

# Lock Violation for Fault-tolerant Distributed Database System

Hua Guo

School of Information  
Renmin University of China  
Beijing, China  
guohua2016@ruc.edu.cn

Xuan Zhou

School of Data Science And Engineering  
East China Normal University  
Shanghai, China  
xzhou@dase.ecnu.edu.cn

**Abstract**—Modern distributed database systems scale horizontally by sharding its data on a large number of nodes. To achieve fault tolerance, most of such systems build their transactional layers on a replication layer, which employs a consensus protocol to ensure data consistency. Synchronization among replicated state machines thus becomes a major overhead of transaction processing. Without a careful design, synchronization could amplify transactions’ lock duration and impair the system’s scalability. Speculative techniques, such as Controlled Lock Violation (CLV) and Early Lock Release (ELR), prove to be effective in shortening lock duration and boosting performance of transaction processing. An intuitive idea is to use these techniques to optimize geo-replicated distributed databases. In this paper, we show that direct application of speculation is often unhelpful in a distributed environment. Instead, we introduce Distributed Lock Violation (DLV), a speculative technique for geo-replicated distributed databases. DLV minimize the cost of to conduct lock violation, so that it can achieve good performance without incurring severe side effects.

**Index Terms**—Database System, Distributed Transaction, Locking, High Availability

## I. INTRODUCTION

Modern distributed database system scale out by partitioning data across multiple nodes, so that it can run transactions on multiple servers in parallel to increase throughput. When a transaction needs to access multiple partitions, it has to employ a coordination protocol to ensure atomicity. It is commonly known that such distributed transactions can lead to significant performance degradation. This is mainly due to the following reasons [1]:

1. Coordinated commit requires a chatty protocol (e.g., Two-Phase Commit), which introduces tremendous network overheads;
2. Message transmission overlaps with the critical path of transaction commit, which worsens the contention among transactions.

These issues can be more serious for geo-replicated databases, which face increased network overheads. The replication layer of a geo-replicated database often uses a Paxos-like consensus protocol to ensure consistency among replicas. This introduces a significant amount of message transmission, which further increase the duration of transactions. To facilitate replication, we usually split data into small chunks

and replicate each chunk independently. As a result, cross-chunk transactions (rather than cross-partition transactions) all become distributed transactions. This makes distributed transaction even more inevitable.

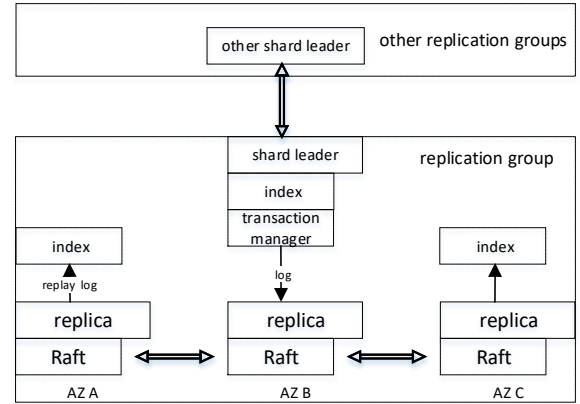


Fig. 1: Typical architecture of geo-replicated distributed DBMS

Fig. 1 presents the typical architecture of geo-replicated distributed databases. The database partitions its data into a number chunks. For each chunk, the replication layer replicates its data to several Availability Zones (AZ) [2]. Among the available zones, the replication layer employ a consensus protocol to shield data consistency.

As previous work has discussed [1] [3] [4], this architecture has to rely on a chatty protocol (which integrates the commit protocol and the consensus protocol) to ensure the correctness of transactions. It may fail to scale when confronted with highly contended workload. Nevertheless, this architecture supports a wide range of transaction processing methods. Most industrial data systems choose this two-layer architecture, including Google Spanner [5] [6], NuoDB [7], CockroachDB [8], TiDB [9], etc.

Looking more closely, the main issue is that the commit and consensus protocols enlarge the timespan of the critical paths in transaction processing. This significantly amplifies the overhead of contention. Fig. 2 shows the message flow of a

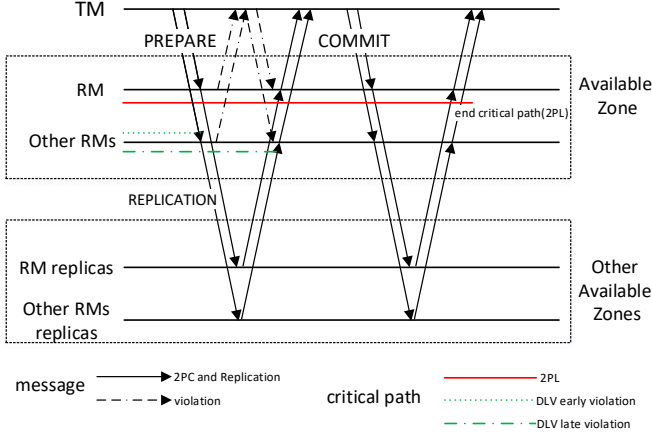


Fig. 2: The message flow and lock holding critical path when the DBMS uses S2PL for concurrency control and 2PC as the commit protocol. The dash arrow lines represents the messages introduced by Distributed Lock Violation (DLV). The red lines represent the critical paths of S2PL. The green dash lines represent the critical paths of DLV.

distributed transaction in such an architecture. We assume that it uses S2PL for concurrency control and 2PC as the commit protocol, and the replication layer is deployed over a WAN. When a transaction requests to commit, the *TM* (transaction manager) issues a ‘prepare’ message to each *RM* (resource manager). Then, each *RM* replicates its vote (‘prepare commit’ or ‘prepare abort’) to the corresponding replicas through a consensus protocol, before it sends its vote to *TM*. After the *TM* collects all the votes of the *RMs*, it broadcasts the final decision (‘commit’ or ‘abort’) to all the *RMs*<sup>1</sup>. Once a *RM* receives the final decision from the *TM*, it replicates the decision to the corresponding replicas. After the consensus of commit or abort is reached, the *RM* can release the locks that it had retained over the accessed data.

We depict lock duration by red lines in Fig. 2. As we can see, the lock duration span multiple message round trips, including those over the WAN. This will severely impair the concurrency of transaction processing in face of a high degree of contention.

Speculative techniques, such as Early Lock Release (ELR) [10] and Controlled Lock Violation (CLV) [11], prove to be effective in optimizing centralized transaction processing. These techniques can be extended to a distributed environment.

Speculative techniques improve concurrency by excluding logging from lock duration, Transactions on geo-replicated distributed databases require much more time to ship and persist logs. Fig. 3 shows the WAN RTT, LAN RTT and disk seek approximate time costs comparison. This provides more opportunities for exploiting speculation. However, application of speculation in a distributed environment is complex. A number of design choices need to be considered. For instance,

<sup>1</sup>Depending on variants of implementations, the *TM* can choose to persist its decision on its log or not.

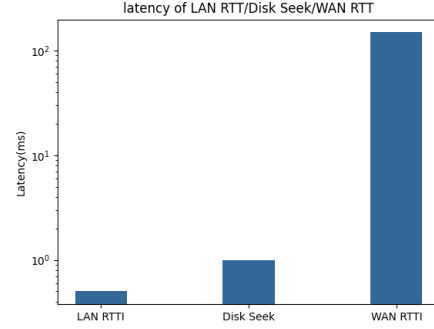


Fig. 3: approximate latency comparison [12]

when working with 2PC, we need to decide to violate (or release) lock at which phase. As previous work showed [11], lock violation at the first phase may enable a higher degree of concurrency. However, dependency tracing and cascade abort may incur excessive overheads. On the contrary, lock violation at the second phase costs less in dependency tracing and cascade abort, while it has to sacrifice a certain amount of concurrency. On the other hand, not all transactions can benefit from CLV or ELR, e.g., workload with little conflict. It unnecessary and even harmful to apply speculation to all cases.

In this paper, we propose a technique called Distributed Lock Violation (DLV) to enhance the performance of geo-replicated distributed DBMS. DLV performance lock violation and minimize the cascade abort rate and dependency tracing cost. It introduces much less penalty compared to previous implementation [11].

The remainder of this paper is organized as follows. Section II reviews related work. Section III shows that a strict schedule is not necessary and hurt the performance of distributed DBMS. Section IV introduces DLV. Section V evaluates DLV and compare it against previous work. Section VI concludes this paper.

## II. RELATE WORK

### A. Transaction Processing on Replicated Databases

Most replicated databases rely on state-machine replication(SMR) to realize high availability. SMR often uses a consensus protocol to synchronize different replicas. Synchronization introduces significant amount of network traffic and becomes a major overhead of replicated database systems. Paxos [13] [14] is the most well known consensus protocol. To reach a consensus on a single data update in Paxos, it costs two message round trips, one for choosing a proposal and another for proposing the entry. Multidecree Paxos [15] elects a leader as the only proposer to eliminate the first message round trip. Raft [16] is a similar consensus protocol to Paxos. As it is more understandable, it is widely used in modern database systems. In Raft, it costs at least one message round trip to reach a consensus. Google Spanner [5] [6], NuoDB [7], CockroachDB [8] and TiDB [9] are all geo-replicated DBMSes

built upon Paxos or Raft. They all face heavy costs incurred by data synchronization.

To minimize the cost of synchronization, a number of techniques have also been proposed. VoltDB [17] and Calvin [1] employ a deterministic transaction model to reduce the coordination cost of distributed transactions. Deterministic scheduling enables active replication, which allows transactions to kickoff synchronization at the earliest possible time. Tapir [3] relaxes the consistency requirements of the replication layer, so that it can reduce the message round trips to reach consensus. Janus [4] aims to minimize wide-area message round trips by consolidating the concurrency control mechanism and the consensus protocol. It uses deterministic serializable graph tracing to ensure atomicity of transactions. In a nutshell, Tapir and Janus both co-designed the transaction and replication layers of distributed databases, so that they only need to incur one wide-area message round trip to commit a transaction. VoltDB, Calvin, Tapir and Janus all impose strict constraints on the implementation of the transaction layer, making them incompatible with existing systems, such as Spanner and CockroachDB. In contrast, this work focuses on the optimization opportunities in the general implementation of geo-replicated databases (as depicted in Fig. 1).

### B. Optimization on Locking based Concurrency Control

Two-phase locking (2PL) is the most widely used concurrency control mechanism. As a pessimistic method, 2PL assumes a high likelihood of transaction confliction. It uses locks to enforce the order of conflicting transactions. Strict 2PL (S2PL) is a brute force implementation of 2PL. It requires a transaction to preserve all its lock until it ends. As S2PL can easily guarantee transactions' recoverability, many databases choose to use it. When extending S2PL to distributed databases, the lock holding time will be substantially enlarged, as the commit critical path will involve a number of message round trips.

All S2PL implementations adopt a certain approach to resolve deadlocks. In the *no-wait* [18] approach, a transaction immediately aborts if it fails to place a lock. Previous works showed that this is a scalable approach in a distributed environment [19] [18], as it eliminates blocking completely. However, it works poorly when dealing with workload of high contention. Another approach is *wait-die* [20]. It avoids some false-positive aborts encountered by *no-wait* by utilizing timestamps. The *Deadlock detection* approach [21] detects deadlock by explicitly tracing wait-for relationship among transactions. Many centralized database systems [22] [23] adopt this approach, as can deal with contention better. However, deadlock detection in a distributed environment is highly costly, making it the least favorable approach in our case.

To optimize the performance of locking based concurrency control, speculation can be used. Early lock release (ELR) [24] [25] [26] [10] [27] is a typical speculative technique. ELR allows a transaction to release its locks before its commit log is flushed to disk. It was first proposed by DeWitt et al. [24]. Soisalon-Soininen et al. [25] analyzed its correctness in

various settings. Many other works applied and evaluated ELR in different types of systems [26] [10] [10] [26]. However, previous works on ELR were limited to centralized database systems.

Control lock violation (CLV) [11] is a more general speculative technique than ELR. It allows certain transactions to ignore certain locks, instead of releasing a lock completely. CLV has been tested on distributed databases. The results show that it can optimize both phases of two-phase commit. CLV needs to trace dependency among transactions. In [11], the authors use a Register and Report (RARA) approach [28] to implement the tracer. RARA work well on a centralized database. In a distributed environment, dependency tracing becomes much more costly. Cascade abort is another side effect of speculation. It can also leads to severe performance downgrade for distributed databases.

## III. SCHEDULING WITH LOCK VIOLATION

In the following, we describe the basic idea of scheduling with lock violation.

### A. Preliminaries and Assumptions

Our scenario is a geo-replicated distributed database, which shards its data by primary keys. Each data chunk is replicated across a number of remotely located AZs. Physiological logs are transmitted among the AZs to keep the data synchronized. The physiological logs record the row-level write operations of each transaction. A consensus protocol is used to prevent the synchronization from going wrong. We assume that there is a replica leader for each data chunk, which is responsible for coordinating the synchronization process.

We assume that two types of distributed transactions can be conducted on the system. They are known as one-shot and interactive transactions. Fig. 4 shows the message flow during the commit phase of the two types of transactions. We can see that an interactive transaction costs more message round trips than a one-shot transaction, as the *TM* has to explicitly notify the *RM*s that they can prepare to commit.

### B. Concepts of Scheduling

Before presenting the idea of lock violation, we first review the conceptual model of transaction processing. Most of the contents can be found in previous work, such as [20].

1) *Transaction and History*: Suppose that a distributed database resides on  $n$  nodes, represented by  $R = \{r_1, r_2, \dots, r_n\}$ . A transaction  $T_i$  can run on any  $m$  of the  $n$  nodes ( $1 \leq m \leq n$ ), represented by  $S = \{s_1, s_2, \dots, s_m\} \subseteq R$ . The transaction  $T_i$  is composed of a series of operations, where each can be a read or a write or other command operation including abort, commit, etc. Let  $r_i[x]$  denote that  $T_i$  reads the record  $x$ . Let  $w_i[x]$  denote that  $T_i$  writes on the record  $x$ . Let  $c_i$ ,  $a_i$ ,  $p_i^c$ ,  $p_i^a$  denote the commit, abort, prepare-to-commit and prepare-to-abort operations of  $T_i$  respectively.

A transaction history is a collection  $H = \{h_1, h_2, \dots, h_n\}$ , in which each  $h_u$  is the local history of  $H$  on the node  $s_u$ , which is a sequence of operations issued by different transactions.

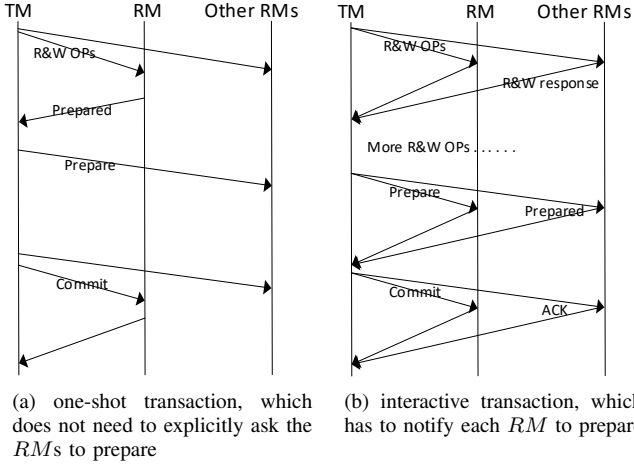


Fig. 4: Commit message flows for one-shot and interactive distributed transactions

2) *Deterministic and Non-deterministic Abort*: A number of reasons can cause a transaction to abort. In general, they can be categorized as:

1. User requested abort. These are aborts specified by the program logic (e.g., requiring a transaction to abort if it has accessed non-existent records).
2. Violation of isolation (e.g., serializability). These are aborts commanded by the database system.
3. Database node failure. For simplicity, we assume fail-stop failures only.

We call the first two types of abort *deterministic abort* and the last one *non-deterministic abort*. In a database system, deterministic aborts occur much more frequently than non-deterministic abort. Therefore, the prices we are willing to pay for the two kinds of abort are very different.

3) *Dependencies among Transaction*: There are three kinds of data dependency among transactions, known as *write-dependency*, *write-after-write* and *read-after-write*. In a local history  $h$ , if  $T_j$  reads  $T_i$ 's write on  $x$ , we call it a *write-after-read* (*wr*) dependency and denote it by  $w_i[x] \rightarrow r_j[x]$ . Analogically, if  $T_j$  overwrites  $T_i$ 's write on  $x$ , it is called a *write-after-write* (*ww*) dependency and denoted by  $w_i[x] \rightarrow w_j[x]$ . If  $T_i$  reads  $x$  before  $T_j$  writes on  $x$ , it is called a *read-after-write* (*rw*) dependency and denoted by  $r_i[x] \rightarrow w_j[x]$ .

Based on data dependencies, we can define *commit dependency*.  $T_j$  has a *commit dependency* on  $T_i$ , written as  $T_i \rightarrow T_j$ , if  $T_j$  cannot commit prior to  $T_i$ . In other words,  $T_i$  aborts,  $T_j$  has to abort too.  $T_j$  has a *commit dependency* on  $T_i$ , if and only if  $T_j$  has a *rw-dependency* on  $T_i$ .

4) *Relaxation on Strictness*: Most traditional database systems adopt Strict Two-Phase Locking (S2PL) [29] to ensure the recoverability of transactions. Strictness implies that a transaction cannot read a previous write by another transaction which has not committed yet.

Strictness is not a necessary condition for a correct schedule. Although it simplifies the implementation of a database system, it sacrifices the concurrency of transaction process-

ing. For instance, the schedule  $H_1$  of Fig. 5 is serializable and strict. (The data dependencies include  $r_1[x] \rightarrow w_3[x]$ ,  $w_1[x] \rightarrow r_2[y]$ ,  $r_2[y] \rightarrow w_3[x]$ , which are not cyclical.) In contrast,  $H_2$  in Fig. 6 is serializable but not strict. In Fig. 6, there are three records  $x, y, z$ , residing on  $s_1, s_2, s_3$  respectively.  $T_1$  writes  $y$  and  $x$ .  $T_2$  reads  $T_1$ 's write on  $x$  before  $T_1$  commits. Transaction  $T_3$  overwrites  $T_2$ 's write before  $T_2$  commits. As we can see, by violating strictness,  $H_2$  enables a higher degree of concurrency.

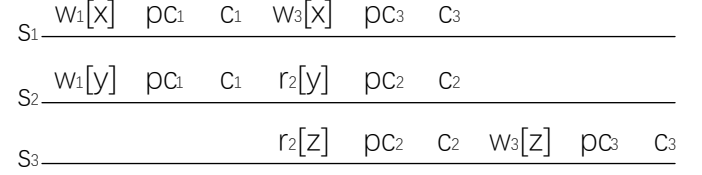


Fig. 5: A strict and serializable schedule  $H_1$

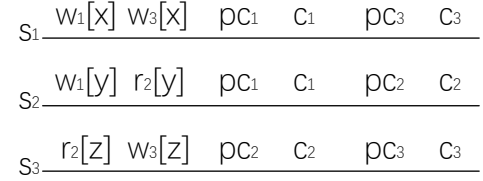


Fig. 6: A non-strict but serializable schedule  $H_2$

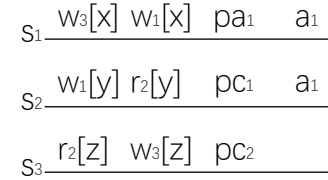


Fig. 7: Schedule  $H_3$ ,  $T_1$  abort due to non-serializable

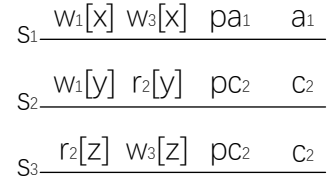


Fig. 8: Schedule  $H_4$ ,  $T_2$  commit ahead  $T_1$ , non-recoverable anomaly

The drawback of strictness is amplified in a geo-replicated and distributed database. The commit of a transaction in such a database usually involve a lot of work, including multiple rounds of communication across the WAN. As strictness requires a transaction to hold locks until it finishes committing, this means a substantially lengthened lock holding time. This will hurt the performance of distributed transactions severely. Our basic idea is to develop a serializable but non-strict scheduler to shorten the lock holding time.

Relaxation on strictness will, however, complicate the recovery mechanism. In a traditional recovery mechanism, such as ARIES, the dependencies among transactions completely comply with the persisting order of their logs. This guarantees the recoverability of scheduling, which requires that a transaction cannot commit before the transactions it has commit dependencies on. When strictness no longer holds, we need other tactics to enforce the commit order, which may incur additional overheads.

### C. Lock Violation and Dependency Tracing

Lock violation is a general technique to enable a non-strict scheduler. Its basic idea is to allow a transaction to speculatively access data locked by other transactions. If such speculative data accesses cause a problem, the system should be able to rectify it.

For lock violation to be correct, it should first preserve serializability. Considering the non-strict schedule  $H_3$  in Fig. 7, it contains three data dependencies:

$$w_3[x] \rightarrow_s w_1[x], w_1[y] \rightarrow_s r_2[y] \quad r_2[z] \rightarrow_s w_3[z]$$

This leads to a circle  $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$ , which makes the schedule non-serializable.  $H_3$  is not possible if we apply S2PL. It becomes possible when lock violation is permitted – we allow  $T_2$  to read from uncommitted  $T_1$ , and  $T_1$  to overwrite uncommitted  $T_3$ . Therefore, to preserve serializability, we need to prevent the data dependencies from forming a dependency cycle. Many techniques can be adopted to achieve this, including the *wait-die* strategy mentioned in the related work.

Secondly, a schedule generated by lock violation must be recoverable. In the schedule  $H_4$  in Fig. 8, due to lock violation,  $T_2$  reads  $T_1$ 's write and commits before  $T_1$ . In this case, if  $T_1$  later on decides to abort, it is no longer possible to reverse  $T_2$ . Therefore,  $H_4$  is not a recoverable schedule. To ensure recoverability, we need to trace the commit dependencies (i.e., *wr*-dependencies) among transactions, and force the commit order to comply with the commit dependencies. For instance, if we know that  $T_2$  depends on  $T_1$ , we can hold  $T_2$ 's commit request until  $T_1$  finishes. If  $T_1$  decides to abort, we abort  $T_2$  too. This is known as cascade abort.

In a geo-replicated distributed database, lock violation can potentially be allowed at two occasions in a transaction. The first occasion is when the work of the transaction has been completed successfully and the transaction is ready to commit. We call lock violation at this occasion *late lock violation*. It is only intended to reduce the contention in the commit phases of transactions. The second occasion is before the transaction is ready to commit. We call lock violation at this occasion *early lock violation*. The early lock violation can also fall into two categories: that which violation when an *RM* begins to decide its prepare decision, and violation when a transaction is even on-going in this *RM*.

While early lock violation allows us to harness more concurrency, it may cause additional issues. First, early lock violation can result in non-serializable schedules. This has been illustrated by  $H_3$  in Fig. 7. In contrast, *late lock violation* is safe. If

a transaction can only violate the locks of transactions that are ready to commit, it cannot result in cyclic data dependencies. Second, early lock violation faces a much higher chance of cascade abort. In late lock violation, only non-deterministic aborts can cause cascade abort, as no lock can be violated before deterministic aborts (since the transaction has already been permitted to commit). In early lock violation, all aborts can cause cascade abort.

Secondly, a schedule generated by lock violation must be recoverable. In the schedule  $H_4$  in Fig. 8, due to lock violation,  $T_2$  reads  $T_1$ 's write and commits before  $T_1$ . If  $T_1$  decides to abort, it is no longer possible to reverse  $T_2$ . As a result, schedule  $H_4$  is not recoverable. To ensure recoverability, we need to trace the *wr*-dependencies among transactions, to ensure that the commit order complies with the *wr*-dependencies.

Whether late or early lock violation is adopted, we need to trace the commit dependencies among transactions and make sure they commit in a right order. In a geo-replicated distributed database, the scheduler should follow the following rules to enforce the commit order. Given that there is a commit dependency from  $T_j$  to  $T_i$ ,

- 1)  $T_j$ 's *RM* can persist its 'prepare' state (including replication of the state) only if  $T_i$  has committed;
- 2)  $T_j$ 's *TM* can send out 'commit' request only if  $T_i$  has committed;
- 3) If  $T_i$  aborts,  $T_j$  must also abort too.

## IV. IMPLEMENTATION OF DLV

In this section, we present how to implement Distributed Lock Violation (DLV). The following contents would include: How DLV can avoid complex recovery algorithm and maintain the most limited amount of dependencies; Different time of enabling violation; The *wait-die* policy of DLV use; The pseudocode code description finally.

### A. No UNDO Log and Speculative Versions

As mentioned earlier, a non-strict scheduler may require a more complex recovery mechanism for a DB use undo log. For instance, consider the following non-strict schedule:

$$H_5 = w_1[x]w_2[x]a_1a_2$$

If transaction  $T_1$  aborts, it forces  $T_2$  to abort too. This is known as cascade abort. To conduct abort, traditional database systems use undo logs to cancel out a transaction's write operations. Undo becomes a bit tricky when non-strictness is allowed. For instance, let  $exp(H_5)$  be an extended schedule of  $H_5$  with undo operations, in which  $w_i^-[x]$  denotes that  $T_i$  undoes its write on  $x$ .

$$exp(H_5) = w_1[x]w_2[x]w_1^-[x]c_1w_2^-[x]c_2$$

To understand why this schedule is wrong and how it can occur, let us take an example. Suppose the initial value of records  $x$  and  $v$  of are both 0. The value of records  $x$  and its undo log formatted after executing every operations in  $exp(H)$  is shown in Table I. Firstly,  $T_1$  write  $x$  and create its undo log by  $x$ 's previous value. Later, transaction  $T_2$  violates  $T_1$ 's lock

on  $x$  after  $T_1$  write its value 1.  $T_2$  overwrite  $T_1$ 's write on  $x$  and construct its undo log by the value written by  $T_1$ . Transaction  $T_1$  abort due to some reason and undo its write. Then  $T_2$  also abort and cancel out its write by a wrong value of  $x$ , which is from  $T_1$ 's write. Finally, after the execution of this schedule, both transaction  $T_1$  and  $T_2$  abort. The value of  $x$  is 1, but the correct result should be the initial value 0.

| operations   | x | undo  |
|--------------|---|-------|
| $w_1[x=1]$   | 1 | $x=0$ |
| $w_2[x=2]$   | 2 | $x=1$ |
| $w_1^-[x=0]$ | 0 |       |
| $c_1$        | 0 |       |
| $w_2^-[x=1]$ | 1 |       |
| $c_2$        | 1 |       |

TABLE I:  $x, y$  values, undo log after the execution of  $exp(H_5)$

In other words, when performing cascade abort, we cannot abort each transaction independently. Instead, we have to follow the reserve order of the write operations to perform the recovery. For schedule  $H_5$ , a correct recovery explanation may be :

$$exp(H_5) = w_1[x]w_2[x]w_2^-[x]c_2w_1^-[x]c_1$$

This problem is intrinsic to non-strictness. Even if we choose to perform *late lock violation* only, which does not permit lock violation before deterministic abort, cascade abort can still be triggered by non-deterministic abort. To tackle this problem, we need a more complex recovery algorithm, such as SOT [30]. To reduce the complexity, DLV maintains uncommitted data versions only in memory and applies the no-steal policy to data persistence. The no-steal policy forbids the system from writing uncommitted data to permanent storages. As a drawback, it may consume more memory space. However, this is not a big concern, considering that today's database servers are usually equipped with large RAM. Based on this design, we do not need undo log anymore. If a transaction aborts, we only need to discard its uncommitted data in memory. The no-steal policy has another advantage in geo-replicated databases. That is, it pushes data synchronization among replicas to the commit phase, so that through lock violation we can exclude data synchronization completely from the lock holding time.

The version list structures from the newest version to the oldest version and the last version of this list is the committed version. Since speculation versions are uncommitted data, they are always stored in main memory and needs no persistence. And in permanent storage, there is the only single version, the committed version. When a transaction commits, it cooperatively cleans its outdated version from the version list. If a transaction would abort, it just needs to remove its write versions from the version list.

Previously, we have discussed that a  $ww$  dependency does not affect recoverability. *Late lock violation* can ignore  $ww$  and  $rw$  dependencies and only trace  $wr$  dependencies for recoverability since it has promised serializability Fig. 9 shows a series of access on two rows,  $x, y$ . The green rectangles are

speculative versions. The red ones are committed versions. Although there is  $ww$  dependency  $w_6[x] \rightarrow w_4[x]$ . The abort of  $T_4$  does not cause  $T_6$  cascade abort.

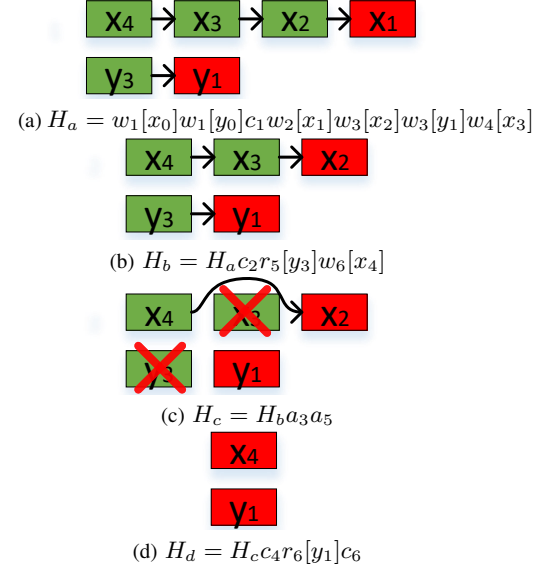


Fig. 9: speculative(green) and committed(red) versions,  $x_i$  expresses this version is from transaction  $T_i$ 's write. So,  $w_j[x_i]$  express transaction  $T_j$  overwrite  $T_i$ 's write.

## B. Lock Violation Occasion

In a distributed transaction,  $TM$  assigns work to  $RM$ . Once the work is done,  $TM$  applies 2PC to end the transaction. In 2PC, both  $TM$  and  $RM$  need to synchronize with their replicas, to make sure that node failures will not corrupt the data. Synchronization usually takes a long time, especially when the replicas are located remotely. DLV aims to prevent data synchronization from coinciding with the lock holding time.

In *late lock violation*, a transaction allows its lock to be violated only after all  $RM$ s confirm that they can commit. Therefore, it requires an additional message round trip between the  $TM$  and the  $RM$ s before the commit starts. It saves a message round trip, but faces a higher chance of cascade abort. We implement four types of DLV scheme which are different on violation time, defined as is the following:

**DLV0:** once a transaction has finished an access operation, the lock which is used to protect this operation can be violated immediately;

**DLV1:** when a transaction located at an  $RM$  has decided its prepare decisions before it writes its log, the locks on this shard hold by this transaction would be violable;

**DLV1x:** if a transaction can guarantee all the  $RM$  related has determined their prepare, although the log has not written, conflict operations from other transactions can violate any locks of this transaction;

**DLV2:** a transaction enters the second phase of 2PC, then all of the transactions' locks are violable.

DLV0 and DLV1 is early lock violation, and DLV1x and DLV2 is late lock violation. DLVx implementation late lock violation by adding a message round trip to prevent deterministic abort. This additional message flow shows in Fig. 2 and the message is showing as dotted arrow lines. Before an *RM* decides to replicate its prepare log, it also sends a *Ready* message to *TM* and tells *TM* it will prepare this transaction. *Ready* message shows that the *RM* will prepare commit or prepare abort. When the *TM* collect all *RM*s *Ready* message, it sends *Violate* messages to tells every *RM*s make their locks are violable. An interactive-transaction can combine these messages with the last operations and prepare requests in passing, as Fig. 4b shows. Interactive transactions use DLV would only work as *late violation*. We mainly focus on discuss one-shot transaction model here. For a one-shot transaction, this message flow also takes less time than the overall message flow of 2PC because of no log replication time dealy. Especially when all the *RM*s and the *TM* are located in LAN, there is no WAN message RTT.

### C. Deadlock Prevention and Dependency Tracing

DLV adopts the *wait-die* protocol to prevent deadlocks and dependency cycles. A transaction is assigned a unique timestamp when it starts. The timestamp is not necessarily globally synchronized. A transaction can generate its timestamp by combining the local clock and unique local node id. Conflicting transactions can then be ordered by their timestamps. Once a transaction will wait if it attempts to access a data item locked by a transaction with a larger timestamp. Otherwise, it would abort. As mentioned earlier, this is a simple and effective approach in a distributed environment. Ealy lock violation also obeys *wait-die* rule. A transaction can only violate a lock with greater transaction id, which is the same as wait rule. By this policy, transaction dependencies graph can not create a circle.

DLV applies the register-and-report method [28] to trace commit dependencies among transactions. Each transaction maintains an in-dependency count (*in* attribute of Context struct in Algorithm 1), which records how many transactions it depends on. Each transaction also maintains an out-dependency transaction set (*out* attribute of Context struct in Algorithm 1), which records all the transactions depending on it. When a transaction *T* reads from *S*, DLV registers the commit dependency by adding *T* to the *out* set of *S* and incrementing *in* of *T* by one. A transaction cannot commit if it's *in* value is greater than 0, which suggests some of its in-dependency transaction has not committed yet. While a transaction aborts, it may make its out-dependency transactions to abort cascade. For cascade abort purpose, it notifies all its out-dependency transactions to abort too, by setting their *in* values to a negative value, for example,  $-\infty$  value. Transactions which has a negative *in* value has to abort for there dependency transaction abort. During a transaction commits, the transaction will traverse its *out* dependency set and decrease the *in* value of its out-dependency transactions by one. The DB can maintain this dependency information in

main memory. If there is node failure, the transaction can drop this information safely.

### D. Pseudocode Description

---

#### Algorithm 1 Transaction Context and Lock Structure Definition

---

```

1: struct Context {
2:   xid;                                ▷ transaction id
3:   in;                                ▷ in dependency transaction count
4:   out;                                ▷ out dependency transaction set
5:   locks;                              ▷ lock acquired
6:   write_set;                          ▷ write row set
7:   violate;                            ▷ EARLY or LATE
8: };

1: struct Lock {
2:   xid;                                ▷ transaction id
3:   key;                                ▷ locked key
4:   mode;                              ▷ lock mode, READ or WRITE
5:   violate;                            ▷ EARLY, LATE or NONE
6: };

```

---

Algorithm 1 describes the transaction context on *RM* and the lock structure. The meanings of each struct attributes are shown in the comments. Both these structures have a *violate* attribute to indicate which lock violation scheme the transaction used. The *violate* value of *Context* are initialized when the transaction context is created. It would be assigned by *EARLY* if DLV0 or DLV1 setting, and *LATE* if DLV1x or DLV2 setting. *Lock* structure has *violate* attribute which is initialized as *NONE* to note it cannot be violated. Depends on the lock violation strategy, it would become *EARLY* or *LATE* to express the lock can be violated early or late.

In Algorithm 2, function *Lock* describes the procedure of transaction locking acquire. It has 3 input arguments: a transaction context struct *T*, the key *K* to lock, lock mode *M*. First, the *Lock* function checks if there's a conflicting lock in the lock table. It also tests whether it can be violated by the current transaction. Transaction *T* can only wait for a lock who hold by another transaction has a greater transaction *xid*<sup>2</sup>. Otherwise, transaction *T* would fail to acquire this lock and abort later. Once any locks are violated, *T* would register its dependency if it is necessary. Finally, *Lock* would return the newly created lock.

Algorithm 3 inclues test violable function and enable violation function. Function *Violable* return weather a transaction *T* can violate lock *L* on *K* in *M* mode. *EnableTxV* would let all the locks on a shard can be violated. *EnableLkV* is used to make a lock can be violated.

Algorithm 4 depicts the execution phase of a transaction includes Read and Write functions. Read and Write function would lock the read or write key. The lock would be added to the lockset(*locks* of Context struct) of transaction context if

<sup>2</sup>we suppose there is no lock upgrade, so there are no conflict locks hold by one transaction



the lock acquire succeed. After the transaction access the key, EnableLkV would be called to enable this lock violable if the current database is in DLV0 setting.

Algorithm 5 describes the prepare phase of a transaction. When DB use DLV1 configuration, transaction enable let its lock violable in early violation scheme. If DLV1x is setting, *RM* issue *TM* a *Ready* message to tell *TM* this *RM* is going to prepare. In the next, the transaction context would wait on its in-dependency. The transaction context cannot enter the prepare phase if its *in* value is greater than 0, which stands it has in dependency transaction. Once *in* become 0, the transaction processing can go to its prepare phase. In some situations, there may be cascade abort. Then the *in* value less than 0 indicates *T* would cascade abort.

Algorithm 6 the final phase of 2PC. When a *RM* receive *Commit* message from *TM*, it would run *Commit* function. In *Commit* function, the transaction would test if DLV2 setting on and enable violation. It cleans its write version, writes log and releases the locks. If a current transaction has out dependencies, the *in* value of the dependency transaction would be decreased one. The transaction reports the dependency transactions to stop waiting if *in* equal 0 after the subtraction. Abort procedure first write the abort log, clean dirty version of this transaction and release log. Transactions who depends on this transaction would be noticed to abort by setting their *in* dependency count negative.

## V. EXPERIMENTS AND EVALUATIONS

We develop a replicated distributed DBMS demo and evaluate implement different types of DLV, as we discussed previously. As a comparison with DLV, we also implement S2PL wait die scheme.

### A. Experiments Setting

Our experiments performed on a cluster of 13 Aliyun general ecs.g6.large server. In which one node was used to issue transaction requests, the others acted as database nodes. Each server had 2 CPU core with 2.5GHz base frequency and 3.2 GHz turbo frequency, 8GB RAM, running Ubuntu 18.04. The data were partitioned by 4 shards, each shard had 3 replicas and the logs were replicated across 3 AZs. These 3 AZs were located at Heyuan South China, Hangzhou East China and Zhangjiakou North China. The internal network bandwidth of each AZ was 1Gbps. Every AZ had a full data copy of each shard. Each transaction would retry after 3 seconds if aborted.

### B. TPCC Performance Evaluation

TPC-C [31] models a real world workload that a company has many warehouses and districts, customers order their products, pay their bills. In our version of TPC-C benchmark, we exclude the think time and only NewOrder procedure executes. The data is partitioned into 4 shards by warehouse id. The item table is replicated across all the shard. We also extend TPC-C workload make all the transactions are distributed

---

### Algorithm 2 Transaction Lock/UnLock a Key

---

*T* transaction context

*K* lock key

*L* the lock to be violated

*M* lock mode(*Read* or *Write*)

---

```

1: function Lock(T, K, M)
2:   wait if  $\exists$  such  $l \in \text{lock\_table}$ ,
      Conflict(K, T, l, M) and
      not Violable(K, T, l, M)
3:     if T.xid > l.xid
4:       return nil ▷ lock die error
5:     end if
6:   end wait
7:   for all such  $l \in \text{lock\_table}$ ,
8:     Violable(K, T, l, M) do
9:        $s \leftarrow$  transaction context retrieved by l.xid
10:      if T.violate = EARLY or ( T.violate = LATE
      and l.mode = WRITE and M = READ ) then
11:        if T.xid  $\notin$  s.out then
12:           $s.out\_set \leftarrow s.out \cup T.xid$ 
13:           $T.in \leftarrow T.in + 1$ 
14:        end if
15:      end if
16:    end for
17:    create lock  $l' \leftarrow \{T.xid, K, M, None\}$ 
18:    return lock  $l'$ 
19: end function

1: function UnLock(T, K)
2:   if  $\exists l \in \text{lock\_table}$ , l.xid = T.xid and
3:     l.key = K then
4:     remove such l from lock_table
5:   end if
6: end function

1: function Conflict(K, T, L, M)
2:   if T.xid  $\neq$  L.xid and K = L.key then
3:     return L.mode = WRITE or M = WRITE
4:   else
5:     return false
6:   end if
7: end function

```

---

transaction by enforcing every transaction orders items from different warehouses located different node.

First, we simulate different log cost on a LAN replicated environment by adding log replication latency. Fig. 10 shows the throughput and abort rate of TPCC NewOrder procedure on 4 shards. Every shard replicated transactional log with user setting latency time. The result shows that DLV can perform better when there is a large log replication latency. On a low replication latency environment, DLV cannot improve performance.

In the following experiments, we all run out evaluations on a real geo-replication environment. The evaluation runs on scattered (leader) mode or gathered (leader) mode. In gathered



**Algorithm 3** Lock Violation Test and Enable Violation $T$  transaction context $K$  lock key $L$  the lock to be violated $M$  lock mode, READ or WRITE $VT$  lock violation type, EARLY or LATE

---

```

1: function Violable( $T, K, L, M$ )
2:   if  $\text{Conflict}(K, T, L, M)$  and
3:    $L.\text{violate} \neq \text{NONE}$  then
4:     if  $L.\text{violate} = \text{EARLY}$ 
5:     and  $T.\text{violate} = \text{EARLY}$  then
6:       return  $T.\text{xid} < L.\text{xid}$ 
7:     else if  $L.\text{violate} = \text{LATE}$  then
8:       return true
9:     end if
10:  else
11:    return false
12:  end if
13: end function

1: function EnableTxV( $T, VT$ )
2:   for all  $l \in T.\text{locks}$  do
3:      $l.\text{violate} \leftarrow VT$ 
4:   end for
5: end function

1: function EnableLkV( $L, VT$ )
2:    $l.\text{violate} \leftarrow VT$ 
3: end function

```

---

**Algorithm 4** Execution Phase of Transaction $T$  transaction context $K$  read/write key $V$  value write

---

```

1: function Read( $T, K$ )
2:    $l \leftarrow \text{Lock}(T, K, \text{READ})$ 
3:   if  $l$  is nil then
4:     return lock failed  $\triangleright T$  will abort later
5:   else
6:      $T.\text{locks} \leftarrow T.\text{locks} \cup l$ 
7:     if DLV0 setting, call  $\text{EnableLkV}(l, \text{EARLY})$ 
8:     return  $v$ 
9:   end if
10: end function

1: function Write( $T, K, V$ )
2:    $l \leftarrow \text{Lock}(T, K, \text{WRITE})$ 
3:   if  $l$  is nil then
4:     return lock failed  $\triangleright T$  will abort later
5:   else
6:      $T.\text{locks} \leftarrow T.\text{locks} \cup l$ 
7:     assign  $K$ 's newest version by value  $V$ 
8:     add tuple to  $T.\text{write\_set}$ 
9:     if DLV0 setting, call  $\text{EnableLkV}(l, \text{EARLY})$ 
10:    return write success
11:   end if
12: end function

```

---

**Algorithm 5** Prepare Phase of Transaction  $T$ 


---

```

1: function Prepare( $T$ )
2:   if DLV1 setting, call  $\text{EnableTxV}(T, \text{EARLY})$ 
3:   if DLV1x setting, issue  $TM$  a  $\{\text{Ready}\}$  message.
4:   wait if  $T.\text{in} > 0$ 
5:   if  $T.\text{in} < 0$  then
6:     write prepare abort log
7:     response  $TM$  message  $\{\text{Prepare Abort}\} \triangleright$  cascade
8:   else if  $T.\text{in} = 0$  then
9:     write prepare commit log
10:    response  $TM$  message  $\{\text{Prepare Commit}\}$ 
11:   end if
12: end function

```

---

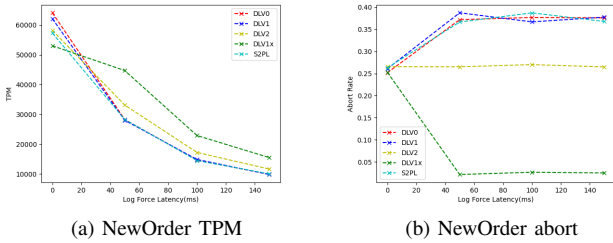


Fig. 10: throughput and abort rate of different terminal different log write costs

mode, all of the replica leaders are located in the same AZs. In scattered mode, the replica leaders are not located in the same AZs.

Fig. 11 is the performance comparison of different warehouse number. DLV perform better when the warehouse number is small. The less warehouse number the data divided, the more contention the transaction would incur and DLV only optimize contention workload. DLV1x perform better among other approaches various warehouse numbers.

Fig. 12 shows the NewOrder performance of when adding terminal numbers of each node in the gathered mode. In gathered mode, the message between replica leaders has the minimize RTT. DLV1x can benefit from less RTT cost. DLV1x scale well when adding terminal numbers of each shard. Fig.

13 shows that even in scattered mode, leader communication has more cost. DLV1x perform better than other DLV scheme.

### C. YCSB Performance Evaluation

YCSB [32] workload is designed for evaluating internet storage. Each YCSB transaction access nearly 10 rows and the access key follows Zipfian distributions. The degree of skew is represented by a parameter  $\theta$ . Our YCSB workload is sharded and range partitioned by the primary key of YCSB's main table. Because we mainly focus on the distributed transaction, our workload guarantees every transaction access at least two shards and all the transaction are distributed transactions. We evaluate DLV performance varies on different data skew.

---

**Algorithm 6** Commit/Abort Phase of Transaction  $T$ 


---

```

1: function Commit( $T$ )
2:   if DLV2 setting, call EnableTV( $T$ )
3:   write commit log
4:   for all  $r \in T.write\_set$  do
5:     make the version written by  $T$  becomes commit
     version of tuple  $t$ 
6:   end for
7:   for all  $l \in T.locks$  do
8:     UNLOCK( $l$ )
9:   end for
10:  for  $t \in T.out$  do
11:     $t.in \leftarrow t.in - 1$  ▷ keep exactly once
12:    if  $t.in = 0$  then
13:      report  $t.in = 0$  ▷ stop waiting on  $t$ 
14:    end if
15:  end for
16:  response  $TM$  message {Commit ACK}
17: end function

1: function ABORT( $T$ )
2:   write abort log
3:   for all  $r \in T.write\_set$  do
4:     remove the version written by  $T$  in  $r$ 
5:   end for
6:   for all  $l \in T.locks$  do
7:     UNLOCK( $l$ )
8:   end for
9:   for  $t \in T.out$  do
10:     $t.in \leftarrow -\infty$ 
11:    report  $t.in < 0$  ▷ stop waiting on  $t$ 
12:   end for
13:   response  $TM$  message {Abort ACK}
14: end function

```

---

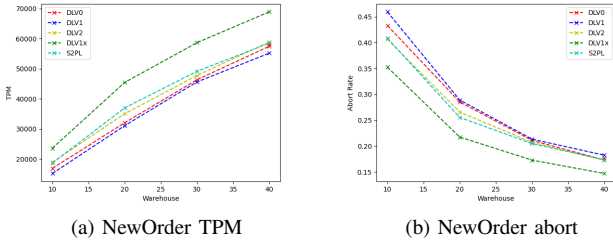


Fig. 11: throughput and abort rate of different warehouse numbers

Fig. 14 depicts YCSB throughput and abort rate when adding data skew  $\theta$ . The result shows DLV1x scales well on different degrees of data skew.

## VI. CONCLUSION

We extend CLV to the distributed transaction and evaluate its performance on a geo-replicated environment. Like traditional single-node CLV, DLV can consolidate many discrete waits during transaction running time into one final wait when

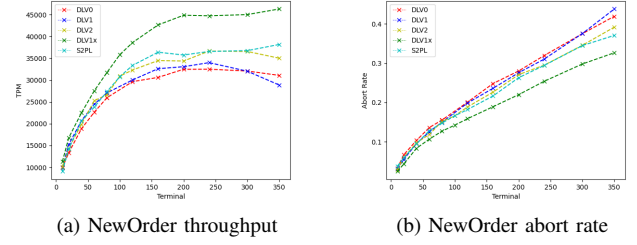


Fig. 12: throughput and abort rate when adding terminal number of each shard, in gathered mode

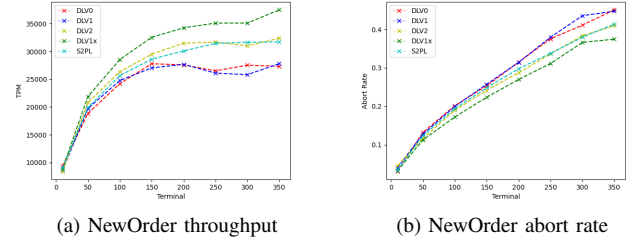


Fig. 13: throughput and abort rate when adding terminal number of each shard, in scattered mode

transaction commit or prepare. According to our evaluation, not all the workload can gain performance improvement due to lock violation cost. DLV minimize transaction dependency tracing cost and possible cascade abort rate compare previous work. By cutting down the transaction critical path of transaction commit and replication and maintaining dependencies, DLV achieves both correctness and performance.

## REFERENCES

- [1] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi, “Calvin: fast distributed transactions for partitioned database systems,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds. ACM, 2012, pp. 1–12. [Online]. Available: <https://doi.org/10.1145/2213836.2213838>
- [2] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, “Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes,” in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018*,

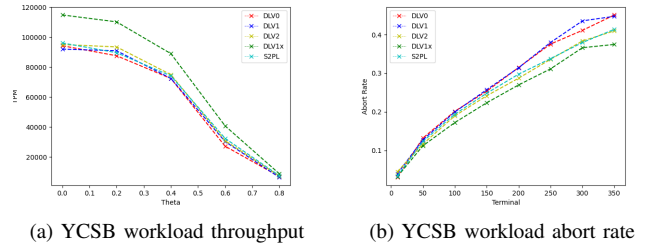


Fig. 14: YCSB performance, throughput and abort rate when adding theta, in scattered mode

- Houston, TX, USA, June 10-15, 2018, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 789–796. [Online]. Available: <https://doi.org/10.1145/3183713.3196937>
- [3] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, “Building consistent transactions with inconsistent replication,” in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSOP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 263–278. [Online]. Available: <https://doi.org/10.1145/2815400.2815404>
  - [4] S. Mu, L. Nelson, W. Lloyd, and J. Li, “Consolidating concurrency control and consensus for commits under conflicts,” in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds. USENIX Association, 2016, pp. 517–532. [Online]. Available: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu>
  - [5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, “Spanner: Google’s globally-distributed database,” in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, C. Thekkath and A. Vahdat, Eds. USENIX Association, 2012, pp. 251–264. [Online]. Available: <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett>
  - [6] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford, “Spanner: Becoming a SQL system,” in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds. ACM, 2017, pp. 331–343. [Online]. Available: <https://doi.org/10.1145/3035918.3056103>
  - [7] “Nuodb,” <https://www.nuodb.com/>.
  - [8] “Cockroachdb,” [3] <https://www.cockroachlabs.com/>.
  - [9] “Tidb,” <https://pingcap.com/en/>.
  - [10] H. Kimura, G. Graefe, and H. A. Kuno, “Efficient locking techniques for databases on modern hardware,” in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012, Istanbul, Turkey, August 27, 2012*, R. Bordawekar and C. A. Lang, Eds., 2012, pp. 1–12. [Online]. Available: [http://www.adms-conf.org/kimura\\_adms12.pdf](http://www.adms-conf.org/kimura_adms12.pdf)
  - [11] G. Graefe, M. Lillibridge, H. A. Kuno, J. Tucek, and A. C. Veitch, “Controlled lock violation,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 85–96. [Online]. Available: <https://doi.org/10.1145/2463676.2465325>
  - [12] “Latency,” <https://gist.github.com/jboner/2841832>.
  - [13] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998. [Online]. Available: <https://doi.org/10.1145/279227.279229>
  - [14] —, “Paxos made simple, fast, and byzantine,” in *Proceedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*, ser. Studia Informatica Universalis, A. Bui and H. Fouchal, Eds., vol. 3. Suger, Saint-Denis, rue Catulienne, France, 2002, pp. 7–9.
  - [15] R. van Renesse and D. Altinbeken, “Paxos made moderately complex,” *ACM Comput. Surv.*, vol. 47, no. 3, pp. 42:1–42:36, 2015. [Online]. Available: <https://doi.org/10.1145/2673577>
  - [16] D. Ongaro and J. K. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, G. Gibson and N. Zeldovich, Eds. USENIX Association, 2014, pp. 305–319.
  - [17] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker, “An evaluation of distributed concurrency control,” *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 553–564, 2017. [Online]. Available: <http://www.vldb.org/pvldb/vol10/p553-harding.pdf>
  - [18] [Online]. Available: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>
  - [19] “Voltdb,” <http://www.voltdb.com/>.
  - [20] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, “Staring into the abyss: An evaluation of concurrency control with one thousand cores,” *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 209–220, 2014. [Online]. Available: <http://www.vldb.org/pvldb/vol8/p209-yu.pdf>
  - [21] P. A. Bernstein and N. Goodman, “Concurrency control in distributed database systems,” *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, 1981. [Online]. Available: <https://doi.org/10.1145/356842.356846>
  - [22] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, “Granularity of locks and degrees of consistency in a shared data base,” in *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, G. M. Nijssen, Ed. North-Holland, 1976, pp. 365–394.
  - [23] “Mysql,” <https://www.mysql.com/>.
  - [24] “Postgresql,” <https://www.postgresql.org/>.
  - [25] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood, “Implementation techniques for main memory database systems,” in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84*, Boston, Massachusetts: ACM Press, 1984, p. 1. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=602259.602261>
  - [26] E. Soisalon-Soininen and T. Ylönen, “Partial strictness in two-phase locking,” in *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, ser. Lecture Notes in Computer Science, G. Gottlob and M. Y. Vardi, Eds., vol. 893. Springer, 1995, pp. 139–147. [Online]. Available: [https://doi.org/10.1007/3-540-58907-4\\_12](https://doi.org/10.1007/3-540-58907-4_12)
  - [27] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki, “Aether: A scalable approach to logging,” *PVLDB*, vol. 3, no. 1, pp. 681–692, 2010. [Online]. Available: [http://www.vldb.org/pvldb/vldb2010/pvldb\\_vol3/R61.pdf](http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R61.pdf)
  - [28] P. A. Bernstein, “Actor-oriented database systems,” in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 13–14. [Online]. Available: <https://doi.org/10.1109/ICDE.2018.00010>
  - [29] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, “High-performance concurrency control mechanisms for main-memory databases,” *PVLDB*, vol. 5, no. 4, pp. 298–309, 2011. [Online]. Available: [http://vldb.org/pvldb/vol5/p298\\_perakelarlson\\_vldb2012.pdf](http://vldb.org/pvldb/vol5/p298_perakelarlson_vldb2012.pdf)
  - [30] Y. Raz, “The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment,” in *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, L. Yuan, Ed. Morgan Kaufmann, 1992, pp. 292–312. [Online]. Available: <http://www.vldb.org/conf/1992/P292.PDF>
  - [31] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H. Schek, and G. Weikum, “Unifying concurrency control and recovery of transactions,” *Inf. Syst.*, vol. 19, no. 1, pp. 101–115, 1994. [Online]. Available: [https://doi.org/10.1016/0306-4379\(94\)90029-9](https://doi.org/10.1016/0306-4379(94)90029-9)
  - [32] R. O. Nambiar, N. Wakou, A. Masland, P. Thawley, M. Lanken, F. Carman, and M. Majdalany, “Shaping the landscape of industry standard benchmarks: Contributions of the transaction processing performance council (TPC),” in *Topics in Performance Evaluation, Measurement and Characterization - Third TPC Technology Conference, TPCTC 2011, Seattle, WA, USA, August 29-September 3, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, R. O. Nambiar and M. Poess, Eds., vol. 7144. Springer, 2011, pp. 1–9. [Online]. Available: [https://doi.org/10.1007/978-3-642-32627-1\\_1](https://doi.org/10.1007/978-3-642-32627-1_1)
  - [33] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, Eds. ACM, 2010, pp. 143–154. [Online]. Available: <https://doi.org/10.1145/1807128.1807152>