A Performance Study of Epoch-based Commit Protocols in Distributed OLTP Databases

Jack Waudby
Newcastle University
j.waudby2@ncl.ac.uk

Paul Ezhilchelvan
Newcastle University
paul.ezhilchelvan@ncl.ac.uk

Isi Mitrani
Newcastle University
isi.mitrani@ncl.ac.uk

Jim Webber
Neo4j
jim.webber@neo4j.com

***Abstract*—Distributed OLTP systems execute the high-overhead, *two-phase commit* (2PC) protocol at the end of every distributed transaction. *Epoch-based commit* proposes that 2PC be executed only once for all transactions processed within a time interval called an *epoch*. Increasing epoch duration allows more transactions to be processed before the common 2PC. It thus reduces 2PC overhead per transaction, increases throughput but also increases average transaction latency. Therefore, required is the ability to choose the right epoch size that offers the desired trade-off between throughput and latency. To this end, we develop two analytical models to estimate throughput and average latency in terms of epoch size taking into account load and failure conditions. Simulations affirm their accuracy and effectiveness. We then present *epoch-based multi-commit* which, unlike epoch-based commit, seeks to avoid all transactions being aborted when failures occur, and also performs identically when failures do not occur. Our performance study identifies workload factors that make it more effective in preventing transaction aborts and concludes that the analytical models can be equally useful in predicting its performance as well.**

***Index Terms*—Distributed Databases, Transactions, Two-Phase Commit, Epochs, Analytical solutions, Simulations, Performance Evaluation, Throughput, Latency**

I. Introduction

When a single-node database reaches its capacity limits, a common option is to partition data across multiple nodes forming a distributed database [1]. When transactions access data within a single node, there is no need for any coordination. However, when workloads contain *distributed* transactions accessing data from multiple nodes, an *atomic commitment protocol*, typically *two-phase commit* (2PC) [2], needs to be executed so that distributed transactions are guaranteed of atomicity and durability when nodes are prone to failures.

Executing 2PC generally extracts a high performance cost [3], [4]. It involves two sequential network round trips, one for each of its phases, and two sequential durable writes. When 2PC is executed at the end of each distributed transaction, it increases the transaction latency by an additional delay of up to $10ms$. In addition, as 2PC execution prolongs the lifetime of a distributed transaction within the database, the potential for data contention among transactions (distributed or otherwise) intensifies which, in turn, leads to further performance degradation. For these reasons, a significant strand of distributed transactions research focuses on minimising the costs inflicted by 2PC executions.

A range of solutions have been proposed, each with their own limitations. Schism [5] attempts to minimise distributed transactions via a workload-driven partitioning scheme. Unfortunately, in real-world scenarios it is often impossible to gain an accurate, prior knowledge on workloads. Another proposal is to eliminate distributed transactions through dynamic data re-partitioning [6], [7], but this incurs non-negligible run-time overheads. Others mandate a declaration of transactions' read and write sets prior to their execution [8], [9], which can be unrealistic in general settings.

A recent practical proposal is *epoch-based commit* [10]. Based on the widely-held notion that node failures are getting less frequent with modern hardware, it views that 2PC need not be executed at the end of every distributed transaction. Instead, 2PC is executed once for all transactions that arrive and get processed within a time interval called an *epoch*. Thus, an epoch is the base unit for doing 2PC and all transactions processed within it either commit or abort through one common 2PC execution. (This idea is a distributed extension of *group-commit* proposed in [11] to reduce disk I/O latency for single-node databases.)

The cost of one 2PC execution is thus amortised over multiple transactions processed within an epoch. Moreover, transactions whose processing is completed can release their locks instead of waiting until they commit at the epoch end; this reduces scope for contention. They can also *asynchronously* log their writes to persistent storage. The effects of all these features are two-fold. On the up side, throughput increases substantially - by four times (4x) as per the experiments conducted in [10]. On the down side, a transaction's results cannot be released until the epoch end when all its writes have become durable. This means that the earlier a transaction arrives during an epoch, the longer it waits before its results can be released; i.e., the average latency of transactions increases.

It is important to note that amortization of 2PC cost over multiple transactions also tends to increase the throughput as epoch size increases, when nodes do not fail. However, if a node fails, then the number of transactions aborted at the end of an epoch (which is when that failure is globally detected) will be large if the epoch was chosen to be long; this wasted work obviously reduces throughput. Thus, the possibility of node failures favours shorter epochs. In fact, there is an optimum epoch length at which throughput is maximised which can be used if increased latency is a minor concern.

(Increased latency is acceptable for many workloads, see [12]–[14].) On the other hand, a user may want a reasonably high throughput as well as an acceptably moderate latency. Such a trade-off requires the means to choose appropriate epoch length. These requirements were left as future work in the original paper [10] and are being fully addressed here.

We derive analytical solutions for estimating throughput and average latency in terms of epoch length and some system and load parameters. Estimating average latency accounts for aborted transactions being completed in subsequent epochs. Our derivations make certain simplifying assumptions, such as the cluster has at most one failed node at any time which holds if failures are independent and less frequent and if a failed node recovers before another node fails.

The protocol of [10] proposes that all transactions executed during an epoch be aborted in case of a node failure. This assumes that each node has directly or indirectly accessed the failed node at sometime during the epoch and therefore all transactions it executed accessed some data now lost due to failure. We examine and find this assumption to be overly pessimistic and develop an *epoch-based multi-commit* version. Each node maintains a list of nodes it interacted with, and uploads the list to 2PC coordinator. The latter then constructs disjoint *commit groups* such that a node within a given commit group woud have interacted, directly or transitively, only with other nodes in that group during the epoch. In case of a node failure, only those nodes in the commit group containing the failed node will abort their transactions and the rest will commit the transactions they processed.

When a workload contains no distributed transaction, each commit group will have just one node and all operative nodes can commit their transactions in the event of a failure. On the other extreme, if the workload has many distributed transactions that cause every node to interact with every other node during an epoch, then there is only one commit group and a failure will cause all operative nodes to abort all their transactions as in the epoch-based commit protocol. Thus, our multi-commit version can automatically take advantage of favourable workload conditions in the event of a failure and avoid excessive aborts, while performing identically as the original version during failure-free epochs.

This paper makes three major contributions. Analytical models for estimating throughput and average latency are developed in Section II for epoch-based commit protocol. They allow an appropriate epoch size to be judiciously chosen for maximum throughput or minimum latency or seeking a trade-off between the two. Secondly, the epoch-based multi-commit protocol is presented in Section III, together with a popular benchmarking case study carried out to support its core design rationale. Finally, a range of simulation experiments involving a cluster of 64 nodes operating for a 100-day period are carried out. They (i) demonstrate the accuracy and efficacy of the models of Section II for choosing the appropriate epoch size, (ii) point to workload features that can aid the multi-commit protocol in minimising aborts when failures occur, and (iii) affirm the effectiveness of the models in selecting the right epoch size also for multi-commit version. Section IV presents the strategies adopted for performance study which is followed by the presentation and discussion of results in Section V. Related works are summarised in Section VI and Section VII concludes the paper.

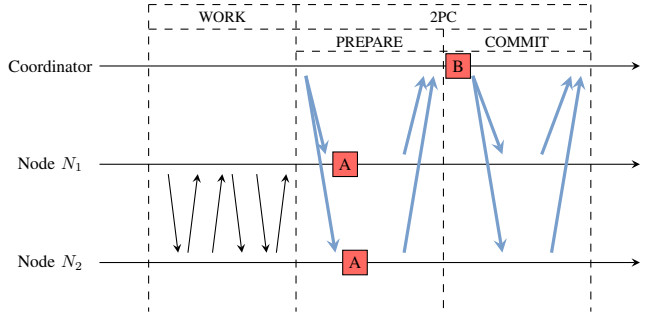


Fig. 1: Work and 2PC intervals of a cycle in the epoch-based commit protocol.

II. Analytical Models for Epoch-Based Commit

A. Protocol Description

To keep the paper self-contained, we briefly describe the protocol before presenting analytical solutions. For greater details on the protocol, readers are referred to [10].

Consider a distributed system consisting of a *coordinator* node and $N, N > 1$ *participant* nodes that are simply called *nodes*. A large OLTP database is partitioned among the latter which execute transactions and 2PC under the control of the coordinator. A (participant) node can fail in a *fail-stop* manner: it functions correctly when operative and fails only by ceasing to be operative. The coordinator, however, is assumed to be built reliably and never fails. More precisely, we assume that the coordinator is internally replicated (primary-backup or state machine replication using atomic broadcast such as [15] or [16]), maintaining a single server abstraction.

The coordinator starts a *cycle* by starting an epoch timer for an interval of a and instructs the nodes to work on transactions. The epoch is referred to as *work* interval in Figure 1 during which transactions are executed but not made durable and their results also not released to end-users. A transaction will release the locks it holds at the end of its execution even though it is not committed. This reduces lock contention but makes the effects of earlier transactions visible to the later ones. This is not a concern as all transactions of a given work interval commit or abort together at the end.

At the end of epoch timer a, the coordinator calls for an execution of 2PC which, as shown in Figure 1, has two phases: *prepare* and *commit*. In prepare phase, coordinator sends a `Prepare` message to each participant node. (Messages exchanged during 2PC are shown by blue arrows in Figure 1.)

When a node receives `Prepare`, it force-logs: (i) ids of all ready-to-commit transactions it executed, and (ii) current epoch number. (These durable writes by participants are indicated in Figure 1 by red squares labelled A.) It then responds with `Prepare-Ack` to the coordinator.

In the commit phase, the coordinator collects responses from participant nodes. If any node has not replied with a `Prepare-Ack` message, *all* nodes are instructed to abort all transactions they executed during the work interval. Else, the coordinator force-logs a *commit* record with the current epoch number (shown as a red square labelled B in Figure 1) and sends `Commit` message to all participant nodes. When a node receives `Commit` message, it commits the transactions it executed and releases the results to clients. It then sends `Commit-Ack` to coordinator and awaits the latter to start the next cycle of work and 2PC intervals.

In summary, multiple transactions are executed during the work interval of each cycle; they are committed (or aborted) in a common 2PC execution. The design rationale behind this approach is that the time taken to execute even a distributed transaction is negligibly small compared to the mean time before failure (MTBF) of nodes. So, the probability of several transactions being executed without encountering any node failure, is fairly high. When the common 2PC execution leads to commit, its overhead per committed transaction becomes very small which, in turn, leads to an increased throughput.

Several studies [17]–[19] analysing node failures in clusters confirm the assumption of MTBF being considerably larger than transaction processing times. For example, Garraghan et. al. [17] analyse Google Cloud Platform traces and find 5,056 out of 12,532 nodes exhibiting 8,954 failure events over a 29 day period. A node's MTBF turns out to be 12.71 hours! Another metric analysed in [17] is also relevant for our analytical models: the mean time to repair (MTTR) is around 6-30 times smaller than MTBF. So, a failed node is most likely to have recovered well before another failure occurs.

B. Modelling Assumptions

We make two assumptions in our analytical modelling and derivation of expressions for estimating maximum throughput and average latency.

Assumption 1: A failed node recovers before another node in the system fails; i.e., between any two consecutive node failures, there is a recovery event of the first failed node.

A common observation in the literature (e.g., [17]–[19]) is that MTBF of nodes is much larger compared to their MTTR: a failed node is almost certain to be repaired before the next failure occurs. This near certainty is here assumed to be a certainty for the number of nodes typically found in a distributed OLTP system. Thus, there is at most one failed node in the system at any time.

Assumption 2: As soon as a node receives `Prepare` message from the coordinator, it *instantly* completes any ongoing transaction execution and enters the 2PC execution.

In reality, some non-zero amount of time will elapse for a node to complete its ongoing execution (if any) and then respond with `Prepare-Ack`. This assumption can lead to an overestimation of throughput.

Our simulations do not have these assumptions and therefore will assess the accuracy of the expressions derived.

C. Maximum Throughput

We derive an analytical expression for estimating maximum attainable throughput by assuming that a node always has a transaction to execute and is never idle.

A participant node fails at exponentially distributed intervals with a low rate, ξ. The repair times are also distributed exponentially with a higher rate $\eta >> \xi$. Recall that nodes go through cycles, each consisting of a *work* interval, U, followed by a random 2PC interval V in which 2PC executed. To start with, we will regard U as random (as opposed to being fixed as a constant a as explained earlier). Denote the probability density functions of U and V by $f_U(x)$ and $f_V(x)$, respectively. Let also $f_W(x)$ be the convolution of $f_U(x)$ and $f_V(x)$, i.e., the p.d.f. of the sum $U + V$.

During a work interval, transactions are served at rate $N\mu$ (transactions per unit time), if all nodes are operative, and at rate $(N-1)\mu$ if one of them has failed. At the end of a cycle, either the commit operation completes successfully and all transactions executed during the work interval U depart, or a node failure has occurred and all transactions executed during U are aborted and remain in the system for re-execution.

The probability, α, that N operative nodes complete one cycle successfully (i.e., with none of them failing), is

$$\alpha = \int_0^\infty f_W(x) e^{-N\xi x} dx = \tilde{w}(N\xi) = \tilde{u}(N\xi)\tilde{v}(N\xi) \ , \quad (1)$$

where $\tilde{u}(s)$, $\tilde{v}(s)$ and $\tilde{w}(s)$ are the Laplace transforms of $f_U(x)$, $f_V(x)$ and $f_W(x)$, respectively.

Consider an interval between two consecutive node failures, to be referred to as the *observation period*. (It is indicated for convenience in Figure 7 in Appendix). The probability that exactly m consecutive work and 2PC cycles are completed successfully during the observation period is

$$p_m = \alpha^m (1 - \alpha) \ . \quad (2)$$

Hence, the average number of successful cycles during that period is

$$\bar{m} = \frac{\alpha}{1 - \alpha} \ . \quad (3)$$

The observation period begins with the repair period of the node that had failed. During that period there are only $N - 1$ operative nodes. By analogy with (1), the probability, β, that they will complete one work and 2PC cycle before the failed node recovers, is

$$\beta = \int_0^\infty f_W(x) e^{-\eta x} dx = \tilde{u}(\eta)\tilde{v}(\eta) \ . \quad (4)$$

Consequently, the average number of consecutive cycles during the repair period (colored red in Figure 7) is

$$\bar{m}_1 = \frac{\beta}{1 - \beta} \ . \quad (5)$$

The total average number of transactions departing during these cycles, J_1, is equal to

$$J_1 = \bar{m}_1 E(U)(N - 1)\mu \ , \quad (6)$$

where $E(U) = -u'(0)$ is the average length of work intervals.

Then we have a cycle that overlaps the repair completion instant (marked in parts with red and green in Figure 7). In addition, if the repair instant falls during the work interval of that cycle, then for the remainder of that interval there is an extra node (i.e., the repaired one) available. The average number of transactions, J_2, departing during this cycle, given that the repair is completed within it, can be expressed as

$$J_2 = E(U)(N-1)\mu + \frac{\mu}{1-\beta}E(U-R)\ , \qquad (7)$$

where $E(U-R)$ is the expected remaining work interval, given that the repair completes within the cycle. Averaging over the distribution of U, we can write

$$E(U-R) = \int_0^\infty f_U(x)\int_0^x (x-y)\eta e^{-\eta y}dydx\ . \qquad (8)$$

After carrying out the integration and substituting the result into (7), the latter becomes

$$J_2 = E(U)(N-1)\mu + \frac{\mu}{1-\beta}\left[E(U) - \frac{1-\tilde{u}(\eta)}{\eta}\right]\ , \qquad (9)$$

where $\tilde{u}(s)$ is the Laplace transform of $f_U(x)$.

Finally, the total average number of departures during fail-free cycles with N operative nodes (marked green in Figure 7), with their average number being $\bar{m} - \bar{m}_1 - 1$, is given by

$$J_3 = (\bar{m} - \bar{m}_1 - 1)E(U)N\mu\ . \qquad (10)$$

The system throughput, T, defined as the average number of departures per unit time, is obtained by dividing the total average number of departures during the observation period by the average length of the observation period :

$$T = (J_1 + J_2 + J_3)N\xi\ . \qquad (11)$$

Now consider the total average number of transactions, D, that are lost during the observation period. Losses occur either during the initial repair period, when transactions attempt to access the failed node, or when the last cycle in the observation period is interrupted by the next node failure.

As soon as a transaction is admitted into a node, it produces a list of a random number, k, of other nodes that it would need to access. If the failed node is on that list, the transaction is instantaneously dismissed and is lost. That procedure is repeated with the transaction that follows, so following a service completion there may be a series of transactions lost instantaneously.

The probability, γ, that a transaction admitted into an operative node has the failed node on its list is

$$\gamma = \frac{E(k)}{N-1}\ , \qquad (12)$$

where $E(k)$ is the average size of the list. This is a given parameter. The average number of transactions lost instantaneously following a service completion during the repair period is therefore equal to $\gamma/(1-\gamma)$.

We conclude that the average number of transactions lost during the $\bar{m}_1$ successful work and commit cycles within the repair period is

$$D_1 = J_1\frac{\gamma}{1-\gamma}\ , \qquad (13)$$

where J_1 is given by (6).

To find the average number, D_2, of transactions lost during the work and 2PC cycle overlapping the repair instant, we proceed as in the derivation of (7), but count only the transactions completed before the repair instant. This leads to the following expression:

$$D_2 = \frac{(N-1)\mu}{1-\beta}\frac{\gamma}{1-\gamma}\left[\frac{1}{\eta}[1-\tilde{u}(\eta)] - \tilde{u}'(\eta)\right]\ . \qquad (14)$$

Finally, we need the average number, D_3, of transactions that are lost from the last work and commit cycle in the observation period, the one that is interrupted by the next failure instant. Since all transactions executed during that cycle are lost, we can write

$$D_3 = E(U)N\mu\ . \qquad (15)$$

The overall rate of transaction losses, D, is given by the total average number of losses during the observation period, divided by the average length of the observation period:

$$D = (D_1 + D_2 + D_3)N\xi\ . \qquad (16)$$

The work interval U is set by the control policy as a constant, a. On the other hand, the commit operation is affected by communication delays, so it is more natural to assume that V is random, possibly distributed exponentially with mean b. In that case, we would have

$$\tilde{u}(s) = e^{-as}\ \ ;\ \ \tilde{v}(s) = \frac{1}{1+bs}\ . \qquad (17)$$

D. Average response time

We will no longer regard that nodes always busy but assume that transactions arrive in a Poisson stream with rate λ and, if there are no available nodes, wait in an external FIFO queue. After being processed, a transaction does not depart immediately but is held in the queue until the end of the current cycle. If commit is the 2PC outcome, all transactions that were executed during the work interval depart together. Otherwise, they are aborted and continue to remain in the queue. The performance measure of interest here is the steady-state average response time, W, defined as the interval between the arrival of a transaction into the system and its departure.

This type of system has been referred to in the literature as a *queue with bulk services*. At certain *service instants*, batches of transactions are removed from the queue. More precisely, if the size of the current batch is m and the number of transactions present just before the service instant was n, then just after the service instant there are $n - \min(n, m)$ transactions present. A model where the intervals between service instants have a general distribution and all batches have the same fixed size was analyzed by Bailey [20] more than half a century ago.

Bailey's result cannot be used in our case because the number of transactions departing at a service instant, i.e. when 2PC execution completes, depends on whether a breakdown occurred during the cycle or not, and also on whether there were N or $N-1$ operative servers. We propose two estimates for W: the first is pessimistic and can be treated as an upper bound on the response time; the second is clearly optimistic and will provide a lower bound.

E. Upper bound, W_u

The first estimate is obtained by assuming that the consecutive intervals between service instants are i.i.d. random variables distributed exponentially with mean $a+b$, where a is the average work interval and b is the average 2PC interval. The parameter of that distribution will be denoted by $\nu = 1/(a+b)$. The reason why this is a pessimistic assumption is that the coefficient of variation of the exponential distribution is 1, while in practice the work interval is likely to be constant, or nearly constant. The 2PC interval tends to be much smaller than the work interval, so even if it is random, the coefficient of variation of a full cycle (comprising both work and 2PC intervals) would tend to be closer to 0 than to 1.

Under the exponential assumption, the probability that a full cycle is not interrupted by a node failure is now approximated by

$$\alpha = \frac{\nu}{N\xi + \nu} \ . \tag{18}$$

When $N\xi$ is small, this value is very close to the one produced by (1).

Hence, a service batch is of size 0 with probability $q_0 = 1 - \alpha$.

Since the average period during which there are $N-1$ operative servers is $1/\eta$ and the average period during which there are N operative servers is $1/N\xi$, we can say that a 2PC interval has $N-1$ operative servers with probability $q_1 = \alpha N\xi/(N\xi+\eta)$, and has N servers with probability $q_2 = \alpha\eta/(N\xi+\eta)$. In the former case, an average of $m_1 = a(N-1)\mu$ transactions are served during the cycle, and in the latter case $m_2 = aN\mu$ transactions are served.

The above arguments support an assumption that the service batch size, m, is equal to

$$m = \begin{cases} 0 & \text{with probability} \quad q_0 \\ m_1 & \text{with probability} \quad q_1 \\ m_2 & \text{with probability} \quad q_2 \end{cases} \ . \tag{19}$$

If m_1 and m_2 are not integers, their integer parts are taken.

The average batch size, B, is

$$B = m_1 q_1 + m_2 q_2 = aN\mu\alpha \frac{(N-1)\xi + \eta}{N\xi + \eta} \ . \tag{20}$$

The necessary and sufficient condition for the stability of the bulk service queue is that the transaction arrival rate should be strictly less than the average service capacity:

$$\lambda < \nu B \ . \tag{21}$$

When the failure rate is small and the repair rate is significantly higher, the right-hand side of this inequality is very close to the maximum throughput, T, obtained in the previous section. Thus, requirement (21) is almost identical to the more accurate stability condition $\lambda < T$.

Let π_n be the steady-state probability that there are n transactions present in the queue. Because of the bulk service assumption, any transactions that are in fact being served, are considered to be in the queue until the next service instant. The number n increases by 1 at arrival instants, and it decreases by 0, m_1 or m_2 at service instants. Equating the up and down transition rates across the boundary between states n and $n+1$, we obtain the following set of balance equations.

$$\lambda\pi_n = \nu \left[\sum_{j=1}^{m_1} (q_1+q_2)\pi_{n+j} + \sum_{j=m_1+1}^{m_2} q_2\pi_{n+j} \right] ; n = 0, 1, \ldots \tag{22}$$

We shall obtain the general solution to this set of equations in geometric form:

$$\pi_n = C z_1^n \ , \tag{23}$$

where C and z_1 are some positive constants. Substituting (23) into (22), we find that the equations are satisfied as long as z is a zero of the following polynomial of degree m_2.

$$P(z) = \lambda - \nu \left[\sum_{j=1}^{m_1} (q_1+q_2) z^j + \sum_{j=m_1+1}^{m_2} q_2 z^j \right] \ . \tag{24}$$

In addition, in order that we may obtain a probability distribution, z_1 must be a positive real number in the interval $0 < z_1 < 1$.

We have $P(0) = \lambda > 0$ and $P(1) = \lambda - \nu(m_1q_1 + m_2q_2) < 0$, according to (21). Therefore, $P(z)$ has a real zero, z_1, in the interval $(0,1)$. This provides a normalizable solution to the set of balance equations and allows us to write

$$\pi_n = (1-z_1) z_1^n \ ; \ \ n = 0, 1, \ldots \ . \tag{25}$$

It is possible to prove formally that $P(z)$ has no other zeros in the interior of the unit disk, but this also follows from the fact that an ergodic Markov process cannot have more than one normalizable distribution.

The steady-state average number of transactions in the system, L, is obtained from (25):

$$L = \sum_{n=1}^{\infty} n\pi_n = \frac{z_1}{1-z_1} \ . \tag{26}$$

The upper bound of the average response time, W_u, is then provided by Little's theorem:

$$W_u = \frac{L}{\lambda} \ . \tag{27}$$

F. Lower bound, W_d

A very simple lower bound is derived by making two optimistic assumptions. The first is that the work interval and 2PC interval are constant, of lengths a and b, respectively. The second is that the transactions arriving during the work interval are cleared at the end of that cycle, while those arriving during the commit operation are cleared during the next cycle,

provided that no breakdown occurs in the meantime. That would be a reasonable assumption if the total average number of arrivals during work and 2PC intervals of a cycle is smaller than the average number that can be served by $N-1$ servers during a work interval: $\lambda(a+b) < a(N-1)\mu$.

When the cycle duration is constant at $(a+b)$, the probability that a cycle does not involve a node failure is

$$\alpha = e^{-N\xi(a+b)} \ . \tag{28}$$

In that case a transaction arriving during a work interval remains in the system for an average of half a work interval plus 2PC interval, while an arrival during a 2PC interval remains in the system for an average of half 2PC interval plus a full cycle. The probabilities that an incoming transaction arrives during a work interval or a 2PC interval are $a/(a+b)$ and $b/(a+b)$, respectively. Hence, the average sojourn time given that the cycle is failure-free can be written as

$$\left[(\frac{a}{2}+b)\frac{a}{a+b} + (\frac{b}{2}+a+b)\frac{b}{a+b}\right] = \frac{a+3b}{2} \ .$$

In the event of a node failure, the average sojourn time is half a cycle plus a full cycle. Thus, the lower bound on the response time becomes

$$W_d = \alpha\frac{a+3b}{2} + (1-\alpha)\frac{3(a+b)}{2} \ . \tag{29}$$

Remember that the average work interval, a, is not a system characteristic but is set by the operating policy. It is natural to ask therefore, how should that interval be chosen in order to minimize W? On the one hand, increasing a may improve the throughput (although that effect is mitigated by an increase in the probability of a node failure during a full cycle). On the other hand, transactions are kept in the system for longer. Intuitively, there should be an optimal value for a.

Note that the lower bound (29) tends to be an increasing function of a. Therefore, if W_d is taken as a criterion, the optimal a is the smallest value that justifies the assumption made above, i.e. that the average number of transactions arriving during a cycle should be significantly smaller than the number that $N-1$ servers can serve during a work interval. We suggest as an empirical rule-of-thumb that one should chose the smallest a that satisfies the inequality $\lambda(a+b) < 0.8a(N-1)\mu$. This yields the value, a^*, that minimizes W_d as

$$a^* = \frac{\lambda b}{0.8(N-1)\mu - \lambda} \ . \tag{30}$$

If the upper bound is minimized, the optimal work interval will, in general, be different. These differences will be investigated experimentally.

III. Epoch-based Multi-commit

A. Rationale and Approach

Epoch-based commit assumes that uncommitted data within any node is used, directly or indirectly, by every other node during a work interval. Hence, all transactions executed are aborted in case of a node failure. In reality, this assumption is pessimistic and does not always hold. For example, if a node is to fail shortly after it starts its work interval; it would be very unlikely that each operative node executes a distributed transaction in that short duration *and* uses uncommitted data held by the node that goes on to fail later.

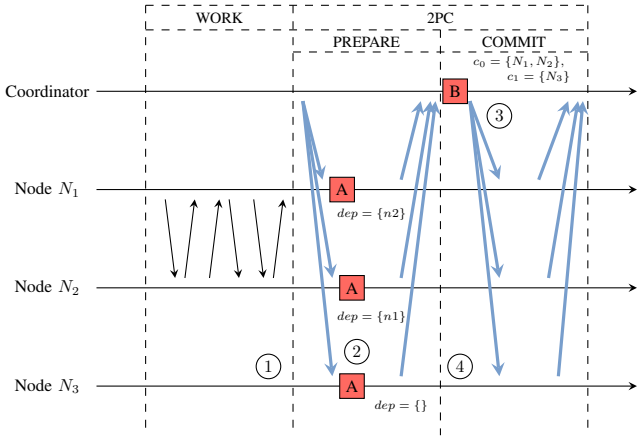


Fig. 2: A cycle in epoch-based multi-commit. A node's epoch-dependency list is indicated as *dep*.

Epoch-based multi-commit avoids, where possible, aborting all transactions and thereby seeks to improve throughput and reduce average latency. Interactions between nodes are monitored in a lightweight, low-overhead manner to determine whether a node needs to abort its transactions, when another node is found to have failed. Interactions that call for aborting of transactions, need not just be directly with the failed node but can also be transitive in nature, as explained below.

Suppose that node N_k updates an object; these updates do not become durable until N_k commits and will be lost if N_k fails before that. Let N_j process a (distributed) transaction by reading an uncommitted update at N_K and updating some local objects which are in turn read by a third N_i while processing another transaction. If N_k crashes, N_j must abort its transactions as some of them involved reading *dirty* data from N_k; this in turn leads to *cascading aborts* at N_i even though there is no direct interaction between N_i and N_k.

We will express the pattern of node interactions during each work interval as a symmetric and transitive binary relation between nodes. This relation is called *epoch dependency* and is defined as follows: Node N_i has epoch dependency with N_j during a given work interval if and only if: (i) N_i accesses data from N_j during that work interval or vice versa, or (ii) there is another N_k for that work interval such that N_i has epoch dependency with N_k and N_k with N_j.

Two remarks are in order. First, the above definition does not distinguish whether a node interaction involves accessing uncommitted or committed data, even though the latter does not call for cascading aborts. Despite some advantages in making this distinction, we avoid it in order to keep the overhead of monitoring node interactions as small as possible.

Secondly, epoch dependency is also reflexive by definition, as each node accesses data from itself. So, it is an *equivalence*

relation defined on participant nodes. It therefore partitions the nodes into disjoint subsets which we call *commit groups*: each node is in exactly one group, any two nodes of a group are related by epoch dependency, and no node has epoch dependency with any node not in its own group. Therefore, if a commit group has no failed node, then its member nodes can commit during 2PC; otherwise, they must abort.

Referring to Figure 2, we see nodes N_1 and N_2 interacting with each other during the work interval and N_3 executing no distributed transactions. So, N_1 and N_2 have epoch dependency with each other and N_3 only with itself; thus, two commit groups, $c_0 = \{N_1, N_2\}$ and $c_1 = \{N_3\}$, emerge. If N_3 fails, N_1 and N_2 can commit their transactions because they did not access any data from N_3. If N_1 fails, $N_2 \in c_0$ must abort its transactions, while N_3 is unaffected.

A failed node can thus prevent nodes of only one group from committing. That is, if node interactions during a work interval lead to multiple commit groups emerging at the end, then all nodes do not have to abort their transactions in the event of a failure. On the other hand, if only one commit group exists at the end of a work interval, then a node failure will make all operative nodes to abort, i.e., the epoch-based multi-commit defaults to the original, epoch-based commit. For example, had N_1 and/or N_2 in Figure 2 interacted with N_3 during the work interval, then $c_0 = \{N_1, N_2, N_3\}$ would be the only commit group and any node failure would mean that all other nodes aborting their transactions.

B. Motivation: TPC-C Case Study

For multi-commit protocol to minimise aborts, multiple commit groups should emerge at the end of a work interval. Such an outcome depends primarily on three workload characteristics: (i) proportion of distributed transactions, (ii) average number of remote nodes accessed by distributed transactions, and (iii) *node affinity* or the likelihood of a transaction in a given node accessing data in another given node due to correlations between data partitions hosted by the two nodes.

The smaller the proportion in (i), the larger is the likelihood of multiple groups. In the limit, if there are no distributed transactions at all, then each commit group is a singleton with a distinct node - as c_1 in Figure 2. The smaller the average in (ii) and the stronger the affinity, the more likely is that multiple groups emerge even if the workload has a higher proportion of distributed transactions. In what follows, we consider a canonical benchmark system to motivate that multi-commit scheme can be very useful in practical settings and its performance is worthy of a detailed evaluation.

TPC-C is the canonical benchmark for evaluating performance of OLTP databases [21]. It models a warehouse order-processing application and consists of five transaction types. Only two types, `Payment` and `NewOrder`, involve accessing remote nodes. A `Payment` transaction involves updating the payment amounts for a given warehouse and then updating customer information. The customer belongs to remote warehouse on another server with a 15% probability. In short, the `Payment` transaction accesses at most two partitions. A `NewOrder` transaction updates 5-15 items in the stock table. Of these items, 99% are local to its home partition, while 1% are at a remote partition.

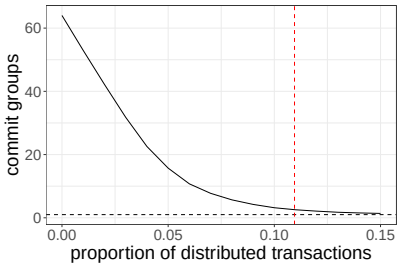


Fig. 3: Number of commit groups *vs* proportion of distributed transactions. The red line indicates the threshold after which single commit group is the only outcome.

In our experiment, the epoch size was fixed to $10ms$ and a throughput of 300K transactions per second in a cluster with 64 servers was assumed. Figure 3 depicts the number of commit groups formed when the the proportion of distributed transactions is varied from 1% to 15%. Commit groups are numerous when the proportion does not exceed 8% which is the typical proportion of distributed transactions encountered in practical settings. Thus, multi-commit is indeed a practical alternative to the original scheme.

C. Multi-Commit Protocol Description

This protocol is identical to epoch-based commit described in Section II, except for the following two additions.

Monitoring node interactions. When node N_i executes a distributed transaction and sends a `Remote-Op` message to another node N_j, it enters the remote N_j in its local, *epoch-dependency* list. Similarly, when N_j receives `Remote-Op` message from N_i, it enters the latter in its list. Thus, interacting nodes have each other in their *epoch-dependency* lists.

Computing commit groups. This is done by the coordinator node if some participant node has not responded to it with `Prepare-Ack` in the prepare phase of 2PC execution. Recall that the coordinator executes 2PC at the end of work interval by sending `Prepare` message to all participant nodes. (2PC messages are shown in blue in Figure 2.) Each operative node will, in turn, respond to the coordinator by sending a `Prepare-Ack` message which will now include its epoch-dependency list as indicated in Figure 2.

If the coordinator receives `Prepare-Ack` from all nodes, then a `Commit` message is sent to all nodes. Otherwise, it constructs a graph where a vertex represents a participant node and an edge an epoch-dependency as reported within the epoch-dependency lists it received. Commit groups are then found by using Tarjan's algorithm to identify strongly connected components [22]. Participant nodes of commit groups that contain no failed node are sent a `Commit` message and the rest an `Abort` message. Based on the descriptions in section III-A, it is easy to see that a node is sent `Commit`, if and only if it has not interacted with a failed node directly or

transitively, during the work interval. (A proof by contradiction is possible and not done here.)

IV. Performance Evaluation Strategies

The principal aim of simulations is two-fold: to assess the accuracy of the analytical models and explore circumstances in which the epoch-based multi-commit can perform better than the original epoch-based commit. The former will assist database administrators in choosing a suitable work interval, a, to accomplish their performance targets and the latter will help to demonstrate that the multi-commit protocol is a practical alternative to the original. Recall that multi-commit cannot perform worse than the original in any circumstance; this is because the former differs from the latter only when a node fails at which time the coordinator expends a small computational cost on trying to minimise aborts.

In assessing the efficacy of the expression for maximum throughput, our discrete event-based simulations [23] will follow the experimental setup in [10]: nodes will spawn a new transaction as soon as they finish executing the current one. Thus, the nodes are never idle and the resulting throughput will be the maximum attainable. In measuring the average latency, simulations will consider such values for transaction arrival rates that the system is kept in a steady state; i.e., the number of transactions waiting to be processed will not grow monotonically with time. Under this steady state condition, throughput is same as the arrival rate and hence not measured.

In our simulations, incoming transactions are distributed ones with 10% probability which is larger than the threshold 8% observed in Figure 3 for having multiple commit groups at the end of a work interval. Thus, we seek to explore the effects of node-affinity when the proportion of distributed transactions does not favour the emergence of multiple groups.

A distributed transaction interacts with one remote node and we consider two policies for choosing that remote node: *random* and *paired affinity*. In the former, the remote node is randomly chosen; in the latter, nodes are paired and a distributed transaction originating in a given node accesses the paired node with 90% probability and a randomly chosen one with 10% probability. Node-pairing captures data correlations between the partitions hosted by the paired nodes.

Note that if a remote node chosen for data access is crashed, then processing of that transaction ceases and all effects of having processed it are undone. Such a transaction is called *'dropped'* in Section II-C and not counted in throughput. Also, 10% of transactions accessing one remote node sets $E(\kappa) = 0.1$ in Equation (12).

The parameters of the analytical models and their values used in simulations are summarized in Table I. A cluster size of 64 nodes and one coordinator is similar to that used in the experimental analysis of concurrency control protocols in [4]. The choice of $\mu = 1$ is guided by the fact that OLTP transactions' useful work consumes about one millisecond and they seldom have user stalls, rather they are executed as stored procedures [1]. The mean time to execute 2PC is represented by b. To estimate b, it was assumed a disk flush takes $10\mu s$ and database nodes are co-located within the same datacenter with a round trip time of $1ms$. Thus, as 2PC operations involve 2 sequential disk writes and 1.5 network calls before results are released back to clients, b is set to $1.7ms$. Following [17], the mean time between failure $1/\xi$ and the mean time to repair $1/\eta$ are taken to be 12 hours and 30 minutes respectively.

TABLE I: Parameters of the analytical models and simulation.

Symbol	Meaning	Values
N	Number of participant nodes	64
a	Work interval (ms)	4-1800
b	Mean time to commit (ms)	1.7
μ	Node's transaction service rate (txn/ms)	1
$1/\xi$	Mean time between failure (hr)	12
$1/\eta$	Mean time to repair (min)	30
$E(\kappa)$	Remote servers accessed by transactions	0.1
$\lambda^{\dagger}$	Transaction arrival rate (txn/s)	30000,40000

† Average response time model only.

Given that $N = 64$, it is possible to encounter more than one failed node in simulations even though $1/\xi \gg 1/\eta$. (Note: the larger the value of N, the less likely it is for Assumption 1 to hold.) Thus, any loss of accuracy due to Assumption 1 in Section II-B is assessed. Simulations measure the following metrics:

System throughput (T): Number of transactions committed per second.

Lost transaction rate (D): Rate at which transactions are being aborted or dropped due to failures.

Committed transactions during failures (CT_f): Average number of transactions committed in cycles with failures.

Average Response time (W): The response time is measured from the point when a transaction enters the system, to the point when it departs, after potentially several retries. Since transaction are processed in batches and some may not be committed in their first execution, the average value (in ms) is computed over the simulation period.

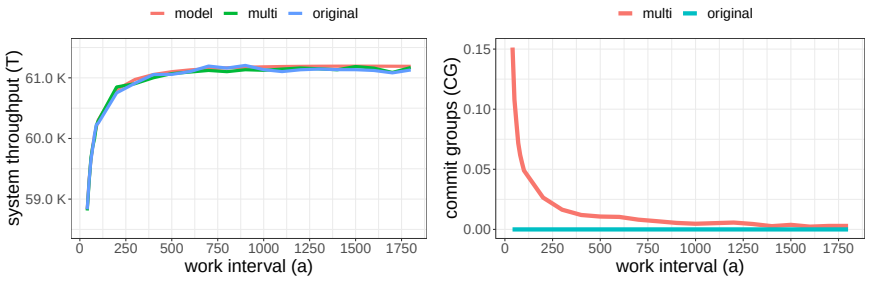
Operational commit groups (CG): the number of commit groups that do not contain a failed node, given that node failure has occurred in a given cycle. It is zero for the epoch-based commit where nodes always form one single commit group.

V. Evaluation

Each simulation run took approximately 12 hours to complete and simulated a cluster operational period of 100 days in order to have thousands of cycles with node failures. (Note: Experiments in [10] did not study the impact of node failures.) For example, when $a = 40\ ms$, 10,972 cycles had failures out of a total of 207 million cycles simulated. So, the number of operational commit groups reported here would be an average of at least 10,000 values obtained. We observed up to 10 cycles having multiple node failures when $a = 1800\ ms$.

A. Maximum Throughput

Figure 4a plots the maximum attainable throughput values against the work interval a which is varied from 40 to 1800 ms. Throughput estimated using the expression in Section II-C is referred to as *'model'* and those measured in simulations for



(a) System throughput $vs.$ a in ms. (b) Operational commit groups $vs.$ a in ms.

Fig. 4: Maximum throughput as work interval a varied from 40 to 1800 ms.

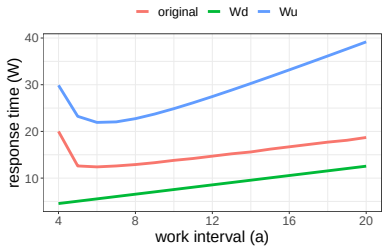


Fig. 5: Average response time (ms) in epoch-based commit $vs.$ a in ms.

epoch-based commit and epoch-based multi-commit are labelled as *'original'* and *'multi'* respectively. Simulations used the random assignment policy when a distributed transaction sought to interact with a remote node.

We can make three observations from Figure 4a. First, the estimated throughput very closely tracks the simulated values at all a. In fact, the maximum difference ever observed was around 3.8%. This suggests that Assumptions 1 and 2 of Section II-B have a negligible impact on the accuracy and the analytical model is nearly exact.

Secondly, throughput values of both protocols are nearly identical. This is explained by Figure 4b that presents the number of operative commit groups (CG) formed in cycles with node failures. CG takes the maximum value of just 0.15 and rapidly falls as a increases. Such insignificantly small values of CG in multi-commit are due to the proportion of distributed transactions (10%) in the workload and the random policy employed for choosing the node to interact with. These two factors lead to almost all operative nodes interacting, directly or indirectly, with the failed node by the time the work interval a completes. This effect is more pronounced for larger a values. Consequently, multi-commit cannot perform significantly better than the original when nodes fail. Moreover, even this minute performance advantage of multi-commit during cycles with failures nearly vanishes when average throughput is taken over all cycles, because failure-free cycles far outnumber those with failures. (Recall, when there are no failures, multi-commit performance is identical to the original.)

Finally, the analytical expression of Section II-C can be reliably used in choosing appropriate a_T when maximum throughput is the primary concern. For convenience, Figure 4a is reproduced in Appendix as Figure 8 without simulated throughput values so that throughput estimates for various a are clear. Referring to Figure 8, we observe that throughput does not decrease until $a = 1500\ ms$; thus, optimal a_T^* is 1500 ms. For some workloads, a work interval around 1500 ms will offer unsatisfactory latency and be unacceptable. Thus, finding a smaller a_T that still offers an acceptable maximum throughput may be desirable and can be guided by the observation that increasing a need not fetch a proportional increase in throughput. Figure 8 shows four distinct regions where the rate of throughput increase is markedly different: the gradient is very large, fairly large, small and very small when $a \in [40, 100)$, $a \in [100, 300)$, $a \in [300, 500)$ and $a \in [500, 1500]$ ms respectively.

B. Average Response Time

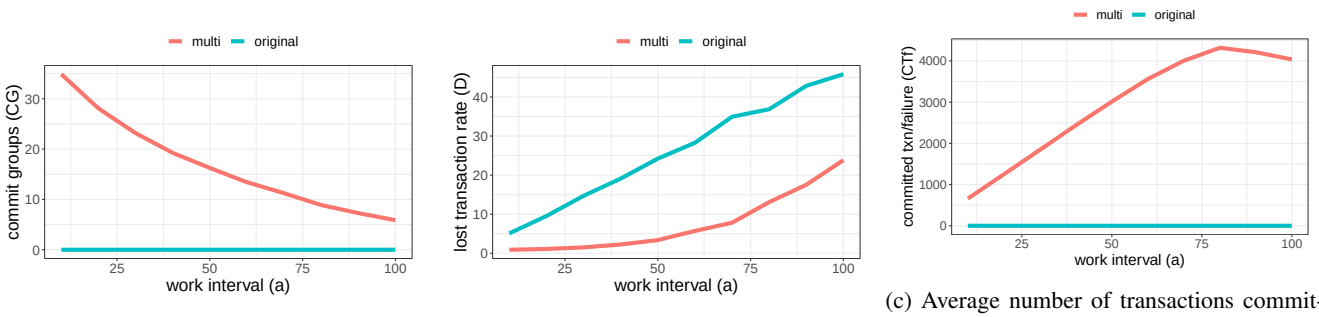
Our second set of experiments focus on assessing the effectiveness of the average response time analytical models in Sections II-E and II-F. Simulations retain the random assignment policy for distributed transactions; transaction arrival rate per second is taken to be $\lambda = 30,000$ which is approximately 90% of the maximum throughput when $a = 5ms$ to ensure the system is in a steady state. Figure 5 plots the model estimates of the lower (W_d) and upper bounds (W_u) and the simulation values measured for epoch-based commit protocol as a is varied from 4 to 20 ms. (The range choice for a is guided by [10] where experiments used $a = 10$ ms.)

The response times measured in simulations are well within the upper and lower bound estimates. The latter increase linearly with a as expected. The W_u plot predicts the optimum a for minimising the average response time as: $a^* = 6$ ms which is consistent with simulations. Though the analytical expression for W_u (Section II-E, Equation (27)) identifies a^* reasonably accurately, the actual response times are much closer to W_d as a increases and the maximum difference we observed was 6 ms. So, to summarise, analytical expressions for W_u and W_d are reasonably accurate in predicting a^* and the actual response times for $a \geq a^*$ respectively. The simulation response times obtained for multi-commit were very close to those presented for the original for reasons explained in Section V-A and hence they are not shown.

C. Paired Affinity

We ran the maximum throughput simulation experiment using the paired affinity selection policy for distributed transactions. Figure 6a displays the number of operational commit groups (CG) formed whenever failures occurred in a cycle. In sharp contrast to Figure 4b, CG for multi-commit starts at a much larger value of 35 when $a = 10$ ms and falls steadily to 5 as a increases to 100 ms. This suggests a strong potential for reducing the number of aborts when failures occur.

Figure 6b plots the lost transaction rates (D) for both protocols and shows that D for multi-commit is consistently smaller



(a) Commit groups $vs.$ a.

(b) Lost transaction rate $vs.$ a in ms.

(c) Average number of transactions committed in cycles with node failure CT_f $vs.$ a.

Fig. 6: Simulations under paired-affinity as work interval a varied from 10 to 100 ms.

than the original. For small a, the difference is small, because the number of transactions lost due to failures is small for both protocols. But as a increases, D for the original increases almost linearly while that for multi-commit does not start increasing significantly until $a = 50$ ms; at that point, multi-commit shows 83% reduction in D with the corresponding CG being around 17 in Figure 6a. As a increases further, CG for multi-commit starts dropping significantly in Figure 6a and consequently D for multi-commit starts increasing rapidly.

Finally, Figure 6c shows the average number of transactions committed by the protocols in those cycles where node failures occur (CT_f). From Figure 6c it is clear that multi-commit avoids a significant number of aborts; note that $CT_f = 0$ in the original. However, these differences in CT_f make insignificant difference when throughput and latency are averaged over the simulation period, because cycles without failures far outnumber (by four orders of magnitude) those with failures and both protocols perform identically in fail-free cycles. Thus, the average maximum throughput and average latency for both protocols, plotted in Figures 9a and 10a (in Appendix), are almost identical. This implies that the analytical expressions obtained for epoch-based commit under the random policy have a wide applicability: comparing Figures 9a and 10b with Figures 9b and 5 respectively suggests that those expressions are equally applicable for (i) obtaining appropriate a for epoch-based multi-commit, and (ii) epoch-based protocols under paired affinity policy.

VI. Related Work

Numerous approaches to improve distributed transaction processing performance have been proposed. Workload-driven partitioning can *minimize* distributed transactions [5] and dynamic data repartitioning can *eliminate* them completely [6], [7]. *Mandating determinism* also avoids 2PC [8], [9], [24]. A caveat with such a mandate is that transactions' read and write sets be known prior to execution [25], else, a reconnaissance phase is executed to discover these sets which amounts to running a transaction twice. Aria [26] avoids this caveat by using a deterministic reordering mechanism, but its performance suffers under high contention workloads [10]. Prognosticator [27] circumvents this limitation by using symbolic execution to build key-level transaction profiles.

Some systems *relax consistency guarantees* to achieve better performance, offering snapshot isolation [28]–[30] or highly available transactions [31]. Others combine concurrency control, replication, and commitment into a unified protocol to reduce WAN round trip instances and thus decrease latency [32]–[37]. Similarly, *Parallel Commits* [38] halves latency by concurrently performing consensus round trips.

Several OLTP databases utilize epochs to exchange improved throughput for higher latency. Silo uses epochs to reduce shared memory writes [13]. Obladi combines them with Oblivious RAM to hide access patterns [12]. STAR [39] runs distributed and single-node transactions in different epochs. COCO [10] leverages epochs to mitigate against the costs of 2PC and synchronous replication. Our epoch-based multi-commit is the first to combine epochs and data access patterns to minimise aborts in the presence of node failures.

Constructing analytical models of data management systems has a rich history. Objectives vary: performance prediction in messaging systems [40], studying the behavior of concurrency control protocols [42]–[44] , and atomic commitment protocols in distributed databases [45], [46]. Our work is the first to present analytical models for epoch-based commit.

VII. Conclusion

We have developed two analytical models for the epoch-based commit protocol which allow database operators to maximize throughput, minimize average response time, or seek a trade-off between them. The accuracy of these models has been validated through a simulation study that considered a cluster of 64 nodes operating for 100 days. We also developed epoch-based multi-commit, which aims to minimise transaction aborts in the event of node failures, but performs identically to the original version under other circumstances. Our simulation study affirms that multi-commit performs better when distributed transactions originating at a given node tend to access specific other nodes in their remote interactions. When failures are rare, the analytical expressions derived for the original protocol can also be used in determining the right epoch intervals for the multi-commit version as well. Thus, we offer a practical alternative to epoch-based commit and analytical solutions to efficiently tune the parameter of epoch-based commit protocols in practical settings.

REFERENCES

[1] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, & P. Helland, "The end of an architectural era: It's time for a complete rewrite," Proceedings of the VLDB Endowment, pp. 1150–1160, 2007.

[2] P. A. Bernstein, V. Hadzilacos, & N. Goodman, "Concurrency control and recovery in database systems", Reading: Addison-wesley, vol. 370, 1987.

[3] R. Guerraoui and J. Wang, "How fast can a distributed transaction commit?", Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database System, pp. 107–122, May 2017.

[4] R. Harding, D. Van Aken, A. Pavlo, & M. Stonebraker, "An evaluation of distributed concurrency control", Proceedings of the VLDB Endowment, vol. 10(5), pp. 553–564, January 2017.

[5] C. Curino, Y. Zhang, E. P. C. Jones & S. Madden, "Schism: a workload-driven approach to database replication and partitioning", Proceedings of the VLDB Endowment, vol. 3(1), pp. 48–57, 2010.

[6] S. Das, D. Agrawal & A. El Abbadi, "G-Store: a scalable data store for transactional multi key access in the cloud", Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 163–174, June 2010.

[7] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K. L. Tan & Z. Wang, "Towards a non-2PC transaction management in distributed database systems", Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1659–1674, 2016.

[8] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao & D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems", Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1–12, 2012

[9] K. Ren, D. Li & D. J. Abadi, "SLOG: serializable, low-latency, geo-replicated transactions" Proceedings of the VLDB Endowment, vol. 12(11), pp. 1747–1761, 2019.

[10] Y. Lu, X. Yu, L. Cao & S. Madden, "Epoch-based commit and replication in distributed OLTP databases", Proceedings of the VLDB Endowment, vol. 14(5), pp. 743–756, 2021.

[11] D. J. DeWitt, Randy H. Katz, Frank Olken, L. D. Shapiro, M. Stonebraker & D. A. Wood, "Implementation Techniques for Main Memory Database Systems", SIGMOD'84, Proceedings of Annual Meeting, pp. 1–8, 1984.

[12] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal & L. Alvisi, "Obladi: oblivious serializable transactions in the cloud", 13th USENIX Symposium on Operating Systems Design and Implementation, pp. 727–743, 2018.

[13] S. Tu, W. Zheng, E. Kohler, B. Liskov & S. Madden, "Speedy transactions in multicore in-memory databases", ACM SIGOPS 24th Symposium on Operating Systems Principles, pp. 18–32, 2013.

[14] N. Narula, C. Cutler, E. Kohler & R. T. Morris, "Phase Reconciliation for Contended In-Memory Transactions", 11th USENIX Symposium on Operating Systems Design and Implementation, pp. 511–524, 2014.

[15] D. Ongaro and J. K. Ousterhout, "In Search of an Understandable Consensus Algorithm", USENIX Annual Technical Conference, pp. 305–319, 2014.

[16] L. Lamport, "The part-time parliament", ACM Trans. Comput. Syst., vol. 16(2), pp. 133–169, 1998.

[17] P. Garraghan, P. Townend, & J. Xu, "An empirical failure-analysis of a large-scale cloud computing environment", IEEE 15th International Symposium on High-Assurance Systems Engineering, pp. 113–120, January 2014.

[18] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann & T. Engbersen, "Failure analysis of virtual and physical machines: patterns, causes and characteristics", 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 1–12, 2014.

[19] G. Wang, L. Zhang & W. Xu, "What can we learn from four years of data center hardware failures¿', 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 25–36, 2017.

[20] N.T.J. Bailey, "On Queueing Processes with Bulk Service", Journal of the Royal Statistical Society, Series B 16, 1, pp. 80–87, 1954.

[21] TPC-C Benchmark C "https://www.tpc.org/tpcc/", 2010.

[22] R. Tarjan, "Depth-first search and linear graph algorithms", SIAM journal on computing, vol. 1(2), pp.146–160, 1972.

[23] I. Mitrani, "Simulation techniques for discrete event systems", Cambridge University Press, 1982.

[24] A. Thomson and D. J. Abadi, "The case for determinism in database systems", Proceedings of the VLDB Endowment, vol. 3(1), pp. 70–80, 2010.

[25] K. Ren, A. Thomson & D. J. Abadi, "An evaluation of the advantages and disadvantages of deterministic database systems", Proceedings of the VLDB Endowment, vol. 7(10), pp. 821–832, 2014.

[26] Y. Lu, X. Yu, L. Cao & S. Madden, "Aria: a fast and practical deterministic OLTP database", Proceedings of the VLDB Endowment, vol. 13(11), pp. 2047–2060, 2020.

[27] S. Issa, M. Viegas, P. Raminhas, N. Machado, M. Matos & P. Romano, "Exploiting symbolic execution to accelerate deterministic databases", 40th IEEE International Conference on Distributed Computing Systems, pp. 678–688, 2020.

[28] J. Du, S. Elnikety & W. Zwaenepoel, "Clock-SI: snapshot isolation for partitioned data stores using loosely synchronized clocks", IEEE 32nd Symposium on Reliable Distributed Systems, pp. 173–184, 2013.

[29] S. Elnikety, W. Zwaenepoel & F. Pedone, "Database Replication Using Generalized Snapshot Isolation", 24th IEEE Symposium on Reliable Distributed Systems, pp. 77–84, 2005.

[30] Y. Sovran, R. Power, M. K.. Aguilera & J. Li, "Transactional storage for geo-replicated systems", Proceedings of the 23rd ACM Symposium on Operating Systems Principles, pp. 385–400, 2011.

[31] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, & I. Stoica, "Highly available transactions: virtues and limitations", Proceedings of the VLDB Endowment, vol. 7(3), pp. 181–192, 2013.

[32] T. Kraska, G. Pang, M. J. Franklin, S. Madden & A. D. Fekete, "MDCC: multi-data center consistency", Eighth Eurosys Conference, pp. 113–126, 2013.

[33] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy & D. R. K. Ports, "Building consistent transactions with inconsistent replication", Proceedings of the 25th Symposium on Operating Systems Principles, pp. 263–287, 2015.

[34] H. Fan and W. M. Golab, "Ocean vista: gossip-based visibility control for speedy geo-distributed transactions", Proceedings of the VLDB Endowment, vol. 12(11), pp. 1471–1484, 2019.

[35] S. Mu, L. Nelson, W. Lloyd & J. Li, "Consolidating concurrency control and consensus for commits under conflicts", 12th USENIX Symposium on Operating Systems Design and Implementation, pp. 517–532, 2016.

[36] F. Nawab, V. Arora, D. Agrawal & A. El Abbadi, "Minimizing commit latency of transactions in geo-replicated data stores", Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1279–1294, 2015.

[37] S. Maiyya, F. Nawab, D. Agrawal & A. El Abbadi, "Unifying consensus and atomic commitment for effective cloud data management" Proceedings of the VLDB Endowment, vol. 12(15), pp. 611–623, 2019.

[38] Taft et al., "CockroachDB: the resilient geo-distributed SQL database", Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1493–1509, 2020.

[39] Y. Lu, X. Yu & S. Madden, "STAR: scaling transactions through asymmetric replication", Proceedings of the VLDB Endowment, vol. 12(11), pp. 1316–1329, 2019.

[40] H. Wu, Z. Shang and K. Wolter, "Performance Prediction for the Apache Kafka Messaging System", IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems, pp. 154-161, 2019.

[41] T. Meng, Y. Zhao, K. Wolter & C. Xu, "On consortium blockchain consistency: a queueing network model approach", IEEE Trans. Parallel Distributed Syst., vol. 32(6), pp. 1369–1382, 2021.

[42] Yu, Philip S., D. M. Dias & S. S. Lavenberg. "On the analytical modeling of database concurrency control", Journal of the ACM, vol. 40(4), pp. 831–872, 1993.

[43] R. Agrawal and D. J. DeWitt, "Integrated concurrency control and recovery mechanisms: design and performance evaluation", ACM Trans. Database Syst., vol. 10(4), pp. 529–564, 1985.

[44] P. di Sanzo, B. Ciciani, R. Palmieri, F. Quaglia and P. Romano, "On the analytical modeling of concurrency control algorithms for software transactional memories: the case of commit-time-locking", Perform. Evaluation, vol. 69(5), pp. 187–205, 2012.

[45] A. Menasce and T. Nakanishi, "Performance evaluation of a two-phase commit based protocol for DDBS", Proceedings of the ACM Symposium on Principles of Database Systems, pp. 247–255, 1982.

[46] M.L. Liu, D. Agrawal & A. El Abbadi, "The performance of two phase commit Protocols in the presence of site failures", Distributed and Parallel Databases, vol. 6, pp. 157–182, 1998.

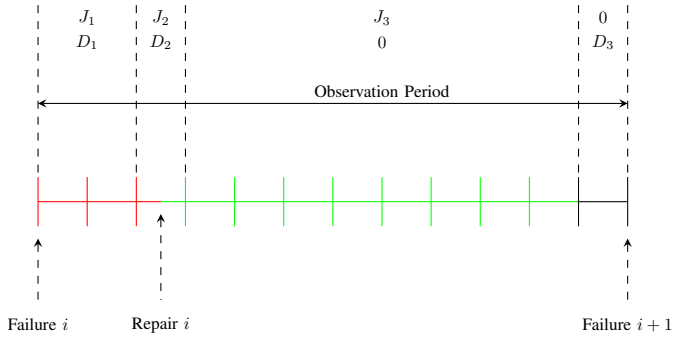


Fig. 7: Observation period with cycles having $N-1$ operative nodes, $N-1$ and N operative nodes, and N operative nodes. (Repair instant can fall anywhere in the mixed cycle.)

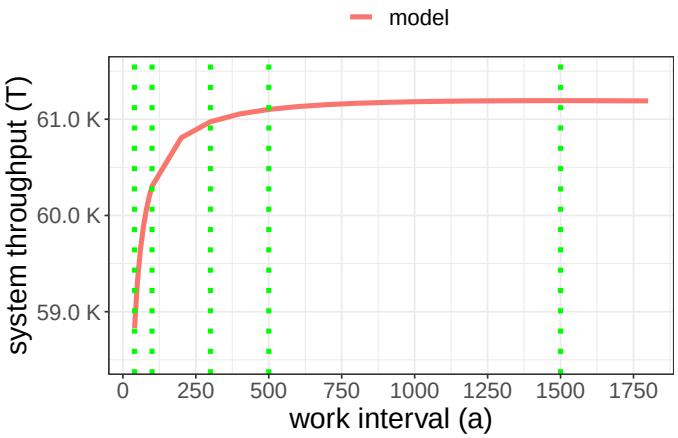
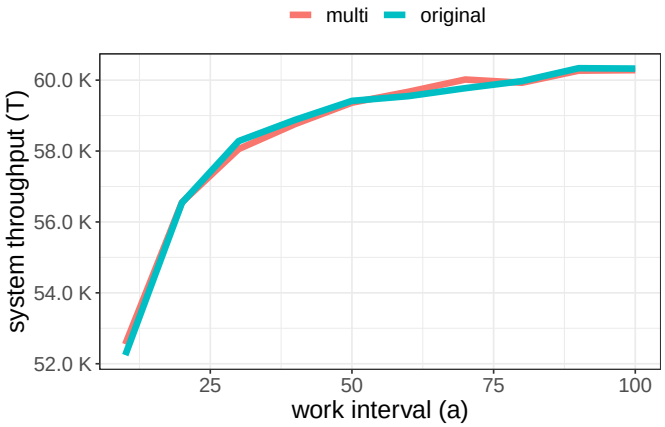
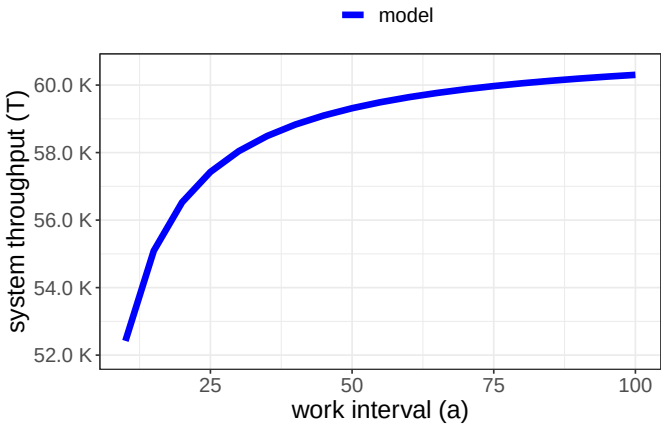


Fig. 8: System throughput estimates $vs.$ a. Green dotted lines indicate regions with different gradients (as discussed in Section V-A).

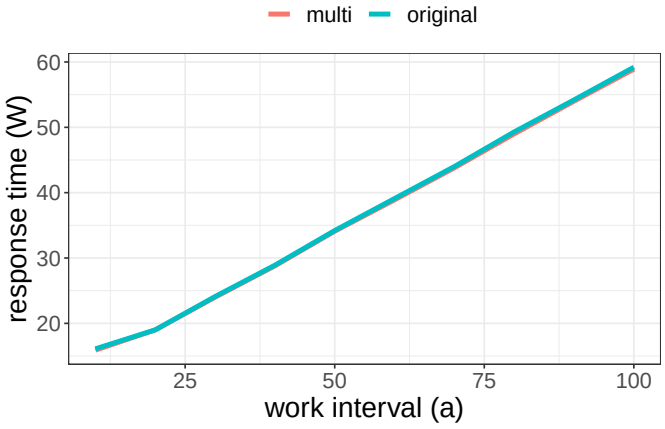


(a) Simulations using paired affinity.

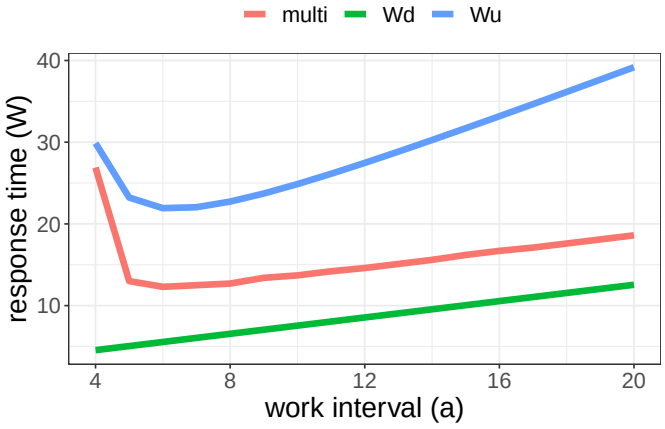


(b) Model using random affinity.

Fig. 9: Maximum throughput as work interval a varied from 10 to 100 ms.



(a) Simulations with paired affinity as a is varied from 10 to 100 ms ($\lambda = 40,000$).



(b) Multi-commit with paired affinity and models with random affinity as a is varied from 4 to 20 ms ($\lambda = 30,000$).

Fig. 10: Average response time (ms).