Dynamic 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*—Many modern cloud distributed database management system (DBMS) scale horizontally by sharding its data on many nodes for scalability. Most of the databases in this category also build their transactional layer upon a replication layer for fault-tolerance. The replication layer uses a consensus protocol to reach consistency and implement automatic fault recovery. Many transactions take less time to enforce a serializable schedule than write its commit log to the replicated state machine(RSM). Thus, the replication layer and consensus protocol amplify transactions' lock duration. Exploit speculative techniques, such as controlled locked violation(CLV) and early lock release(ELR) can shorten lock duration and optimize transaction performance, especially handle a high degree of contention. However, these techniques, which are mainly focused on single-site database and failed to achieve both performance and correctness on a distributed environment. In this paper, we introduce dynamic lock violation(DLV) which we designed for the distributed transaction, especially which is on a geo-replication layer for fault-tolerance. DLV can violate lock at a proper time to get the best performance and achieve both performance and correctness.**

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

I. Introduction

Modern distributed database management system(DBMS) scale-out by partitioning data into multiple nodes, so it can run transactions on different servers in parallel and increase throughput. However, when the DBMS needs to access multiple partitions, it uses a coordination protocol to ensure a transaction's atomicity. Distributed transactions usually lead to significant performance degradation, mainly due to the following reasons [1]:

1. Coordinating to commit needs a chatty protocol (i.e., two-phase commit) which causes more message overhead;

2. The message transmission overlaps with the critical path of transaction commit, which worsens the contention among transactions.

Furthermore, distributed DBMS in this category also use a replication layer below the transaction layer to guarantee fault tolerance. Transactional layer uses a specific concurrency control(CC) scheme to enforce a serializable schedule and a distributed commit protocol if transaction access multiple shards. The replicated layer often uses a Paxos-like consensus protocol to guarantee data replicas consistency. Typical implementation optimized replication performance by splitting data into very small chunks and build replicated state machines on them. Although building multiple replicated state machine improve replication performance, it makes distributed transaction even more inevitable, as distributed transactions can more likely occur on different chunks of data.

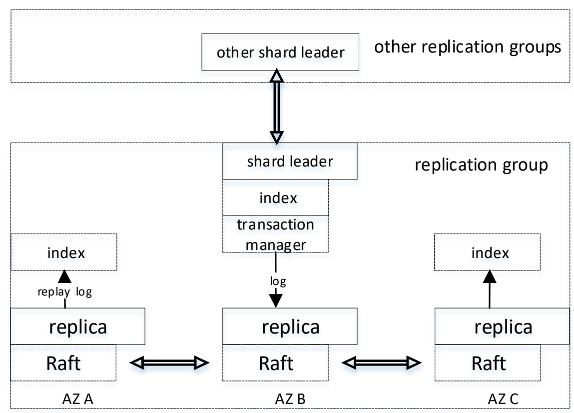


Fig. 1: distributed and replicated DBMS Architecture

Figure 1 presents the typical architecture. First, the database partitions its data with many shards to scale. Second, Each shard works on a replication layer which replicates data in several availability zones(AZ) [2] for fault tolerance. Between different available zones, the replication layer uses a consensus protocol to shield consistency. This architecture uses a chatty message protocol fails to scale high contention workload, as much previous work has discussed [1] [3] [4]. But this architecture supports a wide range of transaction models and runs well on many workloads. Most industry distributed DBMS choose this two-layer architecture, such as Google Spanner [5] [6], NuoDB [7], CockroachDB [8], TiDB [9].

The distributed coordination and replication protocol enlarge the timespan of the critical path and amplified contention cost. We focus on distributed DBMS which uses locking scheme and coordination protocol on a replicated layer, especially who running transaction processing on geo-replicated layer. Figure 2 shows the message flow of a distributed transaction using S2PL and 2PC works on a WAN replication

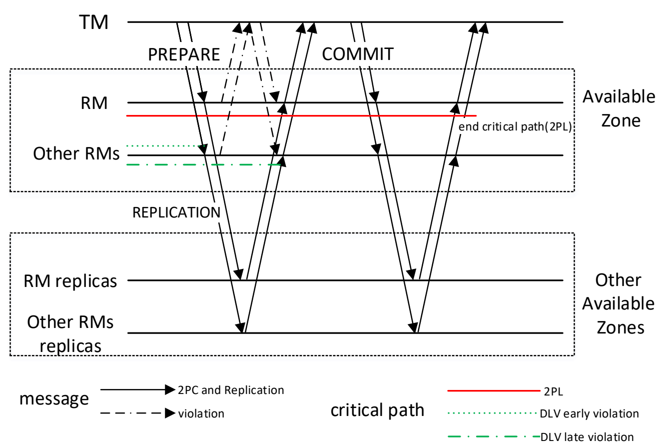


Fig. 2: The message flow and lock hoding critical path of DB who uses 2PL(or with DLV) and 2PC works on replication layer. The dash arrow line message is introduced by DLV. The red lines show the critical path of S2PL and the green dash line shows the critical path of DLV.

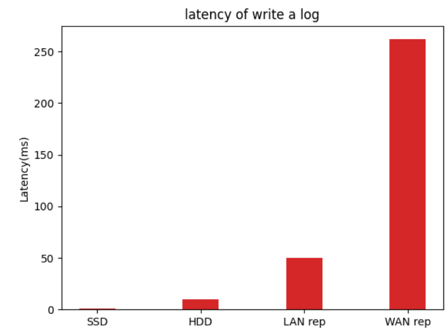


Fig. 3: log write latency in different environment

layer. When a transaction requests its commit, *TM(transaction manager)* issues a prepare message to each *RM(resource manager*). *RM* replicates its decision result(prepare commit or prepare abort) to other replicas. *RM* responses its decision to *TM* after reach a consensus to guarantee fault tolerance. *TM* then collects all the *RMs* decisions. And then it broadcasts the final result(commit or abort) to all *RMs* [1] . Once a *RM* receives the final result(commit or abort) from *TM*, it would replicate these result to other replicas. After the consensus of the commit(or abort) log has reached, *RM* can release the locks that it had retained since it first accesses the specified record.

We depict the lock duration with the red line in Figure 2 when this transaction commit. The lock duration covers many message round trips including those over WAN. Such a commit and replication protocol will severely impair the concurrency when confronted with a high degree of contention.

Previous works used speculative techniques, such as early lock release(ELR) [10], controlled lock violation (CLV) [11] to optimize transaction processing using locking. These techniques can be extended to a distributed environment to improve concurrency. The two-layer architecture shares the same bottleneck with single node DBMS on forcing transaction log and faces even worse conditions. Figure 3 shows the log write latency of different environment(TODO... experiments setting).

Distributed transactions work on geo-replicated DBMS need more time to write a log than non-distributed and non-replicated one. However, to apply these techniques on the distributed environment is complex. There are some design considerations to choose from. To combine the two-phase commit protocol(2PL), violate(or release) lock at which phase transaction? As previous work addressed [11], violating lock at the first phase can exploit more concurrency but takes more dependency tracing cost and cascade abort cost. Violating lock at the second phase need to maintain less dependencies and get less cascade abort rate. But it may not get better concurrency. Transaction models, interactive or one-shot transactions, may have different message flow, how different transaction types could benefit from these techniques? Not all the transactions can benefit from CLV or ELR, little conflicts workloads are such cases.

In this paper, we propose a dynamic lock violation(DLV) to boost the locking-based distributed DBMS, especially who are geo-replicated with a high commit latency. DLV decides best lock violation time by runtime statistic information. It maintains less commit dependencies and bears less cascade abort penalty compares previous implementation [11].

This section is the overall introduction of this paper. Section II is a review of related work. Section III presents a strict schedule is not necessary and hurt the performance of distributed and replicated DBMS. Section IV introduces DLV's implementation. Section V evaluates DLV and compare it with previous work. Section VI draws the final conclusion of this paper.

II. Relate Work

This section introduces the related work of this paper.

A. Distributed Transaction on a Replicated Layer

Recently, there are many scalable DBMS arisen in both academia and industry. Most of the systems in this category supports distributed query processing and replicate data across several data center geo-located in different areas for fault tolerance.

A fault-tolerant database relies on state-machine replication(SMR) log to avoid single point failure. SMR needs to use a consensus protocol to enforce the same order of different replicas. Paxos [12] [13] is the most well-known consensus protocol. Paxos use two messages round trip to accept a value, one roundtrip for choosing a proposal and another to propose the value. Multidecree Paxos [14] elects a leader as the only proposer to eliminate Paxos first message roundtrip during normal processing. Raft [15] is a similar consensus protocol

[1]Depend on varies of implementations, TM can choose to write its log or not.

to Paxos, which is designed for understandability. Consensus introduces significant overhead for its lots of message round trips and heavy network traffic.

Google spanner [5] [6] is a geo-replicated and shared-nothing DBMS that uses hardware clock for timestamp generation. VoltDB [16] is a main memory database who runs single threaded execution per partition. [1] use a deterministic transaction model, Calvin can commit distributed transaction without coordination protocol. VolteDB and Calvin, by using deterministic scheduling, they can use active replication and replicate transaction input rather than transaction effect. Tapir [3] use inconsistent replication layer and build consistent transaction on it to guarantee user level consistent. Janus [4] gets fewer wide-area round trips by consolidating the concurrency control and consensus and use deterministic serializable graph tracing to commit transactions under conflicts. Tapir and Janus, which benefit from their codesign of transaction and replication layer, can commit a distributed transaction in only one wide-area round trips.

B. Locking Concurrency Control

DBMS use concurrency control(CC) to calculate a serializable schedule for concurrent transactions. Two-phase locking(2PL) is the most widely used CC scheme. As a pessimistic method, 2PL assumes that it is likely that transactions will conflict. 2PL uses a lock to enforce the order of conflicting transactions. Strict 2PL(S2PL), in additional to 2PL, preserves its lock until a transaction's termination. S2PL guarantees transaction's recoverability but a 2PL cannot. For enabling a simple recovery algorithm, most locking based databases choose S2PL. When extending S2PL to distributed databases, S2PL can take more lock holding time on its commit critical path for many message round trips.

2PL scheme implementation varies on how they process deadlock. In *no-wait* [17] policy, a transaction would imediantle abort if it fails to lock a record. Previous work has proved it is the most scalable technique against other locking schemes, even in diestirbuted environment [18] [17]. Another policy is *wait-die* [19] which is similar to *no-wait*. Transactions avoid false-positive aborting base on their start timestamps when database using 2PL *wait-die*. In *deadlock detection* [20], transactions can wait for each other without controlling. The transaction would abort only a deadlock occur. *Deadlock detection* detect deadlock by explicitly tracing wait-for graph and testing circles. Many traditional single node database [21] [22] use *deadlock detection* technique because it has no false positive abort. Deadlock detection on distributed DBMS is costly since it requires substantial network messages to identify circles.

C. Exploit Speculation and Lock Violation

Exploit speculation is not a new invention. Similar approaches have been introduced by many previous works. Early lock release (ELR) [23] [24] [25] [10] [26] shares the same idea. ELR can release transactions' lock without waiting for commit record flushed to disk. DeWitt et al. [23] firstly described ELR. Soisalon-Soininen et al. [24] proved that the correctness of ELR. Johnson et al. [25] and [10] evaluated the performance improvement made by ELR. Kimura et al. [10] [25] also address the weakness of previous ELR implementation [23] can produce wrong results for read-only transactions. Previous work exploits speculation mostly designed for single machine database system [24] [25] [10]. Jones et al. [27] use a restirct transaction model [28] implement sepculation. Control lock violation(CLV) [11] achieve the same performance as ELR but with a simple and general implementation. CLV can apply to distributed databases and optimize both phases of two-phase commit. CLV can use a "register and report approach(RARA)" [29] to implement its dependency. RARA work well on a single-site database. When RARA is used to process a distributed transaction, the dependency tracing may be complex and costly. Many cascade abort on distributed transaction also carries a performance downgrade.

III. Beyond Strict Schedule And Lock Violation

In the following subsections, we descript our preliminaries and assumptions, the basic rule to keep transactions' correctness when violating lock.

A. Preliminaries and Assumptions

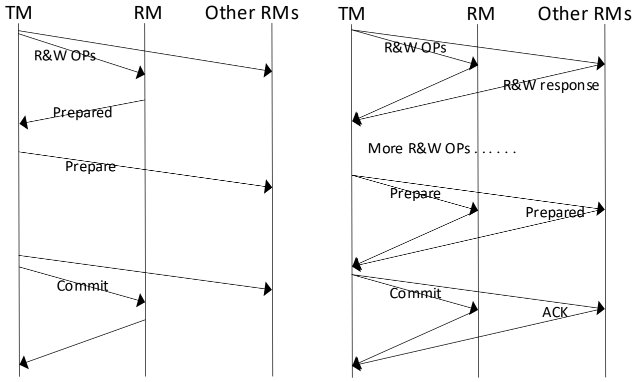
The DBMS shards its data by database primary keys. Each shard replicates its physiological logs across different AZs for fault-tolerance. The physiological logs record the row-level write operations of each transaction. The replicated layer use Raft protocol to maintain consensus of the log order. Other replication protocols may also work. Only the replica leader processes the transactional operations. Both one-shot and interactive transactions are supported.

Figure 4 shows the message flow of distributed commit of these two different type of transaction models. As shown in these figures, The interactive transaction needs more message roundtrips compared to the one-shot one. These two types of transaction model employ different message flow. Lock violation can comply with these different transaction models, which we will explain subsequently.

B. Conceptions and Definitions

Before we develop our method, we firstly review the formal definition and conception of transaction processing, which can be found in previous work [19].

1) Distributed Transaction and History: Given a distributed database has n sites $R = \{r_1, r_2, ...r_n\}$. Transaction T_i runs on m sites $S = \{s_1, s_2, ...s_m\} \subseteq R, 1 \leq n \leq m$. Transaction T_i runs a series of of operations o_i. o_i can be read or write operation, or command operations which include prepare commit(abort), abort or commit. $r_i[x]$ donates transaction T_i reads record x, $w_i[x]$ donates transaction T_i writes record x. c_i, a_i, p_i^c, p_i^a mean transaction T_i commits, aborts, prepares commit or prepare abort respectively. We call T_i's operations on a site s_u as T_i's *partial transaction* on s_u. The transaction history is a collection $H = \{h_1, h_2, ..., h_n\}$, in which $h_u(1 \leq u \leq n) = \Pi_u(H)$ is the local history on site s_u.



(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

$\Pi_u(H)$ is H's projection on site s_u. For any projected history $h_u(1 \leq u \leq m)$, h_u is a collection of operations requests by many transactions.

2) Deterministic and Non-deterministic Abort: Transaction abort due to many reasons, they can be categories:

1. User requested abort, including abort from a user's program logic such as access on non-exists records;

2.Violation of serializability;

3.Database node crash for failure.

We call the first two abort *deterministic abort* and the last one *non-deterministic abort*.

3) Transaction Dependencies: Transaction T_j has a *commit dependency* on transaciton T_i, written as $T_j \rightarrow T_i$, if T_j can commit only if T_i commit. If T_j aborts, T_j need also abort. There are there kinds of dependencies, *wr-dependency*, *ww-dependency* and *rw-dependency*. If in any local history h, transaction T_j read T_i's write on x, we call this dependency *write-read(wr) dependency* and denote it by $w_i[x] \rightarrow r_j[x]$. Similarly if T_j overwrite T_i's write on x, this is *write-write(ww) dependency* and written as $w_i[x] \rightarrow w_j[x]$. And if T_i reads x is precede T_j write on x and these two operation has direct conflict(this one seems necessary???), it is a *read-write(rw) dependency* and recorded as $r_i[x] \rightarrow w_j[x]$. A transaction T_j *speculatively access* a record x, if there is another transaction T_i, T_j has a dependency on T_i and T_i has not committed. We write this *danger dependency* as $w_j[x] \rightarrow_s r_i[x]$, $w_j[x] \rightarrow_s w_i[x]$, $r_j[x] \rightarrow_s w_i[x]$. We also write $T_j \rightarrow_s T_i$ to indicate transaction T_i has a commit *danger dependency* on T_j.

4) Strictness: Traditional transaction schedulers choose strictness [30] to simplify implementation and avoid expensive transaction recovery cost. Strictness implies that a transaction cannot read or overwrite a previous write by another transaction which has not ended yet. For a locking base concurrency control scheme, the lock will hold until the transaction end, namely strict two-phase locking(S2PL). Strictness is not necessary to produce a correct schedule.

C. Strict Scheduler is Too "Strict" for Correctness

In schedule H_1 of Figure 5, there are 3 transactions working 3 shards, s_1, s_2, s_3. There are dependencies, $r_1[x] \rightarrow w_3[x]$, $w_1[x] \rightarrow r_2[y]$, $r_2[y] \rightarrow w_3[x]$ and there is $T_1 \rightarrow T_2 \rightarrow T_3$. There is no circle in this dependency graph and the schedule is serializable and strict. There is no needs to maintain dependencies explicitly.

Figure 6 shows an example of a non-strict but correct schedule H_2. There are three records x, y, z, located at shard s_1, s_2, s_3. Transaction T_1 execute write y, write x. Transaction T_2 read T_1's write on x before T_1 commits. Transaction T_3 overwrite T_2's write ahead T_2's commit. The history

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

in which,

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

is not a strict history, but it's a serializable history.

Both of H_1 and H_2 are serializable equal with the serial schedule, T_1 T_2 T_3. Scheule H_1 and H_2 are correct.

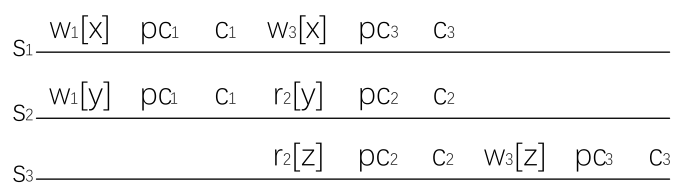


Fig. 5: A strict and serializabile schedule H_1

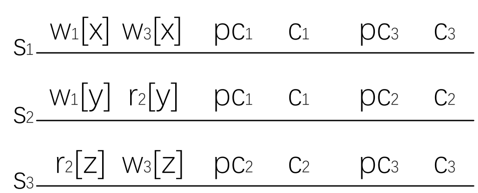


Fig. 6: A non-strict but serializabile schedule H_2

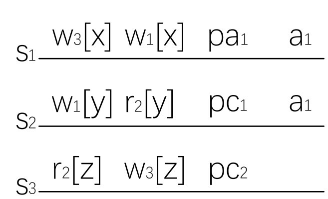


Fig. 7: Schedule H_3, T_1 abort due to non-serializabile

To get better concurrency, the scheduler can produce schedule like H_2 in Figure 6. Suppose locking schedule H_2, then transaction T_1 must release its locks or make its locks can be violative before it knows its commit decision. A transaction

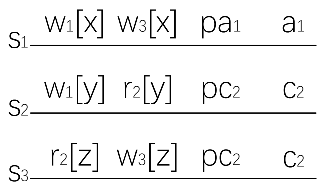


Fig. 8: Schedule H_4, T_2 commit ahead T_1, non-recoverable anomaly

commits by no means an operation in a flash but progress that needs take lots of time, especially when it is a distributed commit on an RSM. Strictness scheduler on a distributed and replicated DBMS has a long critical path. Our basic idea is to develop a serializable but non-strict correct scheduler for distributed transactions and shorten the critical path of committing a transaction. A single node transaction can exploit log order to maintain logical dependencies because dependent transactions write their logs orderly [23] [10]. When we extended the non-strict locking scheme to distributed transaction, transaction dependency maintaining is more complex. The serializable scheduler is a correct one if it can prevent abort. But when there are aborted transactions, it does not. To produce the correct log, the schedule must be both commit serializable and recoverable [31]. The scheduler need to maintain commit dependencies to guarantee serializability and recoverability when exploiting non-strictness.

D. Lock Violating Rules and Dependency Tracing

Lock violation scheduler must follow some rules to preserve correctness. First, a lock violation scheduler should create serializable schedules. Consider a schecule H_3 in Figure 7 as an example. There are danger dependencies in H_3:

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

There is a circle $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$. This schedule is not serializable. Transaction T_2 read from uncommitted transaction T_3. When T_1 abort for non-serializable, T_2 must cascade abort to avoid anomaly. Assume a lock violation scheduler creates schedule H_3. Not formally discussing, a lock violation operation is just as a transaction releases its lock and another later transaction acquire this lock. Then the schedule H_3 failed to comply with 2PL's two-phase principle.

For a traditional locking CC scheme, transaction T_2 needs to wait for T_1's release its lock until T_1 commit. This situation is like Figure 5. Lock violation scheduler must guarantee the dependency graph has no circles. A transaction must trace commit dependencies if it violates locks of another conflict transaction operation.

Secondly, a schedule generated by lock violation must also be recoverable. In schedule H_4 of Figure 8, T_2 read from uncommitted transaction T_1's write. There is a danger dependency, $w_1[y] \rightarrow_s r_2[y]$ and T_2 's commit is ahead T_1's commit. Schedule H_4 is non-recoverable. If T_1 abort, then T_2 will return an error y value. The scheduler requires to maintain dependencies to preserve the recoverability of the schedule.

Additionally, there are several design considerations and choices arisen except the correctness. Can lock violation easily adapt to different transaction models and 2PC? Violating lock at the first phase of 2PC may be superior to violating lock at the second phase because it can shorten the more critical path length. But the first phase violation may bear more cascade aborts. Cascade abort transactions are totally useless work. DLV permits lock violation at two points in the timeline of running transactions. We call them *early lock violation* and *late lock violation*.

Suppose H is a schedule which is created by lock violation, then H is extended by adding the lock, unlock and lock violation operations. We write H by:

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

In H, x, y are not the same records. There are following lock/unlock/violation operations, transaction T_i locks record x; T_j violating T_i's lock on x; T_i locks record y; T_i release locks on x. If there is such $vl_j[x]$ and $l_i[y]$ operations in schedule H, then transaction T_j is *eraly lock violate* T_i's lock on x; otherwise, this is *late lock violation*. A scheduler uses *early lock violation* may cause non-serializable schedule if it runs without waiting dependencies. Then DLV needs maintains all wr, ww and rw dependencies after violating locks and guarantee the dependency graph of the schedule is acyclic if using *earyly lock violation*. On the contrary, *late lock violation* cannot make an acyclic dependency graph to become a cyclic one by adding any dependency edges. This can be proved by formulating *late lock violation* as 2PL proving.

Assume that there is a *wr-dependency* from T_i to T_j. T_j, which can be written as $w_i[x] \rightarrow r_j[x]$. T_j cannot commit if T_i has not committed. Traditional S2PL schedule can guarantee this by release locking when T_i commit. Lock violating violates locking rule and T_j can read T_i's write on x before T_i commits. In a lock violation schedule case, transactions must trace dependencies and commit as dependency orders. Composite with 2PC protocol, we have the following rules:

1) T_j prepares only if T_i commit;
2) T_j commits only if T_i commit;
3) If T_i aborts, T_j must also abort

By tracing dependencies after violating a lock, DLV schedule achieves both serializability and recoverability.

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,

$$H_5 = w_1[x]w_1[y]r_2[x]w_2[y]a_1a_2$$

If transaction T_1 abort, this cause 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.

$$exp(H_5) = w_1[x]w_1[y]r_2[x]w_2[y]\ w_1^-[y]w_1^-[x]c_1w_2^-[y]c_2$$

Suppose the initial value of records x and v of are both 0. The value of records x, y and the undo log formatted after executing every operations in $exp(H)$ is shown in Table I. Finally, after the execution of this schedule, both transaction T_1 and T_2 aborts. The value of y is 1, which the correct result should be the initial value 0.

operations	x	y	undo
$w_1[x=1]$	1	0	x=0
$w_1[y=1]$	1	1	y=0
$r_2[x]$	1	1	
$w_2[y=2]$	1	2	y=1
$w_1^-[y=0]$	1	0	
$w_1^-[x=0]$	0	0	
c_1	0	0	
$w_2^-[y=1]$	0	1	
c_2	0	1	

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

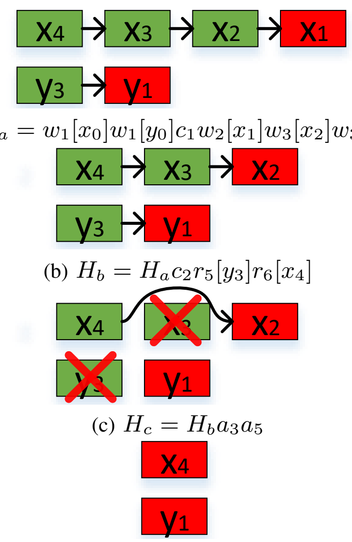
To tackle this anomaly, recovery must use a more complex algorithm such as SOT [31]. For schedule H_5, a correct recovery expandation may be:

$$exp^*(H_5) = w_1[x]w_1[y]r_2[x]w_2[y]w_2^-[y]c_2w_1^-[y]w_1^-[x]c_1$$

The schdeuler must recovery transaction by the reserve order of write operation. If x and y is on the same database node and use *late lock violation*, the recovery of a transaction is simple. Because there is no partial failure, the transaction would commit in log order. No additional work is needed when system recovery using traditional Aries algorithm [32].

However, if x and y are not located at the same node, this undo operation order is hard to accomplish because of partial failure. When using *early lock violation*, there are similar problems since transaction recovery must also undo transactional operations by reserve order. 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 transactions would write a little data except the bulk loading ones and 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. Speculation versions are always stored in main memory and needs no persistence. If a transaction would abort, it only needs to remove its write versions from 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 $w_6[x] \rightarrow w_4[x]$. The abort of T_4 does not cause T_6 cascade abort.



(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

B. Dynamic Decide Early or Late Violation

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.

We 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 4. 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*. We define a partial transaction which enters prepared commit status but failed to commit finally as partial prepare. DLV calculates partial prepare rate at a period to decide which violation strategy to choose. We define that a transaction T running at time τ would access a collection of shards $S_1, S_2, ..., S_n$. The message round trip time(RTT) from T's TM to RM on shard S_i is RTT_i. In a window period time from $\tau - \delta$ to τ, there is N_p partial preapres of total N_t partial transactions. DLV would tests the following conditions where Θ is is a constant coefficient.

$$N_p/N_t < \Theta * max(RTT_i), i \in \{1, 2, .., n\}$$

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

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.

DLV use register and report [29] 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 1. 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 2 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 3 function. If a transaction aborts, it may cause a cascade abort. Line 2 of function CASCADE in Algorithm 3 shows the assign in_dn by a negative value when cascade abort.

D. Pseudocode Description

Algorithm 1 shows the execution phase of a transaction. Algorithm 2 shows the prepare phase of a transaction. Algorithm 3 shows the commit phase of a transaction. Algorithm 4 shows the speculation phase of a transaction.

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).

Algorithm 1 Execution phase of transaction T. Read and write a key

1: **function** READ(T, key)
2:     $newest\_version \leftarrow Head(Tuple(key).version\_list)$
3:     **if** $newest\_version$ is created by transaction S **and** key is ICommit locked by S **then**
4:         **if** $T \notin S.out\_set$ **then**
5:             $S.out\_set \leftarrow S.out\_set \cup T$
6:             $T.in\_dn \leftarrow T.in\_dn + 1$
7:         **end if**
8:     **end if**
9:     **if** key is *write* locked by transaction S **then**
10:         $S.wait \leftarrow S.wait + 1$
11:         wait lock till die
12:         **if** die **then**
13:             $T.no\_da \leftarrow$ **False**
14:             **return** die error.
15:         **end if**
16:     **end if**
17:     **if** key is *IAbort* locked by transaction S **then**
18:         wait lock this lock released
19:     **end if**
20:     Lock($T, key, Read$)
21:     **return** key's value.
22: **end function**

1: **function** WRITE($T, key, value$)
2:     **if** key is *read* or *write* locked **then**
3:         **if** key is *write* locked by transaction S **then**
4:             $S.wait \leftarrow S.wait + 1$
5:         **end if**
6:         wait lock till die
7:         **if** die **then**
8:             $T.no\_da \leftarrow$ **False**
9:             **return** die error.
10:         **end if**
11:     **end if**
12:     Lock($T, key, Write$)
13:     add a new version of key's tuple, assign $value$
14: **end function**

Algorithm 2 Prepare phase of transaction T

1: **function** PREPARE(T)
2:     wait if $T.in\_dn > 0$
3:     **if** $T.in\_dn < 0$ **then**
4:         response *TM* message $\{$*Prepare Abort*$\}$
5:     **else if** $T.in\_dn = 0$ **then**
6:         response *TM* message $\{$*Prepare Commit*$\}$ or $\{$*Prepare Abort*$\}$
7:     **end if**
8: **end function**

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

Algorithm 3 Commit phase of transaction T, commit and (cascade)abort function

function COMMIT(T)
    garbage collect old version in $Tuple(key).version\_list$
    for $key \in T.write\_set \cup T.read\_set$ **do**
        Unlock($T, key, Read/Write$)
    end for
    for $T_{out} \in T.out\_set$ **do**
        $T_{out}.in\_dn \leftarrow T_{out}.in\_dn - 1$ ▷ keep exactly once
        if **then**$T_{out}.in\_dn = 0$
            report $T_{out}.in\_dn = 0$ ▷ stop waiting on function PREPARE line 2
        end if
    end for
    response *TM* message {*Commit ACK*}
end function

function ABORT(T)
    call CASCADE(T)
    for $key \in T.write\_set \cup T.read\_set$ **do**
        Unlock($T, key, Read/Write$)
    end for
    response *TM* message {*Abort ACK*}
end function

function CASCADE(T)
    $T.in\_dn \leftarrow -\infty$
    for $key \in T.write\_set$ **do**
        if key is $ICommit$ locked by T **then**
            ModifyLock($T, key, IAbort$)
        end if
        for $version \in Tuple(key).version\_list$ **do**
            if $version$ is created by T **then**
                remove $version$ from list
                break
            end if
        end for
    end for
    for $T_{out} \in T.out\_set$ **do**
        call CASCADE(T_{out})
    end for
end function

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 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.

Algorithm 4 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

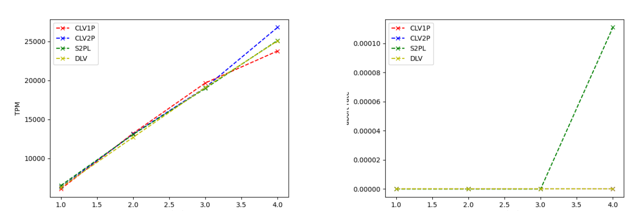
function SPECULATE(T)
    for $key \in T.read\_set$ **do**
        Unlock($T, key, Read$)
    end for
    for $key \in T.write\_set$ **do**
        if key is $Write$ locked by T **then**
            ModifyLock($T, key, ICommit$)
        end if
    end for
end function

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

Figure 10 shows the NewOrder performance of when adding terminal numbers of each node in the gathered mode and scattered mode.

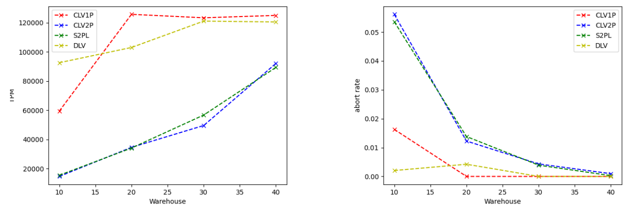
Figure 11, Figure 12 shows the performance of different warehouse numbers.



(a) NewOrder TPM

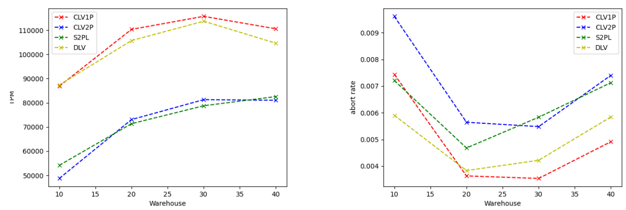
(b) NewOrder abort

Fig. 10: throughput and abort rate of different terminal number of each node when running TPCC NewOrder transactions in gathered mode(TODO .. need more terminals, 4 is too small)



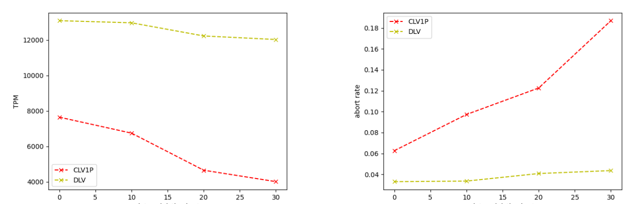
(a) NewOrder TPM (b) NewOrder abort

Fig. 11: throughput and abort rate of different warehouse number when running TPCC NewOrder transactions in gathered mode



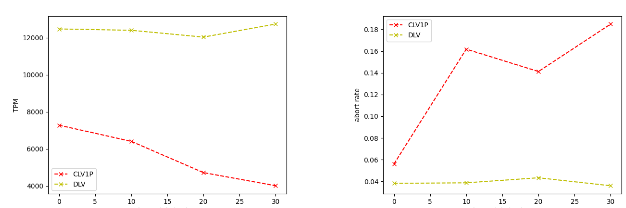
(a) Payment TPM (b) Payment abort

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



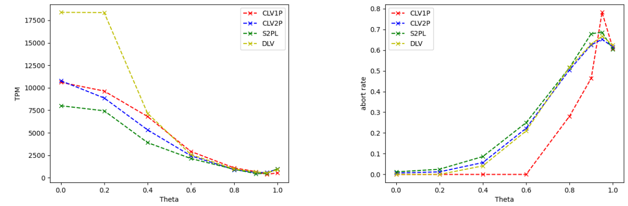
(a) NewOrder TPM (b) NewOrder abort

Fig. 13: throughput and abort rate of different possible cascade abort rate, when running TPCC NewOrder transactions in gathered mode



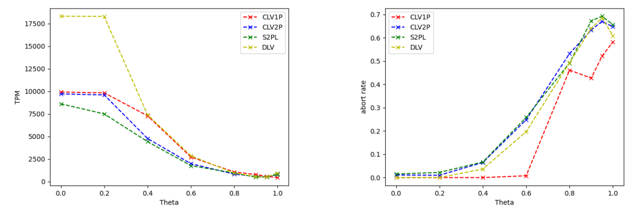
(a) NewOrder TPM (b) NewOrder abort

Fig. 14: 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. 15: throughput and abort rate of different warehouse number when running YCSB transactions in gathered mode



(a) YCSB TPM (b) YCSB abort

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

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 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] E. P. C. Jones, D. J. Abadi, and S. Madden, "Low overhead concurrency control for partitioned main memory databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, 2010, pp. 603–614. [Online]. Available: https://doi.org/10.1145/1807167.1807233

[28] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: a high-performance, distributed main memory transaction processing system," *PVLDB*, vol. 1, no. 2, pp. 1496–1499, 2008. [Online]. Available: http://www.vldb.org/pvldb/1/1454211.pdf

[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_per-akelarson_vldb2012.pdf

[30] 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

[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

[32] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, 1992. [Online]. Available: https://doi.org/10.1145/128765.128770