C5: Cloned Concurrency Control that Always Keeps Up

Jeffrey Helt
Princeton University
jhelt@cs.princeton.edu

Abhinav Sharma
Meta Platforms
abhinavsharma@fb.com

Daniel J. Abadi
University of Maryland, College Park
abadi@cs.umd.edu

Wyatt Lloyd
Princeton University
wlloyd@princeton.edu

Jose M. Faleiro
Unaffiliated
jmfaleiro.work@gmail.com

ABSTRACT

Asynchronously replicated primary-backup databases are commonly deployed to improve availability and offload read-only transactions. To both apply replicated writes from the primary and serve read-only transactions, the backups implement a cloned concurrency control protocol. The protocol ensures read-only transactions always return a snapshot of state that previously existed on the primary. This compels the backup to exactly copy the commit order resulting from the primary's concurrency control. Existing cloned concurrency control protocols guarantee this by limiting the backup's parallelism. As a result, the primary's concurrency control executes some workloads with more parallelism than these protocols. In this paper, we prove that this parallelism gap leads to unbounded replication lag, where writes can take arbitrarily long to replicate to the backup and which has led to catastrophic failures in production systems. We then design C5, the first cloned concurrency protocol to provide bounded replication lag. We implement two versions of C5: Our evaluation in MyRocks, a widely deployed database, demonstrates C5 provides bounded replication lag. Our evaluation in Cicada, a recent in-memory database, demonstrates C5 keeps up with even the fastest of primaries.

PVLDB Reference Format:

Jeffrey Helt, Abhinav Sharma, Daniel J. Abadi, Wyatt Lloyd, and Jose M. Faleiro. C5: Cloned Concurrency Control that Always Keeps Up. PVLDB, 16(1): 1 - 14, 2022.
doi:10.14778/3561261.3561262

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/princeton-sns/c5.

1 INTRODUCTION

Asynchronously replicated primary-backup databases are the cornerstones of many applications [2, 23, 26, 35, 62]. In these systems, after the primary executes a transaction, it sends the resultant writes to a set of backups. The backups apply the writes to reconstruct the primary's state and execute read-only transactions against their local state. To simultaneously execute writes and read-only transactions, a backup implements a *cloned concurrency control* protocol. In addition to providing availability if the primary fails, these protocols improve the database's performance: throughput is increased by serving reads from many backups and latency is reduced by serving reads from a nearby backup.

To reap these benefits without breaking overlying applications, a cloned concurrency control protocol must guarantee *monotonic prefix consistency*, where it exposes a progressing sequence of the primary's recent states to read-only transactions. This ensures backups never return values from states that did not exist on the primary, thereby helping maintain application invariants.

But monotonic prefix consistency makes no guarantees about how quickly writes replicate to the backup. In theory, they could be delayed indefinitely. To be reliable, a cloned concurrency control protocol must also guarantee *bounded replication lag*. Intuitively, a transaction's replication lag is the time between when its writes are first observable by reads on the primary and backup. By guaranteeing bounded replication lag, a cloned concurrency control protocol ensures transactions always appear promptly.

Guaranteeing bounded replication lag is important; significant lag has led to catastrophic failures. For instance, GitLab was unavailable for eighteen hours after a workload change caused such significant lag that replication stopped entirely. In the process of fixing the issue, user data was lost [17, 18]. Similarly, several times in past years, Meta routed all user requests away from a data center because too many of that location's backups had excessive lag.

To guarantee bounded replication lag, a cloned concurrency control protocol must apply writes with as much parallelism as was used by the primary's concurrency control protocol. But guaranteeing monotonic prefix consistency severely constrains the cloned concurrency control protocol, making it difficult to execute with sufficient parallelism. For example, consider two concurrent transactions with both conflicting and non-conflicting writes. If the primary employs two-phase locking [4], the non-conflicting writes can execute in parallel, and the commit order is determined by the lock acquisition order on the first conflicting write. Once their commit order is chosen, however, monotonic prefix consistency mandates that a backup's state reflects it. Thus, the cloned concurrency control protocol must ensure the transactions are serialized correctly, potentially constraining its parallelism.

In the past, slow I/O devices bottlenecked the primary and backup, dominating differences in parallelism. But low-latency persistent storage and large main memories removed this bottleneck, so the primary's concurrency control and the backup's cloned concurrency control protocols are now directly competing.

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:10.14778/3561261.3561262

Existing protocols differ in how much parallelism they leverage while executing writes. On one end of the spectrum are single-threaded protocols [43, 51]. On the other are transaction- [21, 29, 40, 45] and page-granularity [2, 48, 62] protocols. The former execute non-conflicting transactions in parallel; the latter execute writes to different pages in parallel. Such protocols have the potential to keep up with similarly restricted primaries, e.g., single-threaded cloned concurrency control with a single-threaded primary. But no existing protocol can keep up with an unrestricted primary.

In fact, existing protocols cannot keep up with a primary that uses two-phase locking; there are workloads where such a primary always executes with more parallelism. Using these workloads, we prove neither class of protocols guarantees bounded replication lag. In turn, implementations of these protocols are not reliable because changes to the workload or the primary's concurrency control can suddenly lead to unbounded replication lag.

In this paper, we present C5, the first cloned concurrency control protocol to provide bounded replication lag. To always keep up, C5's insight is that the backup's protocol must execute writes at the same granularity as the primary's concurrency control protocol. Thus, its cloned concurrency control has commensurate constraints (C5) with the primary. Because the primary executes writes to non-conflicting rows in parallel, C5 uses a row-granularity protocol.

But row-granularity execution introduces several challenges. First, applying individual row writes to the backup's state can lead to permanent violations of monotonic prefix consistency, where the backup's state ceases to match the primary's. To avoid such violations, C5's scheduler calculates the necessary metadata for its workers to correctly order writes to each row. Second, row-granularity execution does not guarantee monotonic prefix consistency for read-only transactions because transactional atomicity and commit order are not necessarily respected. Imposing additional constraints on workers, however, could reintroduce replication lag. Instead, C5's snapshotter uses three progressing snapshots, ensuring reads observe a consistent state without constraining execution.

We show formally that a row-granularity protocol never imposes more constraints on the backup's execution than a valid concurrency control protocol imposes on the primary's. Thus, C5 can, in theory, always match the primary's parallelism.

In practice, however, row-granularity execution is necessary but not sufficient to provide bounded replication lag. Other bottlenecks, such as a slow scheduler, may get in the way. We thus implement two versions of C5, C5-MyRocks and C5-Cicada, to confirm it always keeps up. C5-MyRocks is backward-compatible and deployed in production at Meta. Making it backward-compatible, however, required some additional constraints to the parallelism in our design. We thus also implemented C5-Cicada, which faithfully implements our design (without additional constraints) and demonstrates C5 can keep up with a cutting-edge concurrency control protocol.

We compare C5-MyRocks and C5-Cicada to a state-of-the-art, transaction-granularity protocol [21]. While it keeps up on some workloads, unbounded replication lag is lurking nearby: Simple optimizations that improve the primary's throughput cause the protocol to lag. In contrast, our implementations always keep up.

In sum, this paper's contributions stem from our key insight that cloned concurrency control must have commensurate constraints with the primary: first, we prove neither a transaction-granularity nor a page-granularity protocol can always keep up with a two-phase locking primary; next, we prove a commensurate-granularity protocol has the potential to keep up with an unrestricted primary; finally, we describe such a protocol, C5, implement two versions of it, and demonstrate both always keep up in practice.

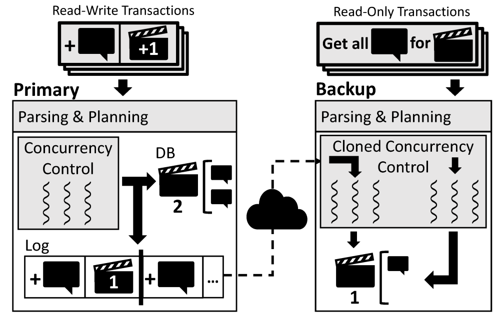


Figure 1: Transaction processing in primary-backup.

Further, Section 8 describes experience from deploying C5 at Meta. Our experience echoes our evaluation: The simple single-threaded cloned concurrency control that was previously deployed could often keep up with the primary, but large replication lag would be exposed by workload changes. The deployment of C5 eradicated these issues and led to noticeably better reliability.

2 BACKGROUND

This section provides background on primary-backup databases, cloned concurrency control protocols, and their guarantees.

2.1 Motivating Example

Throughout the paper, we use the following motivating example. Alice, Bob, and Charlie use a social media platform to share and comment on videos. The platform stores its videos and comments in a database. One table stores each video's name and metadata, including a per-video comment counter; a second table stores each comment's text and metadata. When a user comments on a video, an application server executes a transaction of two operations: it first inserts a new row in the comment table and then increments the video's counter.

The platform replicates the primary's database at a set of backups. The primary implements a concurrency control protocol, and each backup implements a cloned concurrency control protocol.

2.2 Primary-Backup Replication

Figure 1 shows an overview of the primary and backup's processing as they execute transactions. For each operation in a read-write transaction, the primary parses it and plans its execution. Each plan, which may include row queries, local computation, and row writes (i.e., inserts, updates, and deletes), is then executed. For instance, to increment a video's comment counter, the primary reads the counter's current value from the video's row in the video table, increments it, and writes the result back to the row. After all operations execute, the transaction commits by writing to the primary's database and flushing a log of its changes to stable storage.

The primary then sends a copy of its log to the backup. The log reflects a total order of the writes applied by the primary, determined by the primary's transaction commit order and the order of each transaction's operations. The log includes, for each transaction, the written rows and metadata to demarcate its writes from those of others [11, 21, 29, 36, 40, 41, 43, 45, 49].

The backup's cloned concurrency control protocol reads the operations in the log and schedules them for execution by worker threads, bypassing parsing and planning. The workers apply operations to the backup's copy of the database. The protocol also executes read-only transactions using separate threads.

2.3 Monotonic Prefix Consistency

While an asynchronously replicated backup's state inevitably lags, it is ideally otherwise indistinguishable from the primary. Intuitively, the backup should expose a progressing sequence of the primary's recent states. This intuitive behavior is provided by many existing systems [2, 11, 21, 29, 36, 40, 41, 43, 45, 48, 49, 51, 62, 63]. We refer to this guarantee here as *monotonic prefix consistency* (MPC).

We define monotonic prefix consistency relative to the primary's log of transactions. It comprises two guarantees: First, the backup's state must reflect the changes of a contiguous prefix of transactions. Second, the sequence of states exposed to read-only transactions must reflect prefixes of monotonically increasing length.

In our example application, MPC ensures read-only transactions never see a mismatch between the number of comments on a video and the video's comment counter; each transaction's changes appear atomically. Further, MPC ensures comments never seem to disappear. Once a comment becomes visible to a user, all future states exposed by the same backup will include it. Although beyond our scope here, MPC can be guaranteed across multiple backups using sticky sessions [57] or with client-tracked metadata.

Monotonic prefix consistency also maintains implicit application invariants. For instance, suppose Alice first updates her default video permissions to share her future videos only with Bob and then uploads a new video. To make these changes, a transaction first updates her default access control list to only include Bob, and a subsequent transaction adds the new video. An implicit invariant, that Charlie should not see the new video, is expressed by the order of the two transactions. MPC preserves such invariants because states always reflect contiguous prefixes of the log.

2.4 Bounded Replication Lag

Monotonic prefix consistency specifies a cloned concurrency control protocol's correctness but does not clarify its performance requirements. For instance, if Alice calls Bob after commenting on his video, Bob should ideally see her new comment by the time he receives her call. Given only MPC, the comment may be delayed at the backup for an arbitrarily long time.

We define *replication lag* as the time between when a transaction's changes are included in the state returned by the primary and backup. (For the purposes of this paper, we assume the log is always delivered promptly to the backup.) More precisely, we say a transaction T is included in the state returned by the primary or backup once either its writes or later writes are returned to reads. To include a transaction in the returned state requires the backup's protocol to do one of the following: (1) it can eagerly apply the transaction's changes to its copy of the database, making them visible to future reads without additional processing beyond that required to execute the read at the primary [2, 21, 29, 36, 40, 43, 45, 48, 49, 51, 62]; or (2) it can defer part of the execution of the transaction's changes until a corresponding read arrives [63]. For each T, we then define $f_p(T)$ and $f_b(T)$ as the real time when the primary and backup respectively include T in their state. For eager protocols, $f_b(T)$ is the first time at which an arriving read would see T. For lazy protocols, $f_b(T)$ is the first time at which an arriving read would see T, plus the additional time required to finish any deferred execution.

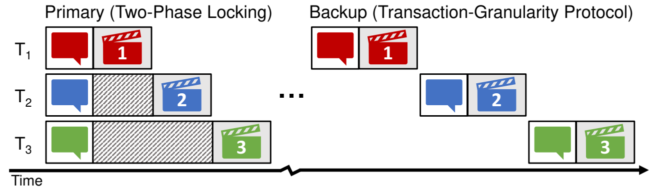


Figure 2: Primary (2PL) and backup (transaction-granularity protocol) executions when three users comment on the same video. Diagonal lines depict waiting for a lock.

A cloned concurrency control protocol guarantees *bounded replication lag* if there exists some finite time L such that for all workloads W and for all transactions T in W, $f_b(T) - f_p(T) \leq L$. (Transactions and workloads are defined more precisely in Section 3.1.) In practice, guaranteeing bounded lag ensures Bob never waits long to see Alice's comment.

3 UNBOUNDED LAG IN EXISTING PROTOCOLS

Guaranteeing bounded replication lag is challenging. To satisfy monotonic prefix consistency, the backup's cloned concurrency control protocol must ensure the backup's state converges to the primary's. To accomplish this, existing protocols serialize conflicting writes [2, 21, 29, 36, 40, 43, 45, 48, 49, 51, 62, 63].

Serialization limits the backup's parallelism. But to always guarantee bounded replication lag, the backup's protocol must be able to match the parallelism used by an unrestricted primary's concurrency control protocol on every workload. Otherwise lag can grow arbitrarily large in some cases.

Transaction- and page-granularity cloned concurrency control protocols are the current best approaches. The former assume logical logs, and the latter assume physical redo logs [2, 48, 62]. In transaction-granularity protocols, writes conflict if they modify the same row, and the protocol serializes transactions with conflicting writes [21, 29, 40, 45]. Page-granularity protocols serialize writes to each page [2, 48, 62]. Both, however, fail to guarantee bounded replication lag because for some workloads, a primary executes with more parallelism.

We show how a transaction-granularity protocol can lag by returning to our motivating example. Suppose Alice, Bob, and Charlie simultaneously comment on the same video. Figure 2 shows a primary and backup's executions of the six resultant operations. The primary uses two-phase locking [4] and stored procedures (i.e., no parsing and planning). The backup implements a transaction-granularity protocol [21, 29, 40, 45].

On the primary, three threads insert rows in the comments table in parallel, but updates to the video's comment counter are serialized by a row lock. On the backup, however, the transaction-granularity protocol serially executes all of the operations. Even if the backup uses different workers to execute each transaction (as shown), their execution is not faster than that of one worker. Thus, a fundamental gap exists between the parallelism available to the primary and backup, which can cause arbitrarily long lag.

Page-granularity protocols have similar issues. Concurrency control protocols using logical locking may allow concurrent transactions to update distinct rows residing on the same physical page [4, 47]. Such concurrency control can cause arbitrarily long replication lag when paired with page-granularity cloned concurrency control because writes that execute in parallel on the primary are serialized on the backup. Thus, a fundamental gap again exists.

In the remainder of this section, we prove this problem is general to all transaction-granularity protocols, and thus, no such protocol guarantees bounded replication lag. We subsequently summarize the comparable theorem for page-granularity protocols.

3.1 Transaction-Granularity Protocols

System Model. A *database* $\mathcal{D}$ stores sets $\mathcal{K}$ of *keys* and $\mathcal{V}$ of *values*. The database's *state* is a mapping from $\mathcal{K}$ to $\mathcal{V}$. A *transaction* T is an ordered set of *operations* (reads and writes) on individual keys. For simplicity, we assume each value is uniquely identifiable, so two identical transactions are, too. We define $\mathfrak{R}(T)$ and $\mathfrak{W}(T)$ as the sets of keys read and written by the operations in T. We define transaction arrival times at the primary and backup as $a_p(T)$ and $a_b(T)$, respectively. Finally, we define the real time at which a transaction is included in the primary and backup's state as $f_p(T)$ and $f_b(T)$, respectively (as in Section 2.4).

We assume a primary-backup system where both have *m cores*. In isolation, the primary's cores each execute an operation in $e > 0$ time units. The primary uses 2PL [4], so an operation may wait for a lock if there is a concurrent operation on the same key. If there are multiple conflicting operations, assume they are granted the lock in the order requested. To account for both eager and lazy cloned concurrency control protocols, assume the backup's cores execute each operation in $0 < d \leq e$ time units. We assume $d \leq e$ because backups often avoid some processing.

When the primary finishes executing transaction T's operations, it records T's writes in its log. $T_1 \prec T_2$ denotes T_1 precedes T_2 in the log. The log is then sent to the backup. For simplicity, we assume this occurs instantaneously.

A *workload* $W \in \mathcal{W}$ is a tuple $(\mathcal{T}, A_{\mathcal{T}})$ where $\mathcal{T}$ is a set of transactions and $A_{\mathcal{T}}$ is a function from $\mathbb{R}$ to finite sets of transactions $T \in \mathcal{T}$ representing the transaction arrival process at the primary. $\mathcal{W}$ is the set of all definable workloads.

Transaction T's *replication lag* is given by $f_b(T) - f_p(T)$. A cloned concurrency control protocol has finite replication lag for workload $W = (\mathcal{T}, A_{\mathcal{T}})$ if there exists some finite L such that for all $T \in \mathcal{T}$, $f_b(T) - f_p(T) \leq L$, and it guarantees bounded replication lag if it has finite replication lag for all $W \in \mathcal{W}$.

Definitions & Assumptions. A *transaction-granularity cloned concurrency control protocol* guarantees that for all pairs of transactions T_1 and T_2, if $\mathfrak{W}(T_1) \cap \mathfrak{W}(T_2) \neq \emptyset$ and $T_1 \prec T_2$, then all of T_1's writes execute before any of T_2's. (This definition matches existing implementations [21, 29, 40, 45].)

The proof below requires $m > \lceil \frac{e}{d} \rceil$, but this assumption is reasonable in practice. Server CPUs commonly contain at least 64 physical cores [25]. Thus, the assumption is not satisfied only if the backup executes operations more than 63 times faster than the primary. Stored procedures and sophisticated concurrency control protocols [30, 44, 61, 64] make such an advantage unlikely.

Theorem 1. *If $m > \lceil \frac{e}{d} \rceil$, then a primary-backup system using two-phase locking on its primary and a transaction-granularity cloned concurrency control protocol on its backup cannot guarantee bounded replication lag.*

Proof. Assume we have a primary-backup system as described above, and assume to contradict that it guarantees bounded replication lag. Then there exists some L such that for all $W \in \mathcal{W}$, the system executes all transactions in W with replication lag $\leq L$.

We now construct a workload $W \in \mathcal{W}$ that includes at least one transaction with replication lag greater than L. Each transaction comprises $m \geq n > \lceil \frac{e}{d} \rceil$ writes, and there are $\lceil \frac{L}{nd-e} \rceil$ such transactions. Because $nd > e$, the number of transactions is well-defined. The first $n-1$ writes of each transaction modify unique keys, and the last updates key k_0. Define $A_{\mathcal{T}}$ such that a new transaction arrives at the primary every e time units, starting at time 0.

Because the primary uses 2PL, it executes the first $n-1$ writes of each transaction in parallel but serializes their final updates to k_0. For convenience, we index the transactions in the order they appear in the primary's log.

For the first set of m transactions, $f_p(T_0) = ne, \ldots,$ and $f_p(T_{m-1}) = (n+m-1)e$. Because $m \geq n$, the core that executed T_0 is free when T_m arrives. Thus, T_m finishes e time units after T_{m-1}. In general, we see $f_p(T_i) = (n+i)e$.

The backup uses a transaction-granularity protocol, so it serially executes all writes in the workload. Thus, the backup finishes executing T_0 at $n(e+d)$. By construction, $nd > e$, so $f_b(T_0) > f_p(T_1)$. Thus, the backup immediately starts executing T_1 after T_0. The same is true for all subsequent transactions. In general, we see $f_b(T_i) = ne + (i+1)nd$.

Thus, in general, $f_b(T_i) - f_p(T_i) = ne + (i+1)nd - (n+i)e = i(nd-e)+nd$. For the final transaction T in the workload, $i = \lceil \frac{L}{nd-e} \rceil$, and thus $f_b(T) - f_p(T) = \lceil \frac{L}{nd-e} \rceil (nd-e)+nd \geq \frac{L}{nd-e}(nd-e)+nd > \frac{L}{nd-e}(nd-e)$. Equivalently, $f_b(T) - f_p(T) > L$, a contradiction. □

The result above shows that if the primary has sufficient cores, then a transaction-granularity protocol cannot guarantee bounded replication lag. To simplify our formalism, the proof assumes the primary uses 2PL and serializable isolation [3, 50]. We note three important extensions: First, the theorem applies if the primary uses weaker isolation [1] because it can only accelerate the primary.

Second, a similar result can be derived for some optimistic protocols [4, 33]. For example, a similar execution to the one in Figure 2 is possible with multi-version timestamp ordering (MVTSO) [4]. Using MVTSO, the three transactions still insert comments in parallel. If they then read the comment counter, write its new value, and perform validation serially, in timestamp order, all three transactions will commit, and a fundamental gap will again exist. We leave the generalization of our formal framework to optimistic concurrency control to future work.

Third, because the above proof assumes $0 < d \leq e$, it also applies if there is primary-specific processing, such as parsing and planning. This additional processing can be accounted for by increasing e, and the theorem holds as long as $m > \lceil \frac{e}{d} \rceil$. If it does not, then the primary-backup system may be able to guarantee bounded replication lag but only because bottlenecks on the primary make it easy for an inefficient cloned concurrency control protocol to keep up, for example, if bottlenecks in logging, persistence, or log transfer restrict the primary's parallelism.

Despite these cases, however, solving replication lag remains urgent: First, there are many cases where these are not bottlenecks. For parsing and planning, many deployments use stored procedures. For persistence, mechanisms such as early lock release [7, 16, 27] and epoch-based group commit [6, 61] help decouple transaction throughput from I/O latency. Second, we expect advances in research and technology, such as non-volatile memory [24], to eventually remove these bottlenecks.

3.2 Page-Granularity Protocols

Databases historically assumed their data resided on disk. Thus, when persisting writes, they often locked data pages while flushing changes to disk [47, 51]. Page-granularity cloned concurrency control protocols [2, 48, 62] leverage this fact to match their primary's granularity: Since writes to the same page are serialized on the primary, the backup can keep up with it despite also serializing writes to the same page.

The proof described below, however, shows page-granularity protocols cannot keep up with an unrestricted primary. In practice, we believe they are more likely to keep up with an unrestricted primary than a transaction-granularity protocol because they impose fewer constraints on parallelism. For instance, writes may mostly be spread across different pages, which is the best case for a page-granularity protocol. Yet our result shows they do not fundamentally solve the replication lag problem. Optimizations to the primary and changes to workloads can lead to unbounded lag.

3.2.1 Proof Summary. We use a structurally similar proof for page-granularity protocols, but we omit it for brevity. (It can be found in our technical report [20].) The proof shows that if the primary has sufficient cores, can fit enough rows on each page, uses 2PL [4], and guarantees serializable isolation [50], then a page-granularity protocol cannot guarantee bounded replication lag. It does so by constructing a workload where many writes to the same page are executed in parallel on the primary but are serialized on the backup.

The assumption about the number of cores is identical to the one in the proof above. Unlike the previous proof, however, we additionally assume that the number of rows that can fit on a page is greater than $\lceil \frac{e}{d} \rceil$. But this assumption is again reasonable in practice. With a typical cache line size of 64 B—more than enough to store a row with two integer columns—and a page size of 4 KiB, 64 rows can be stored on the same page, each on a different cache line. Thus this assumption will hold provided the backup cannot execute operations more than 63 times faster than the primary.

4 C5 DESIGN

C5 achieves two competing goals: it ensures bounded replication lag and guarantees monotonic prefix consistency for read-only transactions. To accomplish this, C5 comprises three components: a scheduler, a set of workers, and a snapshotter. The scheduler and workers ensure bounded replication lag by executing writes at a sufficiently fine granularity, and the snapshotter guarantees read-only transactions only see changes that are valid under monotonic prefix consistency. Together they implement C5's row-granularity cloned concurrency control protocol.

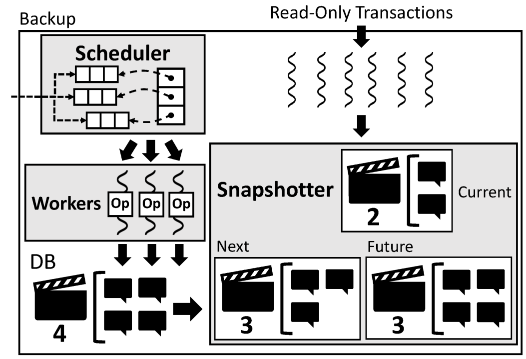


Figure 3: C5's scheduler, workers, and snapshotter.

As shown in Section 3, transaction- and page-granularity protocols fail to provide bounded replication lag because they cannot always execute with the same parallelism as the primary. The primary executes writes to different rows in parallel, so to provide bounded lag, C5's workers execute writes at row granularity.[1]

Unconstrained row-granularity execution, however, can lead to permanent violations of monotonic prefix consistency because conflicting writes may execute in the wrong order. For instance, suppose two transactions T and U each update rows x and y. If different workers execute the resultant writes (denoted $w_T[x]$, $w_T[y]$, $w_U[x]$, and $w_U[y]$), $w_T[x]$ may finish before $w_U[x]$ and $w_U[y]$ before $w_T[y]$. If there are no further writes to these rows, the backup will forever reflect $w_U[x]$ and $w_T[y]$, violating transactional atomicity and thus MPC. C5's scheduler helps avoid permanent consistency violations by constraining the workers' execution. These constraints ensure writes to each row are applied in the same order as on the primary. Thus, each row reflects monotonically increasing prefixes of the log.

But per-row monotonicity is insufficient to guarantee global monotonic prefix consistency. In the example above, a write to a third row z from a third transaction V may be scheduled after T and U but applied first. If this occurs, then a read-only transaction of rows x, y, and z would violate monotonic prefix consistency. Instead, C5's snapshotter uses a set of three progressing database snapshots to allow uninterrupted execution of non-conflicting writes while guaranteeing MPC.

Figure 3 shows C5's design. The scheduler orders writes and schedules them for execution by the workers. The snapshotter exposes a monotonic-prefix consistent view of the database to read-only transactions, which are executed by a separate set of threads. We now describe C5's components in turn.

4.1 Row-Granularity Scheduling & Execution

As described in Section 2.2, the backup continuously receives a log of operations from the primary, including the rows written by each operation and metadata to delimit transactions. To guarantee bounded replication lag, C5's workers must execute individual row writes while obeying the constraints specified by the scheduler.

[1] Some concurrency control protocols allow two transactions to update a row's cells in parallel [22]. For ease of exposition, we assume they cannot, but rows are not fundamental to our design—C5 could be adapted for finer granularities.

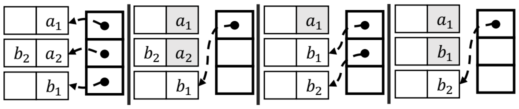


Figure 4: Left to right shows the scheduler's queues as two workers execute four writes. Grey writes are being executed.

To avoid permanent consistency violations, the scheduler logically constructs a FIFO queue for each row whose order reflects the order of the row's writes in the log.

As the scheduler processes writes, it assigns each a sequence number, which reflects the write's position in the log. The scheduler then enqueues the write in the appropriate FIFO queue.

A write is *safe* to execute when it reaches the head of its FIFO queue and the prior head has finished executing. This assumes the backup receives the log of each row's writes and the scheduler processes them in order. (The log shipping subsystems in many commercial databases satisfy this assumption [43, 47, 51].) Given this, the scheduler is assured that when it processes a write, all conflicting writes that precede it in the log are either already in the queue or executing.

To keep replication lag small, the scheduler ensures workers execute safe writes promptly. To do so, the scheduler uses a FIFO queue to order the queues described above. To avoid ambiguity, we refer to a scheduler queue and per-row queues. Thus, a worker chooses the next write for execution by first removing the per-row queue at the head of the scheduler queue and then executing the write at its head. When the worker finishes executing the write, the per-row queue is reinserted into the scheduler queue.

To demonstrate how C5's scheduler and workers operate, we return to our motivating example when Alice and Bob concurrently comment on the same video. Assume Alice's transaction A commits first. It performs two operations: operation a_1 inserts one comment row, and a_2 increments the video's comment counter. Bob's transaction B performs comparable operations b_1 and b_2.

Figure 4 shows how two workers execute the four writes. The first panel shows the initial data structures after processing all operations. a_1 and a_2 then begin executing (second panel). (b_1 remains queued because there are only two workers.) a_2 finishes before a_1, so its corresponding queue is reinserted at the tail of the scheduler queue (third panel). Finally, b_1 starts executing (fourth panel). The workers continue until they execute all of the writes.

We now prove that row-granularity execution never imposes more constraints on the backup than any concurrency control protocol imposes on the primary.

4.1.1 Row-Granularity Execution Can Keep Up. Both the primary's concurrency control and the backup's cloned concurrency control protocols can be viewed as functions from a set of logs to a set of sets of *execution schedules*. Given a log, the primary's threads and the backup's workers execute its writes according to one of the schedules in its image. We say a schedule is *valid* if the schedule of writes produces an equivalent database state as serially executing the writes in the log. In the remainder of this section, we only consider the set of *valid protocols*, those whose images contain only sets of valid schedules, and denote it as $\mathfrak{S}$. Note that a primary's concurrency control protocol is always in $\mathfrak{S}$ because the primary's durability guarantees that its log, when serially executed, reproduces its state.

Let $w_T[x]$ denote a write to row x by transaction T. As before, $T \prec U$ denotes T precedes U in the log, and in a slight abuse of notation, let $w_T[x] \prec w_U[y]$ denote write $w_T[x]$ precedes write $w_U[y]$ in the log. Similarly, $w_T[x] < w_U[y]$ denotes $w_T[x]$ precedes $w_U[y]$ in an execution schedule. A *row-granularity protocol* guarantees that for all logs and all pairs of writes $w_T[x]$ and $w_U[y]$, if $x = y$ and $T \prec U$, then $w_T[x] < w_U[y]$ in all of its schedules.

Theorem 2. *Let $R \in \mathfrak{S}$ be a row-granularity protocol. Given a log, there does not exist a valid protocol $P \in \mathfrak{S}$ that imposes fewer constraints on its corresponding set of execution schedules than R.*

Proof Sketch. Given a log and two transactions T and U such that $T \prec U$, R imposes one constraint on the possible executions of their writes: if $w_T[x]$ conflicts with $w_U[x]$, then $w_T[x] < w_U[x]$.

Assume to contradict there is a valid protocol P that does not impose the above constraint. Then in one of the resulting schedules, $w_U[x] < w_T[x]$. A serial execution of the log, however, always executes $w_T[x]$ before $w_U[x]$ because $T \prec U$ and thus $w_T[x] \prec w_U[x]$. As a result, this execution is not equivalent to the serial execution of the log, contradicting that P is valid. □

The proof shows that all valid concurrency control protocols, regardless of isolation level, must impose, at a minimum, the constraints imposed by a row-granularity protocol. If a backup employs such a protocol, then regardless of how much parallelism is exploited by the primary's concurrency control during its execution, an execution with an equal degree of parallelism is available to the backup. Thus, the proof shows row-granularity execution never hampers the backup's ability to keep up.

The proof, however, elides many practical details. As a result, while it shows that a backup using row-granularity execution can keep up in theory, it does not guarantee that a specific design or implementation will actually execute according to the necessary schedule to keep up in practice.

Regarding the design, a row-granularity scheduler, for instance, may impose more constraints than are strictly necessary for row-granularity execution, and these additional constraints may prevent the backup from keeping up in some cases. Regarding the implementation, a poor one, such as one that improperly uses concurrency mechanisms, may cause the scheduler to bottleneck the backup. Similarly, components outside the scope of a cloned concurrency control protocol may prevent it from keeping up. For instance, row-granularity execution may reduce cache locality when executing writes and could in theory make the backup's workers slower than the primary's threads.

To avoid overly complicating the formalism above, we thus do not claim to prove that a specific design or implementation guarantees bounded replication lag. (We leave these theoretical investigations to future work.) Instead, our experimental evaluations in Sections 6 and 7.3 verify that the principles learned here translate into bounded lag in practice.

4.2 Snapshotter & Read-Only Transactions

C5's snapshotter uses database snapshots to guarantee monotonic prefix consistency without blocking workers. To do so, it requires

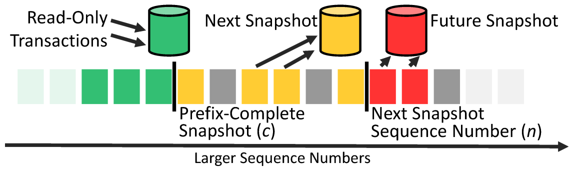
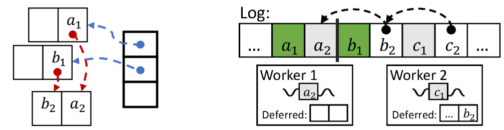


Figure 5: C5's snapshotter. Writes in color (grey) have finished (not finished) executing.



(a) C5-MyRocks **(b) C5-Cicada**

Figure 6: Scheduling and execution in each implementation. Refer to the accompanying text for details.

tight control over the creation and updating of the snapshots (as described below). The required operations are not backward-compatible with the storage engines in some commercial databases [12, 19, 42, 52] but can be implemented efficiently in many modern databases where workers can assign timestamps to their writes [8, 28, 32, 33]. We elaborate on how this difference affects the snapshotters of C5-MyRocks and C5-Cicada in Sections 5.2 and 7.2, respectively.

The snapshotter creates new snapshots from the database. Logically, a snapshot is a sequence of writes and is initially empty. Workers apply writes (i.e., inserts, updates, and deletes) to a snapshot. Two snapshots $\mathcal{S}_1$ and $\mathcal{S}_2$ can be merged to produce a third $\mathcal{S}_3$ that reflects the writes applied to both, with all writes in $\mathcal{S}_1$ ordered before those in $\mathcal{S}_2$. Finally, the latest version of a row's value can be read from a snapshot.

The snapshotter uses the operations above to maintain three database snapshots, logically representing the current, next, and future. The current snapshot is initially empty, always prefix-complete, and serves read-only transactions. The next and future snapshots are initially empty. Workers only modify the next and future snapshots.

Figure 5 illustrates how the snapshotter incorporates a write into the current snapshot while maintaining monotonic prefix consistency. C5's snapshotter uses two sequence numbers to delimit the three snapshots. The current snapshot includes all writes up to sequence number c. All writes with sequence numbers between c and n (inclusive) update the next snapshot; all writes with sequence numbers greater than n update the future snapshot.

When all writes with sequence numbers between c and n finish executing, the current and next snapshots together form a new, prefix-complete snapshot. The snapshotter then merges them, and the result replaces the current. At the same time, it performs four additional operations: c is updated to reflect the new current snapshot; n is advanced; the next snapshot is replaced with the future snapshot; and a new future snapshot is created. (We elaborate on how these steps are implemented in Sections 5.2 and 7.2.)

To satisfy monotonic prefix consistency, the snapshotter always aligns n with a transaction boundary. Thus, the next snapshot always reflects a set of complete transactions before being merged.

Because they execute against different snapshots, workers and read-only transactions execute in parallel. But to guarantee bounded replication lag, workers must be given higher scheduling priority than read-only transactions threads. To avoid starvation, we assume they execute on separate cores (beyond the m assumed in Section 4.1.1), or if they execute on the same cores, there are enough spare CPU cycles to process all read-only transactions.

5 C5-MYROCKS IMPLEMENTATION

C5-MyRocks was developed to solve replication lag at Meta, so backward compatibility and ease of deployment were primary concerns. To remain backward-compatible with MyRocks (a fork of MySQL that uses RocksDB as its storage engine [11, 12, 43]), C5-MyRocks imposes some additional constraints on its execution, beyond those discussed in Section 4. In this section, we describe the implementation, highlighting how it leverages MyRocks's existing features [11, 43] and differs from our design.

5.1 Scheduling & Execution

The scheduler leverages MyRocks's row-based-logging subsystem [11, 43], so C5-MyRocks does not require any changes to the primary or its log. For each operation, the log includes the set of written keys and corresponding values.

The logging subsystem assumes all of a transaction's writes are executed by the same worker. To keep its changeset small (630 lines of C++ code), C5-MyRocks thus also enforces this constraint.

To do so, the scheduler processes a transaction in the log in three steps: First, it builds a linked list of its writes. Second, it adds each write to their per-row queues. Third, it puts the transaction's first write in the scheduler queue. After a worker dequeues one such write, it repeats the following for each write in the transaction: First, it waits until the write reaches the head of its per-row queue (i.e., it is safe to execute), and second, it follows the pointer to the next write, if any, in the transaction.

Figure 6a illustrates C5-MyRocks's data structures after the scheduler processes the two transactions described in the example in Section 4.1. The red arrows here denote pointers linking each transaction's writes, and the blue denote pointers from the scheduler queue to each transaction's first write. Unlike in Figure 4, b_1 (as opposed to a_2) is executed second. Further, when the second worker finishes executing b_1, it will always execute b_2 next, after waiting until a_2 is executed and dequeued. In C5's design, the second worker would be free to execute another write that is immediately safe to execute.

C5-MyRocks's one-thread-per-transaction execution model and having workers pick up transactions in commit order greatly simplified the implementation, but as demonstrated above, its implementation is more constrained than our design. It thus executes some workloads with less parallelism. Nonetheless, our evaluation demonstrates C5-MyRocks keeps up with its primary.

5.2 Snapshotter & Read-Only Transactions

The storage engines in some widely deployed databases [12, 19, 42, 52], including MyRocks, cannot easily implement the operations described in Section 4.2. Unfortunately, they are also complex, comprising tens to hundreds of thousands of lines of code [12, 19]. To again keep C5-MyRocks's changeset small, we opted to implement its snapshotter without requiring changes to RocksDB.

In MyRocks, snapshots are read-only and can only be taken of the database's current state. Neither workers nor the snapshotter have fine-grained control over which writes are included in a snapshot (e.g., by taking a snapshot as of some specified timestamp or version number). As a result, the snapshotter must impose some additional constraints on the workers to ensure the entire database is prefix-consistent when taking a new snapshot.

Instead of three snapshots, C5-MyRocks's snapshotter logically maintains two: It uses a current snapshot, which is always prefix consistent and used to serve read-only transactions. The next snapshot, however, is replaced by the database. c still tracks the writes included in the current snapshot.

To merge the current and next snapshots, the snapshotter performs the following: First, it chooses n, the sequence number of the last write to be included in the next snapshot. To ensure the merged snapshot is prefix-consistent, choosing n also blocks workers from executing writes with sequence numbers greater than n until after the snapshot is taken. (If the storage engine supports transactions, as in RocksDB [12], the workers only need delay committing these writes.) Second, after all writes with sequence numbers between c and n (inclusive) execute, the snapshotter takes a new snapshot of the database and replaces the current one. c and n then advance as described previously. Advancing n also allows blocked workers to proceed with their writes.

If taking a snapshot is computationally expensive, the blocking above may lead to spikes in replication lag. To combat this, our implementation allows database administrators to tune the approximate snapshot frequency, I, in milliseconds. Because the storage engine may not be prefix consistent exactly every I milliseconds, the snapshotter advances n by an estimate of the number of writes workers will execute in the next I milliseconds.

By tuning I, administrators can ensure replication lag returns to satisfactory levels between snapshots provided lag can decrease between snapshots. This implies the backup must execute each write marginally faster than the primary. But we found this assumption reasonable in practice, and our evaluation further supports this.

6 C5-MYROCKS EVALUATION

Our evaluation explores the following questions:

(1) Does C5-MyRocks help engineers avoid potential disasters caused by optimizations of realistic workloads? (§6.1)
(2) Does C5-MyRocks always keep up with the primary? (§6.2)
(3) Does C5-MyRocks guarantee MPC for read-only transactions without causing unbounded replication lag? (§6.3)

Experimental Setup. All experiments ran on the CloudLab Wisconsin platform [9] with three servers located in one data center: one for load generation, the primary, and the backup. Round-trip times between machines were less than 100 µs. Each machine had two 2.20 GHz Intel Xeon processors with ten cores each, hyper-threading disabled, 192 GB of RAM, a 10 Gb NIC, and a 480 GB SSD.

For each experiment, a fixed number of closed-loop clients executed read-write transactions at the primary. The log of writes was then sent to the backup and executed by the cloned concurrency control protocol's workers. The number of clients and workers were set to maximize the primary and backup's throughput, respectively, and there were always fewer workers than primary threads.

For experiments including both read-write and read-only transactions (i.e., Section 6.3), an additional set of closed-loop clients sent read-only transactions to the backup. The backup's workers and read-only threads were pinned to separate cores, and their throughput is shown separately.

All results were from 120-second trials. We omit all data from the first and last 15 seconds of each trial to avoid experimental artifacts. Unless otherwise specified, we ran each experiment five times and report the median result.

To stress the cloned concurrency control, the primary used read committed isolation [3]; further, the log and MyRocks's state on the primary and backup were kept in memory [12]. For all implementations, we used read-free replication and disabled MyRocks's 2PL on the backup [11, 12] since the scheduler already prevents conflicting writes from executing concurrently. In all experiments, memory bandwidth and the network were not bottlenecks.

Workloads. We use three workloads. The first is TPC-C [60], an OLTP benchmark simulating an order-entry application. All experiments use one warehouse, so the database initially contains about 300,000 rows. The other two, insert-only and adversarial, are synthetic. In each, the database contains one table, initially with one row, with two integer columns: a primary key and a value.

Each transaction in the insert-only workload comprises a variable number of unique inserts. Each transaction in the adversarial workload comprises a variable number of unique inserts and one update. The updates set the same row's value to a random integer, so all transactions conflict.

Baselines. KuaFu [21], a state-of-the-art, transaction-granularity cloned concurrency control protocol, is our baseline. KuaFu's protocol is nearly identical to MySQL 8's write-set-based parallel replication [40] and is strictly better than the database-granularity and epoch-based protocols used in earlier versions of MySQL [41, 43] and its variants [36]. We re-implemented KuaFu in MyRocks.

6.1 C5-MyRocks Prevents Potential Disasters

Because software and hardware improvements have accelerated the primary's processing, primary-backup systems using transaction-granularity protocols are brittle. Simple changes to a workload may cause unbounded replication lag. To demonstrate this problem, we use TPC-C [60]. While KuaFu [21] keeps up on the standard benchmark workload, simple optimizations and non-standard transaction mixes cause unbounded replication lag with the same protocol [21]. We discuss the optimizations and our results in turn.

We optimize two of TPC-C's transactions: the NewOrder and Payment transactions [60]. In both cases, we defer higher-contention operations as much as possible while preserving application semantics. (Similar optimizations were observed in prior work [66].) In the NewOrder transaction, the highest contention write is the increment of the district's next order ID. In the Payment transaction, it is the update to the warehouse's balance [60]. Deferring these writes allows more parallelism on the primary.

Figure 7 shows the primary and backup's throughput while executing read-write transactions for a 100% NewOrder and a Payment workload, each before and after optimization. (TPC-C's read-only transactions were not used in these experiments.) For the NewOrder workload, the optimization increases the primary's throughput from 2,527 to 4,067 transactions/s. For the Payment workload, the primary's throughput increases by over 700% from 1,249 to 9,105 transactions/s.

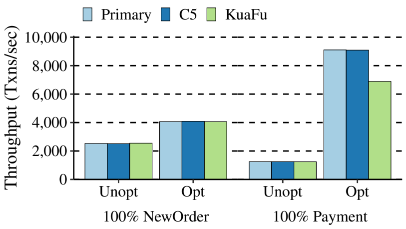


Figure 7: Throughput on 100% NewOrder and 100% Payment before and after optimization.

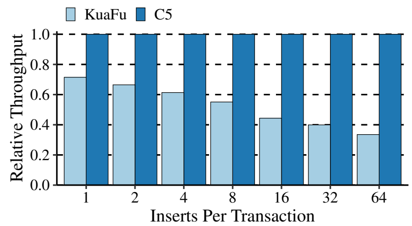


Figure 8: Backup's throughput relative to primary's on adversarial workload as inserts per transaction increases.

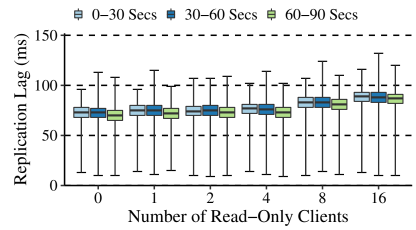


Figure 9: Replication lag of read-write transactions with increasing number of read-only clients.

KuaFu keeps up with the optimized NewOrder workload. Data dependencies between operations limit how late the district row write can be deferred within each transaction and in turn, limits the primary's parallelism. But KuaFu cannot keep up on the optimized Payment workload; its throughput peaks at 6,889 transactions/s. Conversely, C5-MyRocks keeps up.

If the optimization to the Payment transaction were made to a production workload, KuaFu would cause significant replication lag, with 2,216 transactions queuing at the backup every second. This rate is larger than the one that induced replication lag of nearly 2 hours in production at Meta (discussed further in Section 8). On the other hand, C5-MyRocks thwarts such a disaster.

6.2 C5-MyRocks Always Keeps Up

To validate that C5-MyRocks always keeps up, we measured the primary and C5-MyRocks's throughput using the insert-only and adversarial workloads. The two are on opposite ends of the contention spectrum: no transactions conflict in the former, while all do in the latter. We also present KuaFu's results.

Because all transactions are non-conflicting, the insert-only workload stresses the primary's concurrency control and backup's cloned concurrency control. Here, MyRocks's throughput is about 40,500 transactions/s. C5-MyRocks keeps up with the primary, indicating that its scheduling mechanisms have sufficiently low overhead. As expected, KuaFu also keeps up. With both protocols, incoming writes can be executed immediately.

To verify C5-MyRocks's scheduler is not a bottleneck, we ran the same experiment offline. We loaded the primary with inserts as above but delayed replication. Once all writes finished and the resultant log was transferred, we enabled C5-MyRocks's scheduler and workers. We used sufficient workers, so the scheduler was the bottleneck. C5-MyRocks's scheduler processed 95,683 transactions/s, more than double MyRocks's throughput.

Figure 8 shows each implementation's performance on the adversarial workload. We plot the backup's throughput relative to the primary's as we vary the number of non-conflicting inserts per transaction from 1 to 64. Every transaction updates the same row. Despite the high contention, the primary, using 2PL [4], executes the non-conflicting inserts that precede the conflicting update in parallel. Because all transactions conflict, KuaFu serializes them. Thus, the primary's advantage over KuaFu increases with the number of inserts. KuaFu's throughput drops from 70% to just 38% of the primary's. On the other hand, C5-MyRocks executes the non-conflicting inserts in parallel so always keeps up.

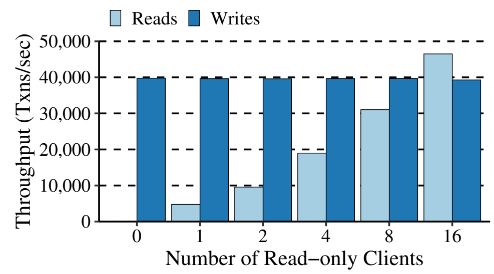


Figure 10: Backup's read-only and read-write transaction throughput as the read-only load increases.

6.3 C5 Serves Reads with Bounded Lag

C5-MyRocks's implementation blocks writes from committing while taking a snapshot. But it also exposes a parameter to tune the frequency of snapshots. With periodic snapshots, C5-MyRocks serves read-only transactions in parallel with writes, and thus, steady-state replication lag remains bounded despite additional load from read-only clients. To validate these claims, we measure replication lag as we increase the read-only load on C5-MyRocks.

Figure 9 plots the distribution of replication lag for read-write transactions over three 30-second periods with the insert-only workload. The whiskers show the minimum and maximum, and the boxes show the quartiles. We show one trial; the results from others were similar. For each read-write transaction, we measure replication lag as the difference between when it commits on the primary and when it is included in the current snapshot. Snapshots were taken every 10 ms. Each read-only transaction executes a random point query on the table's primary key; queries could select a nonexistent key. We vary the number of clients from 0 to 16.

Replication lag remains bounded in all cases and across all time periods. With 16 read-only clients, the median and maximum lag are about 88 ms and 135 ms, respectively. With zero clients, the median lag is about 73 ms. The latter is lower because it avoids contention on some of MyRocks's internal data structures between the read-only threads and workers. We expect this contention can be removed with further optimizations.

Figure 10 plots the backup's throughput of read-only and read-write transactions for the same experiment. C5-MyRocks's throughput always matches the primary's. (Differences in the primary's throughput from prior experiments is due to variations in MyRocks's performance across trials.) Further, C5-MyRocks isolates workers from read-only transactions. With steady write throughput, read-only throughput increases from 4,755 transactions/s with 1 client to 46,500 transactions/s with 16.

7 C5-CICADA

Unlike C5-MyRocks, our implementation of C5 in Cicada [33], an in-memory multi-version database, is faithful to our design. In this section, we first provide background on Cicada. We then describe the implementation of C5-Cicada's scheduler, workers, and snapshotter. We conclude with our evaluation, which demonstrates C5 can keep up with a modern concurrency control protocol.

7.1 Cicada Background

Cicada's multi-version storage engine supports reads, inserts, updates, and deletes. The storage engine is implemented as an array indexed by an internal row ID. (Externally meaningful keys are mapped to row IDs through indices.) Array entries are linked lists of row versions in descending timestamp order. In addition to a pointer to the next oldest version, a row version contains data, a status, and read and write timestamps. The latter three are used by Cicada's concurrency control protocol.

Cicada uses a variant of multi-version timestamp ordering [4, 33]. Each client thread maintains a local clock. (Cicada does not support networked clients.) The local clocks are loosely synchronized and individually return increasing values.

A client uses its clock to assign a unique timestamp to each transaction. As the transaction executes, each write creates a new row version, and the transaction's timestamp becomes the version's write timestamp. Further, reading a version updates its read timestamp to the max of the transaction's timestamp and the version's current read timestamp. Cicada uses the timestamps to check if a transaction can commit under serializability [50]. Ordering transactions by their timestamps yields a valid serial schedule.

Logging & Replication. Because Cicada does not support networking, logging, persistence, or replication [33], we emulate primary-backup replication on one server.

To this end, we implement a minimal prototype logger to allow replay of the primary's writes on the backup. The primary only writes logs to memory. After execution and validation but before committing, each client thread logs its changes to a per-thread log. The per-thread logs are coalesced into a single, totally ordered log before the backup's scheduler, workers, and snapshotter start.

The log is divided into fixed-size segments, each backed by a 2 MiB huge page [34]. Each segment's header indicates the number of log records it contains. For simplicity, the logger ensures transactions never span segment boundaries.

A client creates a log record for each write in a transaction. Each record contains the following: a table ID, a row ID, the write's timestamp, and a full copy of the row version. Further, the log contains two metadata fields to be used by the scheduler, as described below: First, each segment header contains a Boolean `preprocessed` flag. Second, each record contains a 64-bit `prev_timestamp` field.

Each log record is split into a header and data, with the former containing everything except the row version. Headers are written from the beginning of the segment, and data are written from the end. Co-locating all headers in a segment was critical for reducing the amount of data the scheduler (discussed further below) needed to process and prevented its throughput from being limited by memory bandwidth, especially in the face of concurrent workers.

7.2 Implementation

Scheduling & Execution. For simplicity, the description below assumes we are replicating one table. The implementation supports multiple tables using a queue for each table and row ID pair.

Because dynamically allocating per-row FIFO queues would prevent the single-threaded scheduler from matching Cicada's high throughput, it instead (logically) embeds the per-row FIFOs in the log. More specifically, it sets each log record's `prev_timestamp` to the timestamp of the write to the same row that precedes it. To do so, it maintains a map of row IDs to write timestamps, initially zero. Then to process a record, the scheduler first reads the value from the map using the log record's row ID, updates the record's `prev_timestamp`, and finally updates the map's value with the record's write timestamp. After the scheduler processes all of a segment's records, it sets its `preprocessed` flag to true.

Workers are assigned to log segments in a round robin fashion. Once the scheduler processes a segment, the assigned worker starts executing its writes, one for each log record, in three steps: First, it uses the record's `prev_timestamp` to see if the write is safe to execute. If `prev_timestamp` is equal to the write timestamp of the row version at the head of the storage engine's version list, then the write should be executed next. Otherwise it is deferred. Second, if the write is safe, the worker allocates a new row version and copies in the necessary data from the log. Third, the new version is installed at the head of row's version list. Each worker maintains a local FIFO of deferred writes and periodically (after each segment) re-checks them to see if they are now safe to execute.

Figure 6b illustrates the scheduler and workers' data structures as they execute three transactions of the type described in Section 4.1. The bold black line delimits the two log segments, and the scheduler has finished embedding the per-row queues in both. Workers one and two have started processing the left and right segments, respectively, and the former already executed a_1 and the latter b_1. Worker two has deferred executing b_2 since worker one is still executing a_2. Instead, worker two executes c_1, which is already safe to execute. As shown, C5-Cicada thus maintains a distributed, approximate version of the scheduler queue described in Section 4.1 comprising the log and each worker's deferred queue.

Snapshotter. Cicada's storage engine can efficiently implement the operations described in Section 4.2 because workers can explicitly assign timestamps to their writes. This allows workers to write to specific snapshots. Further, read-only transactions can execute against the current snapshot simply by using the sequence number c as a timestamp. Their reads will then reflect any previously executed writes with lesser timestamps. The storage engine thus logically contains the current, next, and future snapshots.

This simplifies C5-Cicada's snapshotter. To merge the current and next snapshots, it simply advances c to n, and replacing the next snapshot with the future snapshot and creating a new future snapshot occur implicitly when c and n advance.

Before advancing c to n, however, the snapshotter must guarantee all writes with timestamps less than or equal to n finish executing. To do this, it cooperates with the workers.

Worker i maintains a local variable c_i as one less than the timestamp of the write it most recently executed. Because log records are ordered by timestamp and each worker processes segments in

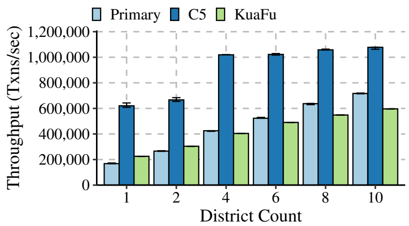


Figure 11: Primary and backup's throughput on 50%-50% NewOrder-Payment with increasing districts.

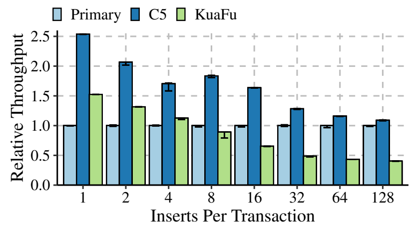


Figure 12: Primary and backup's throughput on adversarial workload as inserts per transaction increases.

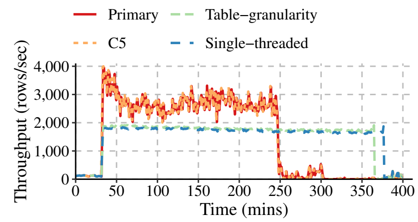


Figure 13: Lag at Meta used to reach 2 hours daily and take 2 hours to recover. With C5, it remains below 3 seconds.

log order, the worker can guarantee it will never execute another write with a timestamp less than or equal to c_i (assuming no writes are deferred). Each c_i is thus an upper bound on c. When a worker defers a write, it also defers updating c_i until the write executes.

The snapshotter's implementation is simple: in a separate thread, it periodically calculates a new n as the minimum across all c_i and then advances c to n. Since each c_i has only one reader and one writer, no coordination or atomic instructions are needed. With x86's total store order, the snapshotter may read a stale copy of a worker's c_i, but this does not violate correctness.

7.3 Evaluation

Our evaluation of C5-Cicada explores whether our design can keep up with advanced concurrency control protocols.

Experimental Setup. Experiments ran on the same machines as described in Section 6. For the primary, each thread is pinned to its own physical core, and for the backup, the same is true of the scheduler, workers, and snapshotter.

As mentioned in Section 7.2, our C5-Cicada implementation emulates primary-backup replication. As a result, unlike in our prior evaluation, the primary and backup ran consecutively on the same machine. We consider a cloned concurrency control protocol as keeping up if its throughput matches or exceeds the primary's.

Because logging slows Cicada, we compare against Cicada's performance without logging, which is an upper bound on that with logging enabled. We again use the optimal number of primary threads and backup workers, with the latter never exceeding the former. We ran experiments five times and report the median. Error bars show the minimum and maximum.

The workloads are the same as in Section 6, and we re-implement KuaFu in Cicada for fair comparison. In fact, we implement two versions of KuaFu, one optimized for low contention and the other for high contention. (The latter matches the published pseudocode [21].) We report whichever achieves better performance.

C5-Cicada Prevents Potential Disasters. Cicada's concurrency control is much better than MyRocks's at handling contention. Thus, we expect significantly more potential for replication lag.

Our results support this: C5-Cicada is necessary to prevent replication lag with the standard 50%-50% NewOrder-Payment workload after applying similar optimizations as in Section 6. Cicada achieves 716,950 transactions/s while KuaFu manages only 596,310 transactions/s, lagging by about 17%. C5-Cicada easily keeps up, committing 1,062,533 transactions/s. The results on the unoptimized workload are similar. The primary's throughput is lower, but KuaFu still lags by about 13,000 transactions/s (3%).

We also explore how the primary and two backups behave under varying levels of contention. Figure 11 compares the throughputs of Cicada, C5-Cicada, and KuaFu on a 50%-50% workload as we vary the number of districts from 10 (the standard setting) to 1. As contention increases (i.e., districts decrease), KuaFu lags until 4 districts, but below that, the additional contention harms Cicada's throughput more than KuaFu's by causing significantly higher abort rates (up to about 75%). Thus, with fewer districts, KuaFu keeps up.

To help confirm KuaFu lags due to the constraints imposed on its execution, we re-ran the experiment above but disabled its scheduler's calculation of transaction-granularity constraints. For each number of districts, we compared KuaFu's throughput while using the same number of workers as shown in Figure 11, and in all cases, KuaFu kept up. For example, with 10 districts and 6 workers, KuaFu's throughput nearly doubles from 596,310 to 1,101,491 transactions/s when its execution is unconstrained. This far exceeds the primary's throughput of 716,950 transactions/s.

These results highlight the complexity of predicting when existing cloned concurrency control protocols will be sufficient to avoid replication lag. As shown in Figure 11, C5-Cicada always keeps up and thus removes the potential for disasters.

C5-Cicada Always Keeps Up. We again validate C5-Cicada using the insert-only and adversarial workloads. On insert-only, Cicada achieves its best performance with transactions of 16 inserts each, amortizing its per-transaction overhead. Here, Cicada inserts about 87M rows/s with 20 threads. KuaFu (96M rows/s with 12 workers) and C5-Cicada (99M rows/s with 10 workers) both keep up. In each case, the scheduler provides sufficient performance.

Figure 12 compares each cloned concurrency control protocol's performance to Cicada's on the adversarial workload. We plot the backup's throughput relative to the primary's median as we vary the number of non-conflicting inserts per transaction.

C5-Cicada mirrors the primary and executes the non-conflicting inserts in parallel, so it always keeps up. This advantage is especially evident as the number of inserts per transaction increases from 4 to 8. With more parallel work per transaction, C5-Cicada leverages additional workers and its relative throughput actually increases.

On the other hand, the primary's advantage over KuaFu increases with the number of inserts per transaction. With 128 inserts per transaction, KuaFu's throughput is just 40% of the primary's.

8 DEPLOYMENT EXPERIENCE

MyRocks databases are deployed and used inside the globally distributed data centers at Meta. Each shard of their social graph uses asynchronous primary-backup replication. Further, their internal cloud provides asynchronously replicated, multi-tenant MyRocks instances. Before deploying C5-MyRocks, ensuring short replication lag throughout their infrastructure was a persistent challenge.

A version of C5-MyRocks has been deployed in production since mid-2017, and full deployment finished in early 2019. Deploying C5-MyRocks at Meta has notably improved the reliability of their applications built atop asynchronous primary-backup databases. Since its deployment, anecdotally, the number of complaints about replication lag by on-call engineers drastically decreased.

Figure 13 shows one example of significant lag fixed by C5-MyRocks. It plots throughput over time for one shard. During daily periods of high insert load, the primary's throughput exceeded the backup's, and lag grew to over two hours both with MySQL 5.6's default, single-threaded cloned concurrency control [11, 43] and Meta's earlier table-granularity protocol. After the load spike ends, both protocols took two hours to return lag to zero. C5-MyRocks eradicated the issue, keeping lag below three seconds.

Similarly, after live videos were deployed, popular videos became significant sources of contention. Like in the motivating example in Section 2.1, all comments on a video caused writes to a single row. With prior cloned concurrency control protocols, popular videos caused significant replication lag. Although this problem was initially fixed by batching comment-update requests, C5-MyRocks would have avoided the problem entirely.

Solving replication lag also had secondary benefits: First, deploying C5-MyRocks revealed bottlenecks in downstream systems [56], which have since been fixed by Meta's engineers. Second, Meta uses cross-region replication. Multiple copies of the data exist on servers within a primary region, and writes asynchronously replicate to backups in other regions. Short lag reduces the number of times that data must be fetched from other regions to satisfy read-your-writes consistency for clients with recent writes. Further, if the entire primary region fails while all machines in the backup region are lagging, some unreplicated writes may be lost. With C5-MyRocks, the probability of user data loss and the magnitude of such loss if it occurs are both significantly reduced. Finally, C5-MyRocks eliminated a noisy-neighbor problem experienced by applications deployed on Meta's internal cloud. Applications using a multi-tenant MyRocks instance previously would sometimes experience replication lag caused by others sharing the instance.

9 RELATED WORK

Deterministic Concurrency Control. Deterministic concurrency control protocols [13–15, 54, 59, 65] ensure that database state is a deterministic function of the input log. C5's processing of writes from the primary is inspired by such protocols. These include the up-front resolution of write-write conflicts prior to executing them [13] and its representation of permissible execution schedules of writes [15, 65]. Databases employing deterministic concurrency control, however, do not designate a single replica as a primary and others as backups. They instead employ active replication [58, 59]. C5, on the other hand, is applicable to primary-backup systems where the primary is non-deterministic.

Database Recovery & Replication. Database recovery [5, 31, 38, 55, 68] and replication [37, 39, 46, 53, 67] are two problems closely related to cloned concurrency control. In database recovery, changes to the database are logged and stored on stable storage. If a database fails, a recovery protocol creates a new copy of the database. Database replication extends database recovery to reduce recovery time by replicating and applying changes to the backup while the primary executes transactions. If the primary fails, the backup executes a synchronization protocol to bring it into a consistent state before processing new transactions. But database replication (and thus recovery) is simpler than cloned concurrency control because backups do not serve read-only transactions, so the backup only needs to be prefix-consistent before processing new transactions. A cloned concurrency control protocol must always be able to serve read-only transactions from a prefix-consistent state.

Cloned Concurrency Control. No existing asynchronous or semi-synchronous cloned concurrency control protocol can guarantee bounded replication lag. (Semi-synchronous protocols require a log of a transaction's writes to be persisted at the backup before the transaction commits at the primary.) Synchronous protocols [10] trivially guarantee it because the primary and backup coordinate before a transaction commits, but they reduce the primary's performance. Thus, asynchronous and semi-synchronous are more widely deployed [2, 23, 26, 35, 62].

Transaction- [21, 29, 40, 45] and page-granularity [2, 48, 62] protocols cannot guarantee bounded replication lag. By similar reasoning, coarser granularity protocols, such as those using groups of transactions [36, 41, 49], cannot either.

To the best of our knowledge, Query Fresh [63] is the only existing row-granularity cloned concurrency control protocol, but as discussed below, it does not guarantee bounded lag. It is a semi-synchronous protocol, and to reduce its workers' processing, instantiation of the backup's copy of the database is deferred to read-only transaction threads. These threads load row versions as necessary to return correct results (as defined by monotonic prefix consistency).

On one hand, Query Fresh's lazy instantiation can cause arbitrarily large replication lag by forcing read-only transaction threads to traverse large portions of the log. We provide a detailed description of this case in the appendix of our technical report [20]. On the other, lazy instantiation allows Query Fresh to avoid some of the work that C5 does by applying all writes. An interesting avenue of future work could explore a partially lazy approach.

10 CONCLUSION

We presented C5, the first cloned concurrency protocol to provide bounded replication lag. C5 comprises three parts: a scheduler, workers, and a snapshotter. C5 is backed by multiple theoretical results showing the necessity of its row-granularity protocol. We also presented two implementations: C5-MyRocks and C5-Cicada. The former is backward-compatible with MyRocks and deployed at Meta, while the latter faithfully implements our design. We demonstrated experimentally they always keep up with their primaries.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd for their feedback. We also thank Princeton's systems group for their comments on earlier drafts of this paper. This work was supported by the NSF under grants CNS-1824130 and IIS-1910613.

REFERENCES

[1] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. MIT, Cambridge, MA. Advisor(s) Barbara Liskov.

[2] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new SQL server in the cloud. In *Proc. ACM International Conference on Management of Data (SIGMOD).* Association for Computing Machinery, Amsterdam, The Netherlands, 1743–1756.

[3] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proc. ACM International Conference on Management of Data (SIGMOD).* Association for Computing Machinery, San Jose, CA, 1–10.

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA.

[5] Sang K. Cha and Changbin Song. 2004. P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload. In *Proc. International Conference on Very Large Data Bases (VLDB).* VLDB Endowment, Toronto, Canada, 1033–1044.

[6] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proc. ACM International Conference on Management of Data (SIGMOD).* Association for Computing Machinery, Houston, TX, 275–290.

[7] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. 1984. Implementation techniques for main memory database systems. In *Proc. ACM International Conference on Management of Data (SIGMOD).* Association for Computing Machinery, Boston, MA, 1–8.

[8] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proc. ACM International Conference on Management of Data (SIGMOD).* Association for Computing Machinery, New York, NY, 1243–1254.

[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proc. USENIX Annual Technical Conference (ATC).* USENIX Association, Renton, WA, 1–14.

[10] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. 2004. *Generalized Snapshot Isolation and a Prefix-consistent Implementation.* Technical Report IC/2004/21. School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland.

[11] Facebook. 2019. MyRocks GitHub Wiki. Retrieved February 8, 2019 from https://github.com/facebook/mysql-5.6/wiki

[12] Facebook. 2020. RocksDB: A Persistent Key-value Store. Retrieved April 13, 2020 from https://rocksdb.org/

[13] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. of the Very Large Data Bases Endowment (PVLDB)* 8, 11 (jul 2015), 1190–1201.

[14] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. of the Very Large Data Bases Endowment (PVLDB)* 10, 5 (Jan. 2017), 613–624.

[15] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. 2014. Lazy Evaluation of Transactions in Database Systems. In *Proc. ACM International Conference on Management of Data (SIGMOD).* Association for Computing Machinery, Snowbird, UT, 15–26.

[16] Dieter Gawlick and David Kinkade. 1985. Varieties of concurrency control in IMS/VS fast path. *IEEE Data Engineering Bulletin* 8, 2 (1985), 3–10.

[17] GitLab. 2017. GitLab.com Database Incident. Retrieved February 8, 2019 from https://about.gitlab.com/2017/02/01/gitlab-dot-com-database-incident/

[18] GitLab. 2017. Postmortem of Database Outage of January 31. Retrieved February 8, 2019 from https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/

[19] Google. 2020. LevelDB. Retrieved April 13, 2020 from https://github.com/google/leveldb

[20] Jeffrey Helt, Abhinav Sharma, Daniel J. Abadi, Wyatt Lloyd, and Jose M. Faleiro. 2022. C5: Cloned Concurrency Control that Always Keeps Up. https://doi.org/10.48550/arXiv.2207.02746

[21] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. 2013. KuaFu: Closing the parallelism gap in database replication. In *Proc. IEEE International Conference on Data Engineering (ICDE).* Institute of Electrical and Electronics Engineers, Brisbane, Australia, 1186–1195.

[22] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proc. of the Very Large Data Bases Endowment (PVLDB)* 13, 5 (jan 2020), 629–642.

[23] Instagram. 2015. Instagration Part 2: Scaling Our Infrastructure To Multiple Data Centers. Retrieved November 18, 2019 from https://instagram-engineering.com/instagration-pt-2-scaling-our-infrastructure-to-multiple-data-centers-5745cbad7834

[24] Intel. 2020. Intel Optane Persistent Memory. Retrieved September 17, 2020 from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[25] Intel. 2020. Intel Server Products. Retrieved April 12, 2020 from https://www.intel.com/content/www/us/en/products/servers.html

[26] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez. 2018. Harmonizing Speculative and Non-Speculative Execution in Architectures for Ordered Parallelism. In *Proc. IEEE/ACM International Symposium on Microarchitecture (MICRO).* Institute of Electrical and Electronics Engineers, Fukuoka City, Japan, 217–230.

[27] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *Proc. of the Very Large Data Bases Endowment (PVLDB)* 3, 1–2 (sep 2010), 681–692.

[28] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proc. ACM International Conference on Management of Data (SIGMOD).* Association for Computing Machinery, San Francisco, CA, 1675–1687.

[29] Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. 1991. Management of a Remote Backup Copy for Disaster Recovery. *ACM Transactions on Database Systems (TODS)* 16, 2 (May 1991), 338–368.

[30] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *VLDB Endowment* 5, 4 (dec 2011), 298–309.

[31] Juchang Lee, Kihong Kim, and Sang K. Cha. 2001. Differential Logging: A Commutative and Associative Logging Scheme for Highly Parallel Main Memory Databases. In *Proc. IEEE International Conference on Data Engineering (ICDE).* Institute of Electrical and Electronics Engineers, Heidelberg, Germany, 173–182.

[32] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *Proc. Conference on Innovative Data Systems Research (CIDR).* cidrdb.org, Asilomar, CA, 1–12.

[33] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proc. ACM International Conference on Management of Data (SIGMOD).* Association for Computing Machinery, Chicago, Illinois, 21–35.

[34] Linux. 2020. libhugetlbfs: preload library to back text, data, malloc() or shared memory with hugepages. Retrieved April 13, 2020 from https://linux.die.net/man/7/libhugetlbfs

[35] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proc. ACM Symposium on Operating Systems Principles (SOSP).* Association for Computing Machinery, Monterey, CA, 295–310.

[36] MariaDB. 2017. MariaDB 10 Parallel Replication. Retrieved April 13, 2020 from https://mariadb.com/kb/en/parallel-replication/

[37] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. 2013. RemusDB: Transparent High Availability for Database Systems. *The VLDB Journal* 22, 1 (Feb. 2013), 29–45.

[38] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (March 1992), 94–162.

[39] C. Mohan, K. Treiber, and R. Obermarck. 1993. Algorithms for the management of remote backup data bases for disaster recovery. In *Proc. IEEE International Conference on Data Engineering (ICDE).* Institute of Electrical and Electronics Engineers, Vienna, Austria, 511–518.

[40] MySQL. 2017. Improving The Parallel Applier With Writeset-based Dependency Tracking. Retrieved February 8, 2019 from https://mysqlhighavailability.com/improving-the-parallel-applier-with-writeset-based-dependency-tracking/

[41] MySQL. 2019. Group Commit of Binary Log. Retrieved February 8, 2019 from https://dev.mysql.com/worklog/task/?id=5223

[42] MySQL. 2019. InnoDB. Retrieved April 13, 2020 from https://dev.mysql.com/doc/refman/5.6/en/innodb-storage-engine.html

[43] MySQL. 2020. MySQL 5.6 Reference Manual. Retrieved April 13, 2020 from https://dev.mysql.com/doc/refman/5.6/en/

[44] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase Reconciliation for Contended In-Memory Transactions. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Broomfield, CO, 511–524.

[45] Oracle. 2001. Method of applying changes to a standby database system. Patent No. US6980988B1, Filed Oct. 1, 2001, Issued Dec. 27, 2005.

[46] Oracle. 2014. Eager replication of uncommitted transactions. Patent No. US9747356B2, Filed Jan. 23, 2014, Issued Aug. 29, 2017.

[47] Oracle. 2020. Oracle 19 Database Administrator's Guide. Retrieved January 7, 2020 from https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/index.html
[48] Oracle. 2020. Oracle Active Data Guard. Retrieved April 6, 2020 from https://www.oracle.com/technetwork/database/availability/dg-adg-technical-overview-wp-5347548.pdf
[49] Oracle. 2020. Understanding Oracle GoldenGate. Retrieved April 13, 2020 from https://docs.oracle.com/en/middleware/goldengate/core/19.1/understanding/getting-started-oracle-goldengate.html
[50] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)* 26, 4 (Oct. 1979), 631–653.
[51] PostgreSQL. 2020. PostgreSQL 12.1 Documentation. Retrieved April 13, 2020 from https://www.postgresql.org/docs/12/index.html
[52] PostgreSQL. 2020. Snapshot Synchronization Functions. Retrieved January 27, 2020 from https://www.postgresql.org/docs/current/functions-admin.html
[53] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2017. Scalable Replay-Based Replication for Fast Databases. *Proc. of the Very Large Data Bases Endowment (PVLDB)* 10, 13 (Sept. 2017), 2025–2036.
[54] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, Virtual Event, 180–194.
[55] David Schwalb, Martin Faust, Johannes Wust, Martin Grund, and Hasso Plattner. 2014. Efficient Transaction Processing for Hyrise in Mixed Workload Environments. In *Proc. International Workshop on In Memory Data Management and Analytics (IMDM)*. Springer, Hangzhou, China, 16–29.
[56] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. 2015. Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Oakland, CA, 351–366.
[57] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. 1994. Session guarantees for weakly consistent replicated data. In *Proc. International Conference on Parallel and Distributed Information Systems (PDIS)*. Institute of Electrical and Electronics Engineers, Austin, TX, 140–149.
[58] Alexander Thomson and Daniel J. Abadi. 2010. The Case for Determinism in Database Systems. *Proc. of the Very Large Data Bases Endowment (PVLDB)* 3, 1–2 (sep 2010), 70–80.
[59] Alexander Thomson, Thaddeus Diamond, Shu-Chen Weng, Kun Ren, Phillip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proc. ACM International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, Scottsdale, AZ, 1–12.
[60] TPC Council. 2010. TPC Benchmark C revision 5.11.
[61] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, Farmington, PA, 18–32.
[62] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *Proc. ACM International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, Houston, TX, 789–796.
[63] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *Proc. of the Very Large Data Bases Endowment (PVLDB)* 11, 4 (Dec. 2017), 406–419.
[64] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proc. of the Very Large Data Bases Endowment (PVLDB)* 10, 2 (oct 2016), 49–60.
[65] Arthur Whitney, Dennis Shasha, and Stevan Apter. 1997. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In *International Workshop on High Performance Transaction Systems*. Springer, Asimolar, CA, 211–217.
[66] Cong Yan and Alvin Cheung. 2016. Leveraging Lock Contention to Improve OLTP Application Performance. *Proc. of the Very Large Data Bases Endowment (PVLDB)* 9, 5 (jan 2016), 444–455.
[67] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proc. of the Very Large Data Bases Endowment (PVLDB)* 12, 11 (jul 2019), 1637–1650.
[68] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Broomfield, CO, 465–477.