SIGMOD* (2016), ACM, pp. 1689–1704.

[40] YAN, C., AND CHEUNG, A. Leveraging lock contention to improve OLTP application performance. *PVLDB* (2016), 444–455.