Dynamic Lock Violation for Cloud 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 OLTP databases scale horizontally by sharding its data on many nodes for scalability. Databases also build their transactional layer upon a geo-replication layer for fault-tolerance. The replication layer use a consensus protocol to reach consistency and implement automatic fault recovery. A transaction takes less time to enforce a serializable schedule than write its commit log to replicated state machine(RSM). Thus, the lock duration is amplified by the replication layer. Exploit speculative techniques, such as controlled locked violation and early lock release can shorten lock duration, can optimize transaction performance, especially handle a high degree of contention. However, these techniques, mainly focus on single site database and failed to scale achieve both performance and correctness on distributed environment. In this paper, we introduce dynamic lock violation(DLV) which isdesigned for distributed transaction. It can violate lock at the right moment by statistic locking and aborting information to get the best performance.**

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

I. Introduction

Modern cloud distributed database scale-out by partitioning data into multiple nodes, so it can run transactions on different servers in parallel and increase throughput. However, when the database needs to access multiple partitions, it uses a coordinate 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, modern distributed databases also use a replication layer below the transaction layer to guarantee fault tolerance. Transactional layer use a specific concurrency control(CC) scheme to enforce a serializable schedule and a distributed commit protocol if transaction access multiple shards. Optimistic CC scales well when there is little contention but suffers high abort rate when it dealing with workload of high degrees of contention. Pessimistic CC has a lower abort rate, but it endures overheads of blocking. In fact, CC matters little on performance if no conflicts and no most CC schemes failed to handle high contention [2] [3]. 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. Many transactional database systems choose this two-layer architecture, such as Google Spanner [4] [5], NuoDB [6], CockroachDB [7], TiDB [8]. The distributed commit and replication coordination protocol

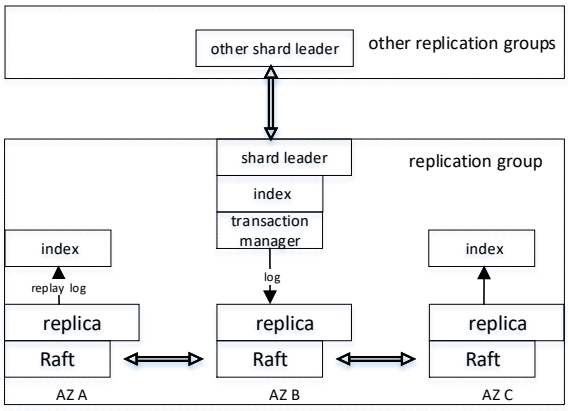


Fig. 1: Distributed and Replicated Database Architecture

using chatty manner protocol enlarge the timespan of the critical path and amplified contention cost. 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 replicated data in several availability zones(AZ) [9] for high availability. Between different available zones, the replication layer uses a consensus protocol to shield consistency.

We focus on a on cloud distributed database using locking scheme and coordinater 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 replica-

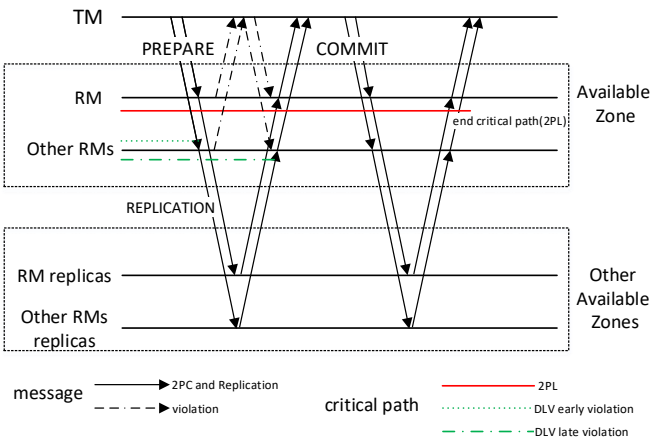


Fig. 2: The message flow and lock hoding critical path of DB who uses 2PL(or with DLV) and 2PC works on replication layer. The dash arrow line message is introduced by DLV. The red lines shows the critical path of S2PL and the green dash line shows the critical path of DLV.

tion layer. This architecture uses a chatty message protocol fails to scale high contention workload, as much previous work has discussed. When a transaction requests its commit, *TM(transaction manager)* issues a prepare message to each *RM(resource manager*). *RM* replicates its result to other replicas to make it fault tolerant. *RM* responses its decision to *TM* after reach a consensus to guarantee fault tolerance. *TM* then collects all the *RMs* decisions. And then it issues either a commit or abort decision to *TM*'s replicas and broadcast the final result to all *RMs*. Once a *RM* receives the final result from *TM*, it can release the locks that it had retained since it first accesses the specified tuple.

We depict the lock duration with the red line in Figure 2 when commit. The lock duration covers many message round trips including those over WAN. Locking for generating a serializable order of concurrency operations is can be overly lengthy to commit contented transactions. 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 technique can be extended to distributed environment to increase concurrency. Two layer architecture shares same bottleneck on force transaction log and faces even worse conditions. Distributed transactions work on geo-replicated database need more time to write a log than non-distributed and non-replicated one. However, extended these technique on distributed environment is complex. There are some design considerations. To combine two phase commit protocol, 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 may not exploit better concurrency but need maintain less dependency and get less cascade abort rate. Transaction model, 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 databases on cload, especially who are geo-replicated with a high commit latency. DLV 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 databases. Section **??** introduces our speculative implementation. Section V evaluates our experiment results. Section VI draws the conclusion of this paper.

II. Relate Work

This section introduces related work of this paper.

A. Distributed Transaciton on Replicated Layer

Recently, there are many scalable DBMS arised in both academia and industry. Most of the systems in this category supports distributed query processing and replicate data accross serveral data center geo-localted in diffferent 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 to Paxos, which is designed for understandability. Consensus introduces significant overhead for its lots of message round trips and heavy network traffic.

Google spanner [4] [5] 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 useing deterministic scheduling, they can use active replication to replicate transaction input rather than transaction effect. Tapir [17] and

Although our work relies on Raft replication protocol, other replication protocols are also adequate since replication is an orthogonal system to database transaction processing.

B. Locking Concurrency Control

Database use concurrency control(CC) to calculate a concurrent 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 schedule cannot. For enabling a simple recovery algorithm, most locking based databases choose S2PL. When extending S2PL to distributed databases, S2PL can take more time blocking on its commit critical path for additional message round trips.

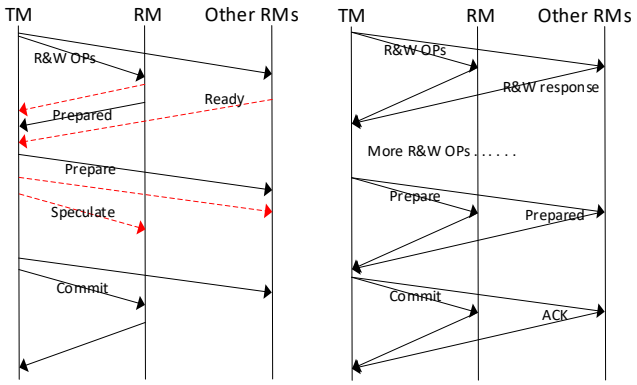
2PL protocol implementation varies on how to process deadlock. In *no-wait* [3] policy, a transaction would imediantle abort the transction if it fails to lock record. Previous work has prove it is the most scalabale technique to handle locking scheme, even in diestirbuted environment [18] [3]. Another policy is *wait-die* [19] which is similar to *no-wait*. Transactions avoid to abort base on their start timestamps comparation when database using 2PL *wait-die*. In *deadlock detection* [20], transactions can wait for each other without controlling. Transaction would abort only if there is a deadlock actually. *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 database requires substential network messages to indentify circles and is costly.

C. Exploit Speculation and Lock Violation

Exploit speculation is not a new idea. Similar approaches have been introduced by many previous works. Early lock release (ELR) [23] [24] [25] [10] [26] shares the same idea with speculative approach. ELR can release transactions' lock without waiting for commit record flushed to disk. DeWitt et al. [23] firstly described ELR without implementation. 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 distributed transaction with no contention, the dependency tracing may be complex and costly. More cascade abort rates on distributed transaction also lead more false positive violations and carry a performance penalty.

D. Interactive Transaction and One-Shot Transaction

Figure 3 shows the message flow of committing these two different type of transactions. As shown in these figures, The interactive transaction needs more message roundtrips compared to one-shot one. These two types of transaction model employ different speculative timing, which we will explain subsequently.



(a) one-shot transaction, the red dotted lines show message added by *DS*, when RM receive a *Speculate* message, transaction can prevent deterministic abort.

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

Fig. 3: One-shot distributed transaction and Interactive distributed transaction commit message flow

III. Beyond Strict Schedule And Lock Violation

.

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

A. Preliminaries And Assumptions

The database shards its data by database primary keys. Each shard replicated 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 sustain consensus of log order. Other replication protocols may also work. Only replica leader processes the transactional operations. Both one-shot and interactive transactions are supported.

B. Strict Scheduler Is Too "Strict" For Correctness But Too "Heavy" For Contention Workload

Before we develop our method, we firstly review the formal definition of the transaction processiong operation. Given a distributed transaction T_i, it runs on m sites $S = \{s_1, s_2, ...s_m\}$. The transaction history is a collection $H = \{h_1, h_2, ..., h_m\}$, in which $h_u(1 \leq u \leq m) = \Pi_u(H)$ is the local history on site s_u. $\Pi_u(H)$ is H's projection on site s_u. For any projected history $h_u(1 \leq u \leq m)$, h_u of transaction T_i includes a collection of operations o_i. o_i can be read write operation, or command operations includes 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, pc_i, pa_i mean transaction T_i commits, aborts, prepares commit or prepare abort respectively. Transaction abort due to many reasons, they can be: 1.User request abort; 2.Violation of serializability; 3.Database node crash for failure. We call

the first two abort reasons *deterministic abort* and the last *non-deterministic abort*.

Transaction T_j has a *commit dependency* on transaciton T_i, written as $T_j \to T_i$, 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 transaction T_i and T_j have direct write-read conflict on record x, in any local history h, T_j read T_i's write record, we call this dependency read-write(wr) dependency and donated by $w_i[x] \to r_j[x]$.

Similarly, if T_i and T_j have direct write-write conflict on record x, T_j overwrite T_i's write record, this is write-write(ww) dependency and written as $w_i[x] \to w_j[x]$.

And if T_i and T_j have direct read-write conflict on record x, T_j write x after T_i reads x, it is a read-write(rw) dependency and recorded as $r_i[x] \to w_j[x]$. A transaction T_j *speculative access* a record x, if there is another transaction T_i, T_j has a commit dependency on T_i and T_j access x before T_i has committed. We write this *danger dependency* as $w_j[x] \to_s r_i[x]$, $w_j[x] \to_s w_i[x]$, $r_j[x] \to_s w_i[x]$. We also write $T_j \to_s T_i$ to indicate transaction T_i has a commit *danger dependency* on T_j.

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 locked 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.

Figure 4 shows 3 transactions work 3 shards, S_1, S_2, S_3. Ttheres are dependencies, $r_1[x] \to w_3[x]$, $w_1[x] \to r_2[y]$, $r_2[y] \to w_3[x]$ and there is $T_1 \to T_2 \to T_3$. There is no circle in this dependency graph and the schedule is serializabile and strict. Figure 5 shows an example of a non-strict but correct schedule. There is 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, $h_1 = \{w_1[x]w_3[x]pc_1c_1pc_3c_3\}$ $h_2 = \{w_1[y]r_2[y]pc_1c_1pc_2c_2\}$, $h_3 = \{r_2[z]w_3[z]pc_2c_2pc_3c_3\}$) is not a strict history, but it's a serializable history. Both of these two schedules are serializable equate with the serial schedule, T_1, T_2 and T_3. Both scheule H_1 and H_2 are correct.

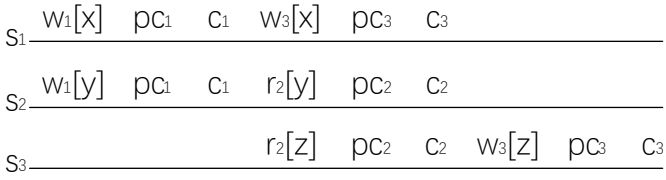


Fig. 4: A strict and serializabile schedule H_1

To get better concurrency, the scheduler can produce schedule like H_2 in Figure 5. Supporse schedule H_2 is created by locking, then transaction T_1 must release its locks or

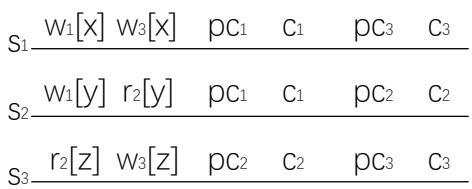


Fig. 5: A non-strict but serializabile schedule H_2

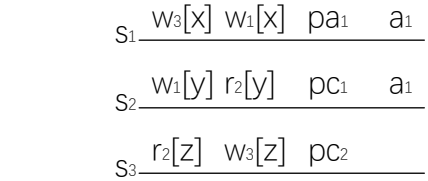


Fig. 6: Schedule H_3, T_1 abort due to non-serializable

make its locks violatable before it knows its commit decision. Then the following write or read cannot wait previous conflict access operations commit. A transaction 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. A distributed transaction need coordinating to decide an agreement on commit. RSM need to reach consensus on every log record. Strictness scheduler on a distributed and replicated database leads to 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 when commit.

A single node transaction can exploit log order to maintain dependency because dependent transactions write their logs orderly [23] [10]. When we extended non-strict locking protocol to a distributed transaction, transaction dependency maintaining is more complex. If transactional scheduler can prevent all aborted transaction, the serializable scheduler is a correct one. But when there is an aborted transaction, it does not. To produce the correct log, the schedule must be both commit serializable and recoverable [31]. Transaction need to maintains commit dependencies to guarantee serializabile and recoverable when exploit non-strictness.

C. Lock Violating Rules and Dependency Tracing

First, lock violation should produce a serializable schedule. Take a schecule H_3 in Figure 6 as an example. There are danger dependencies. $w_3[x] \to_s w_1[x]$, $w_1[y] \to_s r_2[y]$,

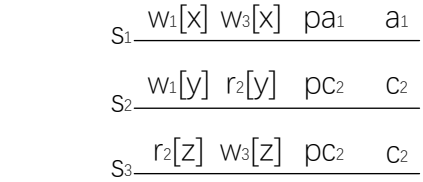


Fig. 7: Schedule H_4, T_2 commit ahead T_1, non-recoverable anomaly

$r_2[z] \rightarrow_s w_3[z]$ in H_3. 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. If schedule H_3 is created by lock violation, not formally, lock violation is just as a transaction releases its lock and another transaction acquire locks, then the locking is not two phases. For a traditional locking scheme, transaction T_2 needs to wait for T_1's release its lock until T_1 commit. Transaction scheduler needs to trace commit dependencies and ensures dependency graph has no circles if a transaction violating locks of another conflict transaciton.

Second, lock violation must be recoverable. In schedule H_4 of Figure 7, 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 also need to maintain dependencies to keep schecule recoverable.

There are serveral questions arised. When should the transaction begin to violate lock? How lock violation could composite with two phase protocol and different transaction models? Violating lock at the first phase of two-phase can shorten more critical path and it isdesigned is superior to violating lock at the second phase. But the database may bears more cascade aborts which are useless work. DLV permits lock violation at two points in the timeline of running transactions. We call these *early-violation* and *late-violation*.

$$H = l_i[x]o_i[x]...vl_j[x]o_j[x]...l_i[y]o_i[y]...ul_i[x]...\ ^{1}\ ,$$

Suppose H is a schedule which is created by lock violation, H is extended by adding lock, unlock and lock violation operations. If there is such $l_i[y]$ operations in schedule H, then transaction T_j is *eraly lock violate* T_i's lock on x; otherwise, this is *late lock violation*. If a transaction use *early lock violation*, it may lead to non-serializable schedule. DLV needs maintains all wr, ww and rw dependencies after violating locks and guarantee the dependency graph of the schedule is acyclic. 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 lock violation case, transactions must tracing dependencies and commit as dependency orders. Composite with 2PC protocol, we have the following rules:

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

By tracing dependencies after violating a lock, DLV schedule achieves both serializabile and recoverable.

IV. DLV IMPLEMENTATION

.

A. In Memory Speculative Versions

Non-strict scheduler needs more complex recovery algorithm to keep correctness. Take a schedule H as an exapmple,

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

If transaction T_1 abort, this cause cascade abort. Traditional database use undo log to process recovery transaction write operations. Implementation undo maybe a little tricky when using exploint non-strict. A wrong recovery expand schedule of T_1 may be like $exp(H)$, in which $w_i^-[x]$ means transaction T_i undo its write on x.

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

Suppose the initial value of x and v of are both 0. The value of x, y after executing every operations in $exp(H)$ is shown in Table I. The final value of y is 1, which should be 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 $exp(H)$

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

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

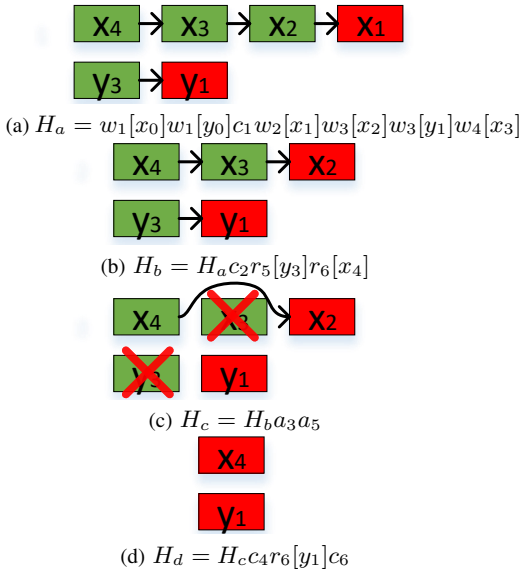
The schdeuler must recovery transaction by the reserve order of write operation. If x and y is on the same database node and transaction is a single node one, to generate this expanding schedule, recovery algorithm can simply execute undo operation follow as log's reserve order, just like traditional Aries algorithm does. However, if x and y are not located at the same node, this undo operation order is hard to accomplish because of partial failure.

In order to avoid this complexity , DLV maintains uncommitted speculative versions in memory and obeys no-steal policy when writting data. No-steal policy need storage cannot write uncommitted data to permernant storages. Since no-steal policy is designed for space efficiency. For most transactions would write a little data excpet the bulk loading ones and modern database runs on a machine with large RAM, using

[1]
$l_i[x]$: T_i lock tuple x
$l_j[y]$: T_j lock tuple y
$vl_j[x]$: T_j violating T_i's lock on x
$ul_i[x]$: transaction T_i release locks on x

no-steal policy 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 efficiency. 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 has no effect on recoverability. *Late-violation*, since it has promised serializability, so it can ignore ww and rw dependencies and only trace wr dependencies for recoverability. Figure 8 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) $H_b = H_a c_2 r_5[y_3] r_6[x_4]$

(c) $H_c = H_b a_3 a_5$

(d) $H_d = H_c c_4 r_6[y_1] c_6$

Fig. 8: 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-violation can is more appropriate then *Late-violation* when there are less cascade abort caused by non-deterministic abort. *Early-violation* also need more dependency tracing costs. Too many cascade abort can lead a lot useless work.

We implementation *Late-Violation* by adding a message round trip to prevent deterministic abort. 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 send a *Ready* message to TM and tells TM its 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 violatable. An interactive-transaction can combine these messages with the last operations and prepare requests in passing, as Figure 3 shows. For a oneshot 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 needed.

DLV records transaction statistic information to decide use *early-violateion* or *late-violation*. We define a distributed transaction working on one RM as partial transaction and a partial transaction enter prepared commit phase but failed to commit finally as partial prepare. DLV calculates partial prepare rate at a time period to decide which violation strategy to choose. We supporse 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 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 following conditions where Θ is is a constant coefficient.

$$N_p/N_t < \Theta * max(RTT_i)$$

DLV would choose *early-violation* if this conditions stisfied. Otherwise it would use *late-violation*.

C. Locking Violation and Maintain Commit Dependency

DLV use *wait-die* protocol to avoid deadlock. At the beginning of a transaction, the transaction current timestamps to generate a transaction id. The conflict transaction operation are queued base their transaction id's order.

DS use register and report [29] to maintain dependency. Every transaction content store an in dependency transaction(in_dn) number record how many transactions is the transaction dependent on. The transaction also keeps 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. This steps are described by line 4 7 in function READ 21. 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 10 of COMMIT 12 function. If a transaction aborts, it may cause a cascade abort. Line 2 of function CASCADE 16 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.

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

```
1: function READ(T, key)
2:     newest_version ← 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 ∉ S.out_set then
5:             S.out_set ← S.out_set ∪ T
6:             T.in_dn ← T.in_dn + 1
7:         end if
8:     end if
9:     if key is write locked by transaction S then
10:        S.wait ← S.wait + 1
11:        wait lock till die
12:        if die then
13:            T.no_da ← 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 ← S.wait + 1
5:         end if
6:         wait lock till die
7:         if die then
8:             T.no_da ← 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
```

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

```
1: function COMMIT(T)
2:     garbage collect old version in
   Tuple(key).version_list
3:     for key ∈ T.write_set ∪ T.read_set do
4:         Unlock(T, key, Read/Write)
5:     end for
6:     for T_out ∈ T.out_set do
7:         T_out.in_dn ← T_out.in_dn − 1      ▷ keep exactly
   once
8:         if then T_out.in_dn = 0
9:             report T_out.in_dn = 0      ▷ stop waiting on
   function PREPARE line 2
10:        end if
11:    end for
12:    response TM message {Commit ACK}
13: end function
1: function ABORT(T)
2:     call CASCADE(T)
3:     for key ∈ T.write_set ∪ T.read_set do
4:         Unlock(T, key, Read/Write)
5:     end for
6:     response TM message {Abort ACK}
7: end function
1: function CASCADE(T)
2:     T.in_dn ← −∞
3:     for key ∈ T.write_set do
4:         if key is ICommit locked by T then
5:             ModifyLock(T, key, IAbort)
6:         end if
7:         for version ∈ Tuple(key).version_list do
8:             if version is created by T then
9:                 remove version from list
10:                break
11:            end if
12:        end for
13:    end for
14:    for T_out ∈ T.out_set do
15:        call CASCADE(T_out)
16:    end for
17: end function
```

V. EXPERIMENTS AND EVALUATIONS

We develop a replicated distributed database demo and evaluate the performance of *DS*. As a comparison with *DS*, 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 16.04. The data is partitioned by 4 shards, each shard has 3 replicas which is replicated across 3 AZs. 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 workload who has 60% NewOrder procedure and 40% Payment procedure.

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.

```
1: function READY(T)
2:     response TM message
       {Ready, wait ← T.wait, non_da ← T.non_da}
3: end function
1: function DECIDE(T, msgs)
2:     if ∀m ∈ msgs, m.non_da is True and
       ∃m ∈ msgs, m.wait > Θ  then
3:         send all RMs message {Speculate}
4:     end if
5: end function
1: function SPECULATE(T)
2:     for key ∈ T.read_set do
3:         Unlock(T, key, Read)
4:     end for
5:     for key ∈ T.write_set do
6:         if key is Write locked by T then
7:             ModifyLock(T, key, ICommit)
8:         end if
9:     end for
10: end function
```

All the transactions are distributed transactions. The data set has 20 warehouse number and shard by warehouse id. Each transaction will retry after 3 seconds if it aborts.

B. Performance Comparation

Figure 9 shows the performance of when adding terminal numbers of each node in the gathered mode (TMs and RMs are located in the same AZ). In Figure 11a, we can see *DS* scales better than the other algorithms. CLV2P has little optimization against S2PL. CLV1P has grater abort rates, as 11b shows, and has bad performance. Figure 11 shows the performance of when adding terminal numbers of each node in scattered mode. (TMs and RMs are located in the different AZs).

C. Parameter Tune

D. Recovery Non-Available Time Comparation

VI. Conclution

We develop DS optimization for locking scheme on cloud distributed database. DS's speculation is only available if it is worthy. According to our evaluation, DS can improve performance of contention workload for shortening critical path. It also avoids unnecessary dependency tracing cost and cascade abort danger when encounter non-contention workload compares previous work.

Acknowledgment

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

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] M. J. Carey and M. Stonebraker, "The performance of concurrency control algorithms for database management systems," in *Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings*, U. Dayal, G. Schlageter, and L. H. Seng, Eds. Morgan Kaufmann, 1984, pp. 107–118. [Online]. Available: http://www.vldb.org/conf/1984/P107.PDF

[3] 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

[4] 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

[5] 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

[6] "Nuodb," https://www.nuodb.com/.

[7] "Cockroachdb," [3] https://www.cockroachlabs.com/.

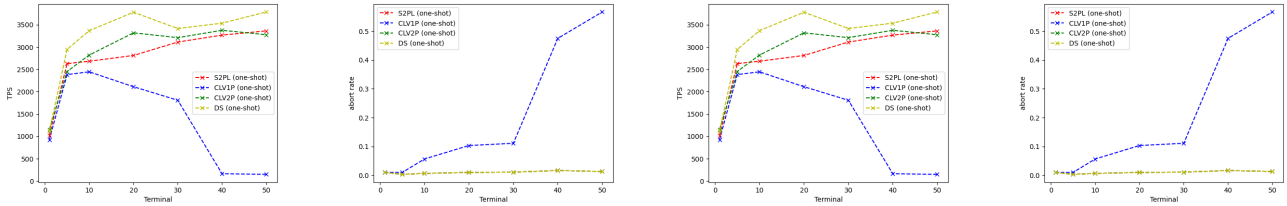
[8] "Tidb," https://pingcap.com/en/.

[9] 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

[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

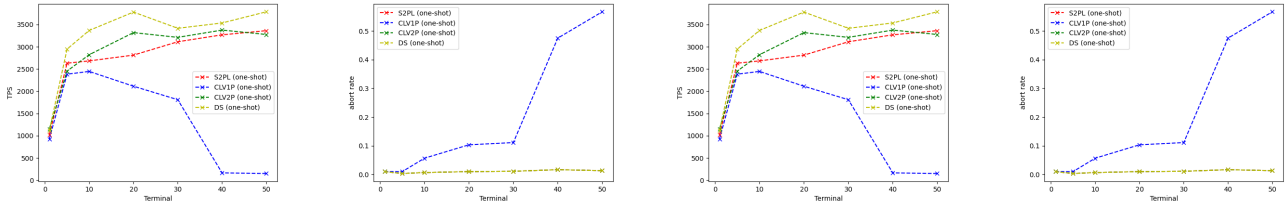
[11] G. Graefe, M. Lillibridge, H. A. Kuno, J. Tucek, and A. C. Veitch, "Controlled lock violation," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 85–96. [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



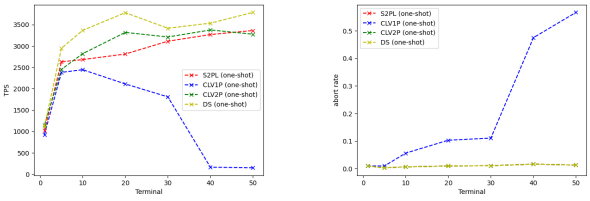
(a) one-shot transaction TPS of different terminal numbers (b) one-shot trnasaction abort rate of different terminal numbers (c) interactive transaction TPS of different terminal numbers (TODO) (d) interactive trnasaction abort rate of different terminal numbers (TODO)

Fig. 9: throughput and abort rate when adding terminal number of each node, gathered mode



(a) one-shot transaction TPS of different terminal numbers (TODO) (b) one-shot trnasaction abort rate of different terminal numbers (TODO) (c) interactive transaction TPS of different terminal numbers (TODO) (d) interactive trnasaction abort rate of different terminal numbers (TODO)

Fig. 10: throughput and abort rate when adding terminal number of each node, scathered mode



(a) transaction TPS of different contention rate (TODO) (b) trnasaction abort rate of different contention rate (TODO)

Fig. 11: throughput and abort rate when adding possible contention

[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] 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

[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_perakelarson_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