Exploit 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.

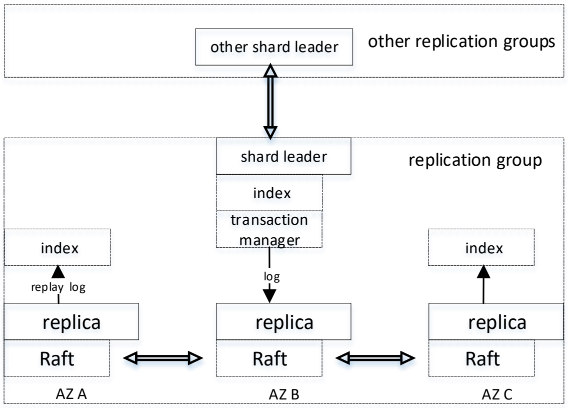


Fig. 1: Typical architecture of geo-replicated distributed DBMS

Figure 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. Figure 2 shows the message flow of a

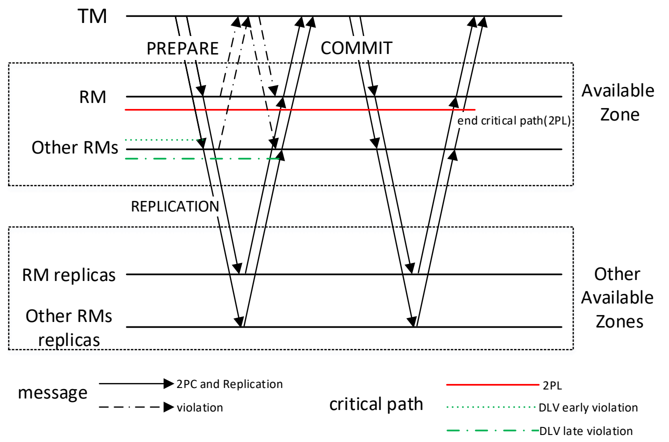


Fig. 2: The message flow and lock holding critical path when the DBMS uses S2PL for concurrency control and 2PC as the commit protocol. The dash arrow lines represents the messages introduced by Distributed Lock Violation (DLV). The red lines represent the critical paths of S2PL. The green dash lines 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* [1] . 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 Figure 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. 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, when working with 2PC, we need to decide to violate (or release) lock at which phase.

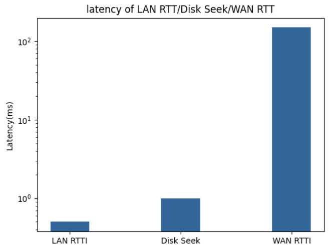


Fig. 3: log write latency in different environment

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 decides the best time to perform lock violation by looking at runtime statistic information. 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 [12] [13] 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 [14] elects a leader as the only proposer to eliminate the first message round trip. Raft [15] 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 [16] and Calvin [1] employ a deterministic transaction model to reduce the

[1] Depending on variants of implementations, the TM can choose to persist its decision on its log or not.

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 Figure 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* [17] 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 [18] [17], as it eliminates blocking completely. However, it works poorly when dealing with workload of high contention. Another approach is *wait-die* [19]. It avoids some false-positive aborts encountered by *no-wait* by utilizing timestamps. The *Deadlock detection* approach [20] detects deadlock by explicitly tracing wait-for relationship among transactions. Many centralized database systems [21] [22] 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) [23] [24] [25] [10] [26] 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. [23]. Soisalon-Soininen et al. [24] analyzed its correctness in various settings. Many other works applied and evaluated ELR in different types of systems [25] [10] [10] [25]. 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 [27] 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.

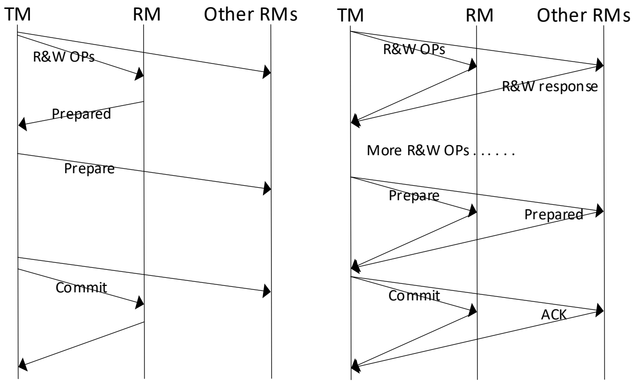
III. Scheduling with Lock Violation

In the following, we describe how to ensure correct scheduling with lock violation.

A. *Preliminaries and Assumptions*

A database system normally shards its data by primary keys. For each data chunk, we replicate its physiological logs across different AZs to achieve fault-tolerance. The physiological logs record the row-level write operations of each transaction. The replicated layer uses consensus protocol to maintain the consistency among the replicas. We assume that there is a replica leader for each data chunk, which is responsible for receiving requests.

We assume that the system supports both one-shot and interactive transactions. Figure 4 shows the message flow during the commit phase of the two types of transaction. We can see that an interactive transaction costs more message round trips than a one-shot transaction. Our lock violation technique must work with both types of transactions, which we will explain subsequently.



(a) one-shot transaction, the prepare message is a suggestive one to let the RM issue response commit/abort message

(b) interactive transaction, when a RM receive a *Prepare* message, transaction can prevent deterministic abort.

Fig. 4: commit message flow of one-shot and interactive distributed transaction

B. Concepts of Scheduling

Before presenting our method, we first review the conceptual model of transaction processing. Most of the contents can be found in the previous work [19].

1) Transaction and History: Suppose a distributed database resides on n nodes, represented by $R = \{r_1, r_2, ...r_n\}$. A transaction T_i can run on any m of the n nodes ($1 \leq m \leq n$), represented by $S = \{s_1, s_2, ...s_m\} \subseteq R$, The transaction T_i is composed of a series of operations, where each operation can be a read or a write or any command operation including abort, commit, etc. Let $r_i[x]$ denote that T_i reads record x, and $w_i[x]$ denotes that T_i writes 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. We call T_i's operations on the node s_u T_i's *sub-transaction* on s_u. A transaction history is a collection $H = \{h_1, h_2, ..., h_n\}$, 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.

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);

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*, as the latter is not predictable using the operational logic of the database system.

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

Based on data dependencies, we can define *commit dependency*. T_j has a *commit dependency* on T_i, written as $T_i \to T_j$, 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* or *ww-dependency* on T_i.

A transaction T_j *speculatively access* a record x, if there is another transaction T_i, T_j has a commit dependency on T_i and T_i has not committed. We call such a commit dependency a *danger dependency*, denote it as $T_j \to_s T_i$.

4) Relaxation on Strictness: Traditional transaction schedulers choose strict two-phase locking (S2PL) [28] to avoid expensive transaction recovery cost. Strictness implies that a transaction cannot read or overwrite a previous write by another transaction which has not committed yet.

Strictness is not a necessary condition for a correct schedule. For instance, the schedule H_1 of Figure 5 is serializable and strict. (The data dependencies include $r_1[x] \to w_3[x]$, $w_1[x] \to r_2[y]$, $r_2[y] \to w_3[x]$.) In contrast, H_2 in Figure 6 is serializable but not strict. In Figure 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. The schedule H_2 can be represented as

$$H_2 = \{h_1, h_2, h_3\},$$

in which,

$$\begin{aligned} h_1 &= w_1[x], w_3[x], p_1^c, c_1, p_3^c, c_3 \\ h_2 &= w_1[y], r_2[y], p_1^c, c_1, p_2^c, c_2 \\ h_3 &= r_2[z], w_3[z], p_2^c, c_2, p_3^c, c_3. \end{aligned}$$

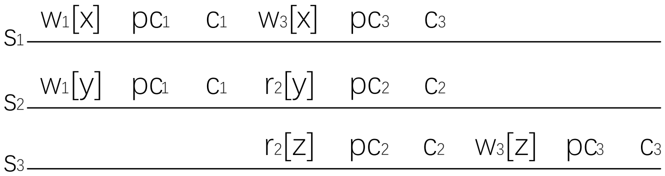

S1 w1[x] pc1 c1 w3[x] pc3 c3
S2 w1[y] pc1 c1 r2[y] pc2 c2
S3 r2[z] pc2 c2 w3[z] pc3 c3

Fig. 5: A strict and serializabile schedule H_1

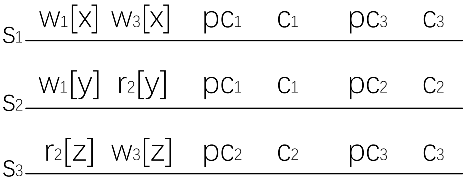

S1 w1[x] w3[x] pc1 c1 pc3 c3
S2 w1[y] r2[y] pc1 c1 pc2 c2
S3 r2[z] w3[z] pc2 c2 pc3 c3

Fig. 6: A non-strict but serializabile schedule H_2

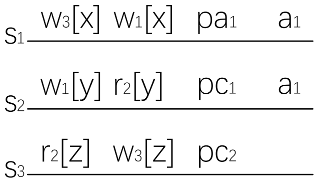


Fig. 7: Schedule H_3, T_1 abort due to non-serializabile

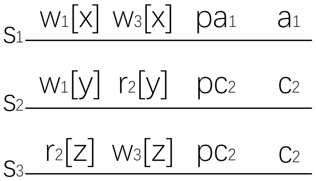


Fig. 8: Schedule H_4, T_2 commit ahead T_1, non-recoverable anomaly

As we can see, by violating strictness, H_2 in Figure 6 enables a higher degree of concurrency. To realize such a schedule, T_1 must release its lock or mark it as violatable before it commits.

The commit of a transaction may involve a lot of work, especially when it is a distributed commit. Strictness requires a transaction to hold locks until it finishes committing. In a distributed environment, the lock holding time will be substantially increased. Our basic idea is to develop a serializable but non-strict scheduler for distributed transactions and shorten the lock holding time.

Relaxation on strictness will, on the other hand, complicate the logging mechanism. In a traditional logging mechanism, the dependencies among transactions completely comply with the persisting order of their logs [23] [10]. This guarantees the recoverability of scheduling, which requires that a transaction can commit only if the transactions which it has read from commit. When strictness no longer holds, we need other tactics to preserve recoverability.

C. Lock Violation and Dependency Tracing

Lock violation is a technique to enable non-strict schedules. For lock violation to be correct, it should first preserve serializability. Considering the non-strict schedule H_3 in Figure 7, it contains three data dependencies:

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

This leads to a circle $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$, making the schedule non-serializable. H_3 is not possible if we apply S2PL. It becomes possible if we perform lock violation – it allows T_2 to read from uncommitted T_1, and T_1 to overwrite uncommitted T_3. Therefore, to preserve serializability, we need to trace the data dependencies among transactions. If a lock violation is about to result in a dependency cycle, we can disable it. For instance, we forbid T_2 from violating the lock of T_1.

In fact, lock violation can occur at two occasions in transaction processing. We call them *early lock violation* and *late lock violation*. Suppose H is a schedule which is created by lock violation. We complement H with the lock, unlock and lock violation operations, so it looks as follows:

$$H = l_i[x]o_i[x]...vl_j[x]o_j[x]...l_i[y]o_i[y]...ul_i[x]...$$

In H, x, y are separate records. T_i first locks x (i.e., $l_i[x]$). Then, T_j violates T_i's lock on x (i.e., $vl_j[x]$). Following that, T_i locks y and releases its lock on x (i.e., $l_i[y]...ul_i[x]$). If a lock violation, such as $vl_j[x]$, occurs before its victim transaction plays another lock, such as $l_i[y]$, we call it an *early lock violation*. Otherwise, we call it a *late lock violation*.

It was proven previously that *early lock violation* can lead to non-serializable schedules. When using early lock violation, we have to trace all data dependencies to ensure that the dependency graph is acyclic. In contrast, *late lock violation* is safe, as it cannot make an acyclic dependency graph cyclic. Our DLV chooses to perform late lock violation only.

Secondly, a schedule generated by lock violation must be recoverable. In the schedule H_4 in Figure 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.

Suppose there is a *wr-dependency* from T_i to T_j, which is written as $w_i[x] \rightarrow r_j[x]$. For recoverability, we must ensure that T_j cannot commit if T_i has not committed. Traditional S2PL can guarantee this by releasing T_i's locks after T_i commit. In the case of lock violation, we trace the wr-dependencies among transactions and make sure that a 2PC scheduler obeys the following rules:

1) T_j's RM can be prepared 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. DLV IMPLEMENTATION

In this section, we introduce DLV implementation. The following contents would include: How DLV can avoid complex recovery algorithm and maintain the most limited amount of dependencies; How DLV choose the proper time of enabling violation; The wait-die policy of DLV use; The pseudocode code description finally.

A. In Memory Speculative Versions

The non-strict scheduler needs more complex recovery algorithm to keep the correctness. Take a schedule H_5 as an example,

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

If transaction T_1 abort, this cause T_2 cascade abort for recoverability. Traditional database use undo log to process recovery transaction write operations. Implementation undo log maybe a little bit tricky when exploiting non-strict. A wrong recovery expand schedule of T_1 may be like $exp(H_5)$, in which $w_i^-[x]$ means transaction T_i undo 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, lets take Table I as 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. First, 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 T_1's write value. Transaction T_1 abort due to some reason and undo its write. Then T_2 also abort and undo its write by a wrong value of x, which is T_2 read from T_1. Finally, after the execution of this schedule, both transaction T_1 and T_2 aborts. The value of x is 1, which the correct result should be the initial value 0.

Transaction use lock violation can produce this type non-strict schedule. Transaction cannot rollback individually. The scheduler needs a complex algorithm to handle undo order properly without lock protection, even when there is a transaction recovery rather than a system recovery. To tackle this anomaly, recovery must use a more complex algorithm such

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 $exp(H_5)$

as SOT [29]. For schedule H_5, a correct recovery expandation may be:

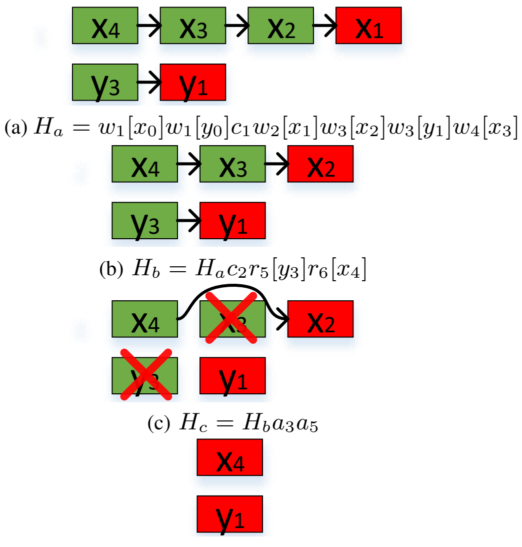
$$exp(H_5) = w_1[x]w_2[x]w_2^-[x]c_2w_1^-[x]c_1$$

The schdeuler must rollback transaction by the reserve order of write operation. To avoid this complexity, DLV maintains uncommitted speculative versions in memory and accepts no-steal policy when writing data. No-steal policy need storage cannot write uncommitted data to permanent storages. For most transactional workloads would write a little data except the bulk loading ones. On ther other hands, the modern database runs on a machine with large RAM, using no-steal policy to save memory is not necessary. By no-steal and speculative versions, the database needs no undo log, transaction rollback and failure recovery would be more simple and efficient. DLV 's speculative version implementation is a little similar with many multi-version concurrency control scheme. The list is structured 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 only single version, the committed version. When a transaction commit, it cooperatively clean its outdated version from the version list. If a transaction would abort, it only needs to remove its write versions from the speculative version list.

Previously, we have discussed that a *ww* dependency does not affect recoverability. *Late lock violation*, since it has promised serializability, so it can ignore ww and rw dependencies and only trace wr dependencies for recoverability. Figure 9 show a series of schedule access on two contention rows, x, y. The green rectangles are speculative versions and the red ones are committed versions. Although there is ww dependency $w_6[x] \to w_4[x]$. The abort of T_4 does not cause T_6 cascade abort.

B. Lock Violation Time

DLV exploit *early lock violation* when transaction is on going even preceding transaction request its (prepare) commit. In early lock violation mode, if a transaction encouter a lock timeout when waiting a lock of another transaction, it could violate this lock immediately. *Early lock violation* can is more appropriate then *late lock violation* when there are less cascade abort caused by non-deterministic abort. *Early lock violation* also need more dependency tracing costs. Too many cascade abort can contributed to a lot of useless work.



(a) $H_a = w_1[x_0]w_1[y_0]c_1w_2[x_1]w_3[x_2]w_3[y_1]w_4[x_3]$

(b) $H_b = H_ac_2r_5[y_3]r_6[x_4]$

(c) $H_c = H_ba_3a_5$

(d) $H_d = H_cc_4r_6[y_1]c_6$

Fig. 9: speculative(green) and committed(red) versions, x_i , y_i express this version is from transaction transaction T_i's write

DLV implementation *late lock violation* by adding a message round trip to prevent deterministic abort when use lock violation. This additional message flow shows in Algorithm 7. In Figure 2, the message is show 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 *RMs* make their locks are violative. An interactive-transaction can combine these messages with the last operations and prepare requests in passing, as Figure 4b shows. So, the interactive transaction needs no dynamic decides the lock violation time. 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 RMs and the TM are located in LAN, there is no WAN message RTT.

DLV records transaction statistic information to decide use *early lock violateion* or *late lock violation*. Assume at some point there are n most recency ending (commit or abort) transactions $T_1, T_2, ...T_n$. For any transaction T_i, T_i would access $|S(i)|$ shards, we record this collection of shards by: $S(i) = \{s_1(i), s_2(i), ..., s_{|S(i)|}(i)\}$.

A partial transaction $t_{ij}, 1 \leq j \leq |S(i)|$, is T_i's transactional sequence operations working on shard $s_j(i)$. We define following conceptions on partial transaction t_{ij}:

partial span time, $\gamma(i,j)$: t_{ij}'s *partial span time* is from when t_{ij} begin its first access operation to its finish writing its final end log(commit or abort) on its shards. If the database saves more lock cost by violation than possible cascade abort cost, it would choose *early lock violation*. *partial max lock wait time,* $\delta(i,j)$: The maximum waiting time on a lock of all the locks t_{ij} acquires on its shards.

partial prepare: t_{ij} successfully prepared to commit(enter phase1 of 2PL) but failed to commit finally on its shards. We use function $\eta(i,j)$ to express wheathor t_{ij} is partial prepare or not. When t_{ij} is partial prepare, $\eta(i,j)$ returns 1. Otherwise, it returns 0.

DLV would choose *early lock violation* if the following conditions satisfied. Otherwise, it would use *late lock violation*.

$$\frac{\sum_{i=1}^{n}\sum_{j=1}^{|S(i)|}\gamma(i,j)}{\sum_{i=1}^{n}\sum_{j=1}^{|S(i)|}\delta(i,j)} > p\frac{\sum_{i=1}^{n}\sum_{j=1}^{|S(i)|}\eta(i,j)}{\sum_{i=1}^{n}|S(i)|}$$

In this inequality, p is constant coefficients. The term on the lefthand side of the inequality can be considered the cost saved by lock violation and the right term is the cost of possible cascade abort.

C. Locking Violation and Maintain Commit Dependencies

DLV use *wait-die* protocol to avoid deadlock. At the beginning of a transaction, the transaction uses the current timestamp to generate a transaction id. The conflict transaction operations are queued base their transaction id's order. In early lock violation mode, DLV would consolidating all the wait in the final wait. Ealy lock violation also obeys *wait-die* rule. A transaction can only violate a lock with less transaction id. This approach can reduce possible false positive abort when exploit early lock violation. *DLV* use register and report [27] to maintain dependencies. Every transaction context stores an in dependency transaction(in_dn) number count to record how many transactions is the transaction depend on. The transaction also records an out transaction set(out_set) attribute record the transactions depend on it. When a transaction T speculative read from S. T registers dependency from S by adding T to out_set of S and increase in_dn of T by one. These steps are described by line 4 - 7 in function READ in Algorithm 4. A transaction cannot prepare if it's in_dn value is greater than 0, which means some in dependency transaction does not commit yet. If a transaction's in_dn value is less than 0, the transaction must abort because there is some in dependency abort cause a cascade abort. Algorithm 5 shows how to prepare a transaction. When a transaction commits, this transaction would traversal its out_set and decrease every transaction's in_dn by one, this is shown in line 6 - 11 of Algorithm 6 function. If a transaction aborts, it may cause a cascade abort. Line 2 of function CASCADE in Algorithm 6 shows the assign in_dn by a negative value when cascade abort.

D. Pseudocode Description

Algorithm 4 shows the execution phase of a transaction. Algorithm 5 shows the prepare phase of a transaction. Algorithm 6 shows the commit phase of a transaction. Algorithm 7 shows the speculation phase of a transaction.

Algorithm 3 transaction T lock a key T in M mode, use V violation

```
function LOCK($T, K, M, V$)
    wait if $\exists$ such $l \in lock\_table$,
    $CONFLICT(K, T.xid, l, L)$ and
    not $VIOLATABLE(K, T.xid, l, M, V)$
        if $T.xid > l.xid$ and wait timeout
            return nil                                  ▷ lock die error
        end if
    end wait
    for all such $l \in lock\_table$,
    $VIOLATABLE(K, T.xid, l, M, V)$ is True do
        $s \leftarrow$ transaction context retrieved by $l.xid$
        if $V == EARLY$ or ( $V = LATE$ and
        $l.lmode = WRITE$ and $L = READ$ ) then
            if $T.xid \notin s.out$ then
                $s.out\_set \leftarrow s.out \cup T.xid$
                $T.in \leftarrow T.in + 1$
            end if
        end if
    end for
    create lock $l$ if $l' \notin lock\_table$,
    $l'.xid \leftarrow T.xid, l'.key \leftarrow K, l'.mode \leftarrow M, l'.violate \leftarrow V$
    return lock $l'$
end function

function VIOLATABLE($T, K, L, V$)
    if $CONFLICT(T, l, K, M)$ and
    $l.violate \neq NONE$ then
        if $V = EARLY$ then
            return $T.xid > l.xid$
        else if $V = LATE$ then
            return $l.violate = LATE$
        end if
    end if
    return False
end function

function CONFLICT($T, L, K, M$)
    if $T.xid \neq l.xid$ and $K = l.key$ then
        return l.lmode = WRITE or M = WRITE
    end if
    return False
end function
```

V. Experiments and Evaluations

We develop a replicated distributed DBMS demo and evaluate the performance of *DLV*. As a comparison with *DLV*, we also implement S2PL wait die(S2PL) scheme, CLV optimize both violate at the 1st phase(CLV1P) and 2nd phase(CLV2P).

A. Experiments Setting

Our experiments performed on a cluster of 12 Aliyun ecs.g6.large server. Each server has 2 virtual CPU with 2.5GHz clock speed, 8GB RAM, runs Ubuntu 18.04. The data is partitioned by 4 shards, each shard has 3 replicas which is replicated across 3 AZs, which is located at Heyuan, Hangzhou and Huhehot. Every AZ has a full data copy of each shard. The internal network bandwidth of each AZ is 1Gbps. We

Algorithm 4 Execution phase of transaction T. Read and write a key

```
 1: function READ($T, K$)
 2:     $l \leftarrow LOCK(T, K, READ, T.violate)$
 3:     if $l$ is $nil$ then
 4:         return lock failed                    ▷ $T$ will abort later
 5:     else
 6:         $T.locks \leftarrow T.locks \cup l$
 7:         $v \leftarrow$ retrieve value of $K$ from the newest version
 8:         return $v$
 9:     end if
10: end function
```

```
 1: function WRITE($T, K, V$)
 2:     $l \leftarrow LOCK(T, K, READ, T.violate)$
 3:     if $l$ is $nil$ then
 4:         return lock failed                    ▷ $T$ will abort later
 5:     else
 6:         $T.locks \leftarrow T.locks \cup l$
 7:         assign $K$'s newest version by value $V$
 8:         add tuple to $T.write\_set$
 9:         return write success
10:     end if
11: end function
```

Algorithm 5 Prepare phase of transaction T

```
 1: function PREPARE($T$)
 2:     for all $l \in T.locks$ do
 3:         $l.violate = PARTIAL\_PREPARE$
 4:     end for
 5:     wait if $T.in > 0$
 6:     if $T.in < 0$ then
 7:         write prepare abort log
 8:         response $TM$ message $\{Prepare\ Abort\}$ ▷ cascade
 9:     else if $T.in = 0$ then
10:         write prepare commit log
11:         response $TM$ message $\{Prepare\ Commit\}$
12:     end if
13: end function
```

choose a modifies version TPCC and YCSB workload. All the transactions are distributed transactions. The TPCC data is sharded by the warehouse id. The Item table is replicated to all shards. Each transaction will retry after 3 seconds if it aborts for violation serializability. The evaluation both tested on both scattered (leader) mode and gathered (leader) mode. In gathered mode, all of the replica leaders are located in the same AZes. In scattered mode, the replica leaders are not located in the same AZes.

B. TPCC Performance Evaluation

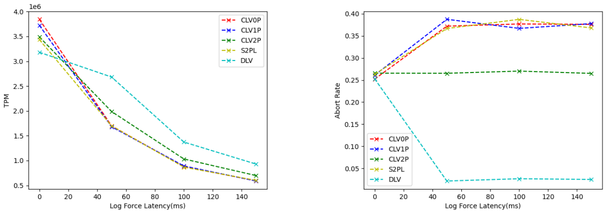
First we simulate log cost on different an LAN environment by adding log write latency. Figure 10 shows the throughput and abort rate of TPCC NewOrder procedure on 4 shards. Every shards replicated transactional log with user setting

Algorithm 6 Commit phase of transaction T, commit and (cascade)abort function

```
 1: function COMMIT($T$)
 2:     write commit log
 3:     for all $r \in T.write\_set$ do
 4:         make the version written by $T$ becomes commit
            version of tuple $t$
 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:         $t.in \leftarrow t.in - 1$                ▷ keep exactly once
11:         if $t.in = 0$ then
12:             report $t.in = 0$          ▷ stop waiting on this
            transaction, in line 2 of function PREPARE
13:         end if
14:     end for
15:     response $TM$ message $\{Commit\ ACK\}$
16: 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 $T$ from verion list
            of $r$
 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 this transaction,
            in line 2 of function PREPARE
12:     end for
13:     response $TM$ message $\{Abort\ ACK\}$
14: end function
```

latency time. The result shows that late violation can perform better when there is a large log replicaiton latency.



(a) NewOrder TPM (b) NewOrder abort

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

In the following experiments, we all run test on a real replication environment.

Figure 11 shows the NewOrder performance of when adding

Algorithm 7 Speculate phase.

Ready and *Speculate* works on *RM*.

TM call *Decide* send when *TM* collects all *RM*'s *Ready* message.

msgs is a collection of *Ready* message which TM receives from all the *RM*s.

Θ is a threshhold value to enable speculation.

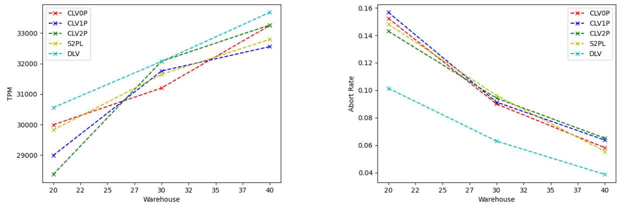
```
function READY($T$)
    response $TM$ message
    $\{Ready,\ wait \leftarrow T.wait,\ non\_da \leftarrow T.non\_da\}$
end function

function DECIDE($T, msgs$)
    if $\forall m \in msgs,\ m.non\_da$ is True and
    $\exists m \in msgs,\ m.wait > \Theta$ then
        send all $RM$s message $\{Speculate\}$
    end if
end function

function SPECULATE($T$)
    for $key \in T.read\_set$ do
        Unlock($T, key, Read$)
    end for
    for $key \in T.write\_set$ do
        if $key$ is $Write$ locked by $T$ then
            ModifyLock($T, key, ICommit$)
        end if
    end for
end function
```

terminal numbers of each node in the gathered mode and scattered mode.

Figure 12, Figure 13 shows the performance of different warehouse numbers.



(a) NewOrder TPM

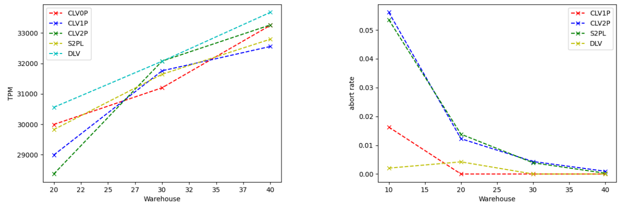
(b) NewOrder abort

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

C. YCSB Performance Evaluation

VI. Conclusion

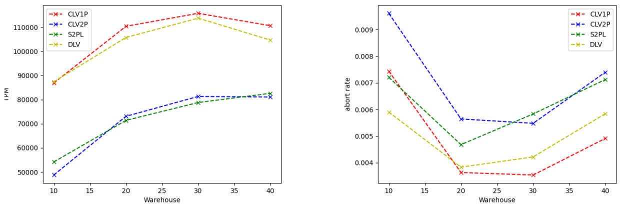
We extend CLV to distributed transaction and evaluate its performance on a geo-replicated environment. Our distributed version CLV, i.e. DLV, can dynamically decide to violate lock at the most suitable time. DLV merge many discrete waits at transaction running time into one final wait when commit. According to our evaluation, DLV can improve performance of contention workload for shortening critical



(a) NewOrder TPM

(b) NewOrder abort

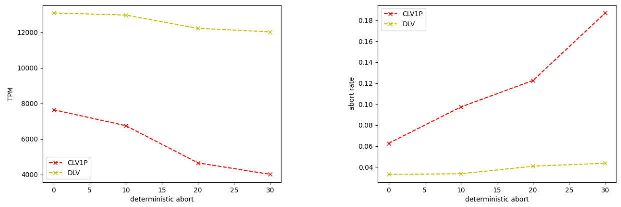
Fig. 12: throughput and abort rate of different warehouse number when running TPCC NewOrder transactions in gathered mode



(a) Payment TPM

(b) Payment abort

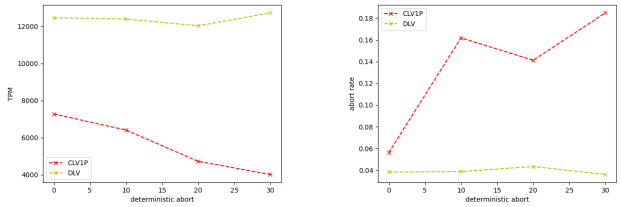
Fig. 13: throughput and abort rate of different warehouse number when running TPCC Payment transactions in gathered mode



(a) NewOrder TPM

(b) NewOrder abort

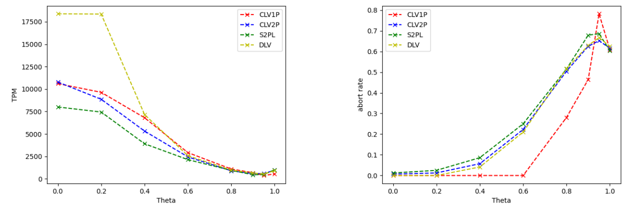
Fig. 14: throughput and abort rate of different possible cascade abort rate, when running TPCC NewOrder transactions in gathered mode



(a) NewOrder TPM

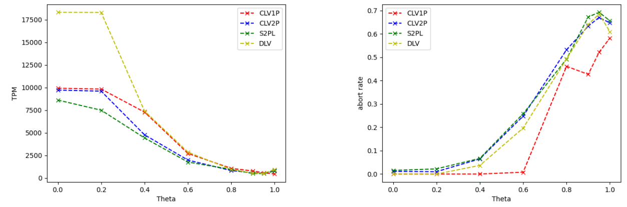
(b) NewOrder abort

Fig. 15: throughput and abort rate of different possible cascade abort rate, when running TPCC NewOrder transactions in scattered mode



(a) YCSB TPM (b) YCSB abort

Fig. 16: throughput and abort rate of different warehouse number when running YCSB transactions in gathered mode



(a) YCSB TPM (b) YCSB abort

Fig. 17: throughput and abort rate of different warehouse number when running YCSB transactions in scattered mode

path. DLV can adapt to different workloads. It minimize unnecessary dependency tracing cost and cascade abort penalty against previous work.

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, 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, SOSP 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

[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] 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

[13] ——, "Paxos made simple, fast, and byzantine," in *Procedings 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.

[14] R. van Renesse and D. Altinbuken, "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

[15] 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. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[16] "Voltdb," http://www.voltdb.com/.

[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] 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

[19] 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

[20] 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.

[21] "Mysql," https://www.mysql.com/.

[22] "Postgresql," https://www.postgresql.org/.

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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_per-akelarson_vldb2012.pdf

[28] Y. Raz, "The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers 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

[29] 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