Bonspiel: Low Tail Latency Transactions in Geo-Distributed Databases (Extended Version)

Fan Cui   Eric Lo   Srijan Srivastava   Ziliang Lai
The Chinese University of Hong Kong

ABSTRACT

Tail latency is crucial as it impacts user satisfaction and service-level objectives (SLOs). However, geo-distributed databases have long struggled with this issue due to wide-area network access, resulting in tail latencies of several or even exceeding ten seconds. In this paper, we highlight that further optimizing atomic commit protocols does not help but hit a tail latency wall. Instead, making concurrency control and access method selection geo-aware can mitigate this issue. To this end, we present Bonspiel, a new geo-distributed database equipped with geo-aware concurrency control and access method selection. In our experiments, Bonspiel successfully caps the tail latency of TPC-C at 1.8 seconds. Remarkably, it achieves this while maintaining full generality – it is fully SQL-compliant and strongly consistent, with both average latency and system throughput remaining at the top of the field.

PVLDB Reference Format:
Fan Cui, Eric Lo, Srijan Srivastava, and Ziliang Lai. Bonspiel: Low Tail Latency Transactions in Geo-Distributed Databases. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fancui-cuhk/Bonspiel.git.

1 INTRODUCTION

Large-scale web applications, ranging from social networks to e-commerce, are often powered by geo-distributed databases [8, 14, 17, 22, 56, 74, 77, 88, 96]. These databases feature high scalability and availability by *partitioning* data and *replicating* the partitions across multiple regions.

With data partitioned, Atomic Commit (AC) plays a vital role in multi-region (MR) transactions. Over the years, the majority of the efforts [29, 34, 38, 53, 55, 61, 62, 65, 70, 73, 77, 85, 87, 88, 93, 94, 96, 97] have focused on minimizing the atomic commit latency. Figure 1 illustrates the improvement in average latency across four geo-distributed databases developed at different times. These are the **general** ones in the sense that they do not impose any requirement on workload (e.g., can support branching in transactions). As shown in the figure, there have been consistent improvements, *provided that* we follow *their* experimental setting, reporting (1) **average latency** using (2) **YCSB-A** with (3) **100% multi-region (MR) transactions** on (4) **low contention** ($\theta = 0.2$).

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

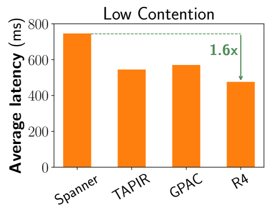

Figure 1: Average latency of four general geo-distributed databases. YCSB-A; 100% MR transactions; Low Contention (θ=0.2). 5 replicas on 5 regions. Detailed setup in Section 4.

Under this setting, we can see that starting with the pioneer, Spanner [22], introduced by Google in 2012, followed by TAPIR [93] (2015), GPAC [55] (2019), till most recently R4 [41] (2024), the average latency of geo-distributed databases has improved by 1.6×. The latency improvements are primarily attributed to the optimizations in their replicated atomic commit protocols, where the corresponding *commit time* has been reduced from 3 wide-area network (WAN) round-trip times (RTT) in Spanner, to 1–2 RTTs in TAPIR and GPAC, and more recently down to 1–1.5 RTTs in R4.

1.1 Wait! What About Tail Latency?

Optimizing the atomic commit protocol can unquestionably reduce the *atomic commit latency* of MR transactions. However, end-to-end *tail latency* is crucial for user-facing applications, and its significance becomes even more critical when a geo-distributed database is offered as a cloud service with service-level objectives (SLOs) [11, 18, 19, 26, 37, 48, 64, 68, 69, 76, 89].

Figure 2 shows the performance of the relevant systems when reporting their **99.9% (p999) tail latency** on both low contention ($\theta = 0.2$) and high contention ($\theta = 0.8$) scenarios. It is apparent that the *optimization of atomic commit in these systems does not translate to improved tail latency.*

1.2 What about TPC-C?

An industrial-strength geo-distributed database should not excel in only one type of workload. Unfortunately, most studies on general geo-distributed databases have been evaluated only on YCSB and some microbenchmarks (e.g., [29, 41, 53, 77, 85, 87, 93]). In light of this, we changed the workload to TPC-C and studied the tail latency, as shown in Figure 3.

TPC-C simulates a **realistic** workload where there are **only 10–15% MR transactions**, with the remainder being single-region (SR) transactions. Under TPC-C, we find that *systems with optimized atomic commit do not have better tail latency* than Spanner either; in fact, they often have worse, even though tail latency is primarily influenced by longer-running MR transactions, and enhancing atomic commit should reduce the number of RTTs in MR transactions.

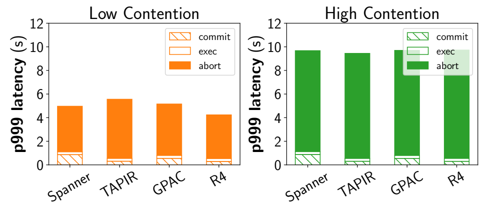

Figure 2: Tail latency (with breakdown). YCSB-A; 100% MR transactions.

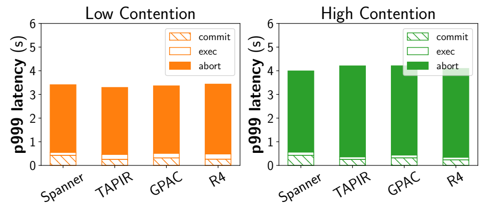

Figure 3: Tail latency (with breakdown). TPC-C.

1.3 Where Does the Time Go?

To provide insights into the factors contributing to tail latency (primarily attributed to slower MR transactions), Figures 2 and 3 include a detailed time breakdown. Specifically, due to contention, a transaction may abort and re-execute multiple times before it can successfully commit. The breakdown shows (1) the `execution` time and (2) the `commit` time of the final successful round of a transaction, as well as (3) the `abort` time, which reflects the time wasted on failed attempts.

The breakdown reveals that optimizing the atomic commit protocol (from Spanner to R4) *does* reduce commit time; however, this reduction is secondary, especially when considering the abort time spent on failed attempts. This issue is even more pronounced under high contention scenarios, which are commonly found in many real-world workloads [15, 19, 86, 89]. From the breakdown, we assert that, to effectively optimize tail latency, the **primary focus should be on reducing the abort time of MR transactions**.

The abort time of an MR transaction is determined by: (1) its *abort rate*, quantified by the number of aborts and retries before it can successfully commit (Figures 2 and 3), multiplied by (2) its *abort penalty*, measured by the time spent on a failed round.

For (1), the high abort rate of MR transactions is primarily caused by contention with SR transactions, as the latter dominate in most realistic workloads [24, 34, 62, 66, 67, 70, 72]. In geo-distributed databases, SR transactions are **not** lightweight because they also need to access the WAN during logging. This results in them holding locks for an unconventionally extended period of time, causing a high number of aborts even under low contention [41, 42, 52, 79]. This issue arises regardless of the concurrency control protocol used (e.g., 2PL [12] or OCC [40]).

Most deterministic concurrency control (DCC) protocols [29, 34, 61, 62, 70, 79] can avoid transaction aborts. However, these protocols require prior knowledge of the transactions' read-write sets, limiting the customer base from a cloud service standpoint.

For (2), the high abort penalty of MR transactions is primarily attributed to the access method employed by most systems. Specifically, many geo-distributed databases (e.g., Spanner and R4) always read records from the leader of the replication group [56]. While this *always* "`read-leader`" access method ensures the transaction reads fresh records, it incurs a high abort penalty if the leader is far away and the transaction eventually aborts. TAPIR does not suffer from this problem by choosing to always read from the nearest replica of the replication group. However, its *always* "`read-nearest`" access method exposes TAPIR to a higher abort rate when records are frequently updated, as the data retrieved from the nearest replica may not be as fresh as that obtained from the leader.

1.4 Bonspiel

In this paper, we introduce Bonspiel, a novel geo-distributed database system that significantly reduces tail latency while maintaining top performance in system throughput and average latency. Bonspiel employs two key techniques: (1) a *geo-distributed concurrency control* (GDCC) protocol designed to reduce the abort rate of MR transactions with minimal impact on SR transactions, and (2) *geo-aware access method selection* (GAMS) to minimize the abort penalty for MR transactions, without negatively affecting the abort rate.

Since MR transactions dominate tail latency, and their latency is heavily interfered by the predominance of SR transactions in realistic workloads, GDCC prioritizes MR transactions over SR transactions to prevent MR transactions from being easily aborted by SR transactions. Existing concurrency control protocols that support transaction prioritization are typically lock-based and tightly integrated with classical deadlock prevention schemes such as WOUND-WAIT or WAIT-DIE [2–4, 71]. However, they become ineffective in a geo-distributed setting. For example, in WOUND-WAIT, high-priority transactions are never blocked by low-priority transactions. When a high-priority transaction (H) attempts to access a record locked by a low-priority transaction (L), H simply aborts L. In geo-distributed databases, if we consider MR transactions to have higher priority than SR transactions, this approach can indeed reduce the abort rate at the cost of *increasing the abort rate of SR transactions*. However, as discussed, SR transactions are not lightweight in a geo-distributed setting, as they also require WAN access during logging. Hence, WOUND-WAIT would severely jeopardize system throughput and average latency.

In response to this issue, the concurrency control in Bonspiel, GDCC, is designed specifically for the geo-distributed context, eliminating all the aborts of MR transactions caused by conflicting SR transactions without jeopardizing the abort rate of SR transactions. GDCC is based on OCC, which generally admits higher concurrency than lock-based approaches [19, 54, 80, 82, 89, 91]. Unlike Polaris [89], a recent OCC designed for the single-node setting that supports priority, GDCC guarantees that MR transactions are both *abort-free* and *wait-minimal* with respect to SR transactions; i.e., an MR transaction is never aborted due to any SR transaction, and almost never need to wait for an SR transaction to finish its WAN logging. Furthermore, GDCC also guarantees SR transactions are always *wound-free*, i.e., once an SR transaction has successfully validated its read and write sets, it would never be wounded (i.e., aborted) by others. This wound-free property allows Bonspiel to achieve low tail latency with high system throughput.

Bonspiel has GAMS, which enables reading different records with different access methods within the same transaction. An MR transaction can therefore choose the better option between `read-leader` and `read-nearest` for each record based on factors

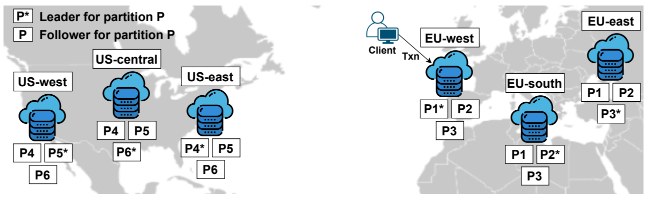

Figure 4: An example geo-distributed database. A data partition P is replicated to several data centers. P^* stands for the the leader replica.

such as record's update frequency and network latency, and selecting the one that would ultimately minimize abort rate and abort penalty. Overall, Bonspiel achieves a tail latency improvement of up to 2.2× on TPC-C and YCSB, while maintaining top-tier average latency and throughput compared to state-of-the-art systems.

2 PRELIMINARY

Figure 4 illustrates a typical setup of a geo-distributed database, where data is *partitioned* by regions or countries. Each data partition and its log are *replicated* to data centers in different regions for fault tolerance and high availability, using a State Machine Replication (SMR) protocol with strong consistency. Partitions can be further sub-partitioned for scalability, and a partition P can be either *fully replicated* across all data centers or *partially replicated* in some but not all data centers. A transaction is successfully *committed* after its log is replicated and becomes durable.

Geo-distributed databases can be categorized into *leader-based* and *leaderless* systems, based on the SMR protocols used. Leader-based geo-distributed databases [14, 22, 41, 55, 77, 85, 87, 88] utilize leader-based SMR protocols (e.g., Multi-Paxos [45], Raft [63]) for logging, where a designated replica serves as the *leader* to replicate transaction logs to other replicas known as *followers*. In such systems, the leader always has the most up-to-date data. By default, the leader is located in the region where its data originates. For example, the partition $P1$ in Figure 4 contains data for `EU-west` clients and has three replicas. The leader replica, $P1^*$, is set to be located in the `EU-west` data center, ensuring that most transactions from that region benefit from fast local accesses [34, 62, 70]. If the leader fails, a new leader is elected to ensure continuous service. In a system with $2f + 1$ replicas aiming to tolerate f simultaneous server failures, the replication quorum size is $f + 1$, meaning that a log is durable if replicated to $f + 1$ replicas. All industrial systems are leader-based [14, 22, 77, 88].

Leaderless geo-distributed databases [38, 61, 93] utilize leaderless SMR protocols [10, 27, 46, 47, 59], aiming to reduce latency through *fast paths* [47], which allows any replica to initiate replication of transaction logs, bypassing the leader and saving WAN round trips. However, fast paths require a larger quorum size of $\lceil 3/2f \rceil + 1$ for fault tolerance. Furthermore, if a transaction fails to gather enough votes to form a quorum during the fast path, it resorts to a classic leader-based *slow path*.

In leader-based systems, transactions typically exhibit low latency variance due to the smaller quorum size, requiring fewer replicas to respond. In contrast, leaderless systems may experience high latency variance, particularly under high contention, as transactions need to wait for more replicas to respond and many fast path failures can occur. We choose to develop Bonspiel as a leader-based system for the above reasons, while including leaderless systems in the evaluation section. Discussions in the following assume a leader-based system setting.

In geo-distributed databases, clients submit transactions to the nearest data center for processing. Transactions can be classified into two types: single-region (SR) transactions, which access data from partitions with leaders located in a single region, and multi-region (MR) transactions, which access data from partitions with leaders distributed across multiple regions.

2.1 Single-Region (SR) Transactions

SR transactions are executed and committed in the data center that hosts the *partition leaders*, i.e., the leader replicas of the required partitions (e.g., $P1^*$ in Figure 4). In systems using OCC protocols, the lifecycle of an SR transaction consists of three phases: the *execution phase*, the *validation phase*, and the *commit phase* [40, 80, 91].

An SR transaction starts the execution phase by reading the required records and their associated timestamps into a private read set and writing the updates into a private write set. Once all operations are completed, the transaction enters the validation phase, where it performs a validation to determine whether to commit or abort.

The validation process checks for serializability conflicts with concurrent transactions, aborting the transaction if any conflicts are detected. Specifically, during validation, the transaction first attempts to lock all records in its write set [40, 80, 91]. If all locks in the write set are successfully acquired, it then checks whether any record in its read set is locked or has been updated by others via cross-checking the records' timestamps. If validation fails, the transaction is aborted by releasing all acquired locks and discarding its read/write sets. Otherwise, the validation succeeds, the transaction enters the commit phase and logs its write set to all replicas using an SMR protocol across the WAN. After successful log replication, the partition leader installs the write set locally, releases the locks on the write set, and acknowledges the client and the other replicas that the transaction has been committed.

Previous studies [19, 54, 80, 82, 89, 91] indicate that OCC generally admits higher concurrency than lock-based concurrency control. Therefore, GDCC, Bonspiel's concurrency control protocol, is based on OCC, and we focus on OCC in the rest of the discussion. Regardless of the specific concurrency control protocol used, it is important to note that in a geo-distributed setting, **SR transactions require logging across the WAN**. This log replication results in a significantly longer lock-holding time (a.k.a. contention footprint) [41, 42, 52, 79] of 1 WAN RTT and longer latency (in *ms*) compared to a non-geo-distributed setting (latency in μs).

2.2 Multi-Region (MR) Transactions

MR transactions access data from partitions whose leaders reside in multiple regions. The data center that receives the transaction from the client serves as the transaction *coordinator* (e.g., `EU-west` in Figure 4). The lifecycle of an MR transaction also consists of an execution phase, a validation phase, and a commit phase.

The execution phase of an MR transaction involves reading data from *participants* that hosts the required data partitions (e.g.,

Read Access	Latency	Data Freshness
`read-leader`	Depends*	Freshest
`read-nearest`	Lowest	Depends*

Table 1: Properties of access methods. *Depends on whether the partition leader is co-located with the coordinator.

`US-east` and `EU-south` for a cross-country transaction in Figure 4). For most geo-distributed databases (e.g., Spanner, R4), when a transaction performs read operations, they adopt a `read-leader` approach that always reads a partition from its leader replica. If the leader replica of the partition happens to co-locate with the transaction coordinator (e.g., $P1^*$ in `EU-west`), then it is a local read, incurring no WAN RTT. Otherwise, the read operation is a remote read (e.g., $P2^*$ in `EU-south`), requiring one WAN RTT.

TAPIR uses a `read-nearest` approach instead. Specifically, transactions always read from the data center nearest to the coordinator that holds the required partition. When the nearest data center happens to host the leader replica of a partition (e.g., $P1^*$ in `EU-west`), the reads are both fast and fresh. However, when the nearest data center only hosts a follower replica of the partition (e.g., $P3$ in `EU-west`), these reads may return fresh or stale data, depending on whether the required data item is being updated at the leader ($P3^*$ in `EU-east`) while the follower is being read. Table 1 summarizes the discussion above.

After the execution phase, an MR transaction steps into the validation phase, using an atomic commit (AC) protocol to coordinate an atomic commit-vs-abort decision among all participants. In a leader-based system, coordination messages are sent to the leader replica only and the leader will replicate those to their respective followers accordingly.

Upon receiving any initial AC messages (e.g., `2PC-prepare` in 2PC [58]), every participant performs OCC validation on their respective partitions. Similar to an SR transaction, this involves locking their responsible write sets, validating the read sets, and logging their decisions using an SMR protocol across the WAN. After gathering the individual decisions from all participants, the coordinator formulates a final decision. Subsequently, the transaction enters the commit phase, during which the coordinator logs the final decision across the WAN, notifies the participants to either install the write set (or discard it if aborted), release the locks, and inform the client of the final decision.

The SOTA AC protocol, R4 [41], has made many steps of the SMR logging and the AC process to run in parallel, effectively reducing the atomic commit latency and the contention footprint of MR transactions in a geo-replicated environment from 3 WAN RTTs in Spanner to just 1 WAN RTT for 3 replicas (1.5 RTT for more than 3 replicas). In any case, **in geo-distributed databases, both SR and MR transactions require WAN accesses, leading to latencies that are in the same order of magnitude (both in *ms*)**.

3 BONSPIEL

Bonspiel is a geo-distributed database that delivers low tail latency transactions across geo-distributed availability zones. It supports both full and partial replication, accommodating general workloads without any specific assumptions, including those with non-deterministic transaction branches and `DATE` functions. Bonspiel excels in handling workloads with any realistic mix of single-region (SR) and multi-region (MR) transactions. It consistently provides low tail latency across varying levels of contention while maintaining top-tier performance in average latency and system throughput.

Bonspiel adopts leader-based replication using Raft [63], and employs R4 [41] as the atomic commit protocol with replication. In the following subsections, we introduce the two key components of Bonspiel: (1) *geo-distributed concurrency control (GDCC)*, which prioritizes MR transactions over SR transactions to minimize the abort rate of MR transactions (Section 3.1), and (2) *geo-aware access method selection (GAMS)*, which dynamically chooses different access methods for different records to reduce the abort penalty for MR transactions (Section 3.2).

3.1 Geo-Distributed Concurrency Control

The concurrency control protocol of Bonspiel is built on optimistic concurrency control (OCC) [40] and is specifically optimized for geo-distributed environments. In this context, both multi-region (MR) and single-region (SR) transactions are costly, as the system's high availability necessitates wide-area network (WAN) access for logging. This leads to the following design principles of Bonspiel's Geo-Distributed Concurrency Control (GDCC) protocol:

- **MR Abort-Free** (with respect to SR): MR transactions should be *abort-free* with respect to SR transactions. MR transactions, as the key determinant of tail latency, should not be aborted by any SR transaction, especially considering the prevalence of the latter in real workloads. Of course, an MR transaction can still be aborted if there are serializability conflicts with other concurrent MR transactions. For brevity, in what follows, we omit the phrase "with respect to SR transactions" when the context is clear.
- **SR Wound-Free**: The Abort-Free nature of MR transactions could easily result in solutions that prioritize MR transactions over SR transactions by *wounding* (i.e., aborting) SR transactions in the event of any conflict. However, since SR transactions are both costly and prevalent, *unconditionally* wounding them would severely jeopardize overall system throughput and average latency. Therefore, SR transactions should remain *wound-free* once they have successfully validated their read and write sets.
- **MR Wait-Minimal** (with respect to SR): To reduce the latency of MR transactions, it is essential not only to ensure they are abort-free but also to minimize their wait on SR transactions. However, completely eliminating the need for MR transactions to wait on SR transactions is practically infeasible, as MR Abort-Free and SR Wound-Free literally leave *waiting* as the only option on the table. Therefore, the guiding principle is to make the majority of an MR transaction's critical path wait-free. If waiting is absolutely necessary, we must minimize the wait time of MR transactions to reduce their tail latencies as much as possible. We will also omit the phrase "with respect to SR" when the context is clear.

Guided by these principles, Bonspiel's Geo-Distributed Concurrency Control (GDCC) is structured as an OCC-based protocol that prioritizes MR transactions over SR transactions. Within GDCC, SR transactions are assigned the lowest priority and MR transactions have varying levels of higher priority. The challenge of GDCC lies in how to achieve all three principles simultaneously.

To support priority under OCC, GDCC adopts the *reservation* concept from Polaris [89]. Reservation provides a priority primitive in OCC. During execution, an MR transaction reserves required records before accessing them, indicating its intent to use those records with priority. SR transactions cannot make reservations. Transactions will check for the following types of conflicts during execution and validation to ensure that lower-priority transactions cannot update records reserved by higher-priority transactions until the latter complete (commit or abort):

(1) Reserve-lock conflict: One transaction requests a reservation on item x while another holds a lock on x.
(2) Reserve-reserve conflict: One transaction requests a reservation on x while another has already reserved x.
(3) Lock-lock conflict: The typical OCC conflict, where one transaction requests a lock on x during validation, but x is already locked.
(4) Lock-reserve conflict: One transaction enters validation and requests a lock on x, but x is already reserved.

When an SR transaction enters the validation phase, it will abort if it detects that some records in its write set are reserved by other transactions. A tricky situation arises when (a) an SR transaction S has locked all records in its write set, successfully validated, but has not released the locks because logging is still in process, and (b) an MR transaction M arrives and attempts to reserve some of those locked records during its execution phase. In this case, we encounter the following issues:

(I1) MR Abort-Free: cannot abort M;
(I2) SR Wound-Free: cannot wound S;
(I3) MR Wait-Minimal: not wanting M to wait for S to release its locks because S's WAN logging is slow.

Note that this is what sets GDCC apart from Polaris, as Polaris was not designed for geo-distributed databases and would choose to wait in I3, leading to poor tail latency in a geo-distributed environment. In Bonspiel, we borrow the NO-WAIT policy from traditional locking schemes [71] and apply it in a novel way to MR transactions during their OCC-style execution phase. NO-WAIT refers to a self-abort (DIE) in locking schemes [71]. In OCC-based Bonspiel, we interpret NO-WAIT in a novel way as a "proceed": M does not wait for S to finish its logging, but proceeds to read the items locked by S and continues to place reservations on all required items. As a result, under Bonspiel, MR transactions genuinely never wait for SR transactions during their execution phase.

When comparing Bonspiel to Polaris, two challenges emerge with the introduction of the new NO-WAIT option. First, while NO-WAIT can make M wait-free during its execution phase, it may lead to M being aborted by S during its validation phase, thereby violating the MR Abort-Free principle. Specifically, by not waiting for S to finish its updates and continuing to read items locked by S, M's reads are *stale*, causing it to abort during validation. Second, although NO-WAIT can render M wait-free in its *execution phase*, it does not reduce the wait time for M during its *validation phase.*

To address the first challenge, Bonspiel allows dirty writes made by SR transactions to be *early visible* [28, 30, 31] to MR transactions once an SR transaction has successfully validated its read/write sets and before it completes its WAN logging. Since locks remain

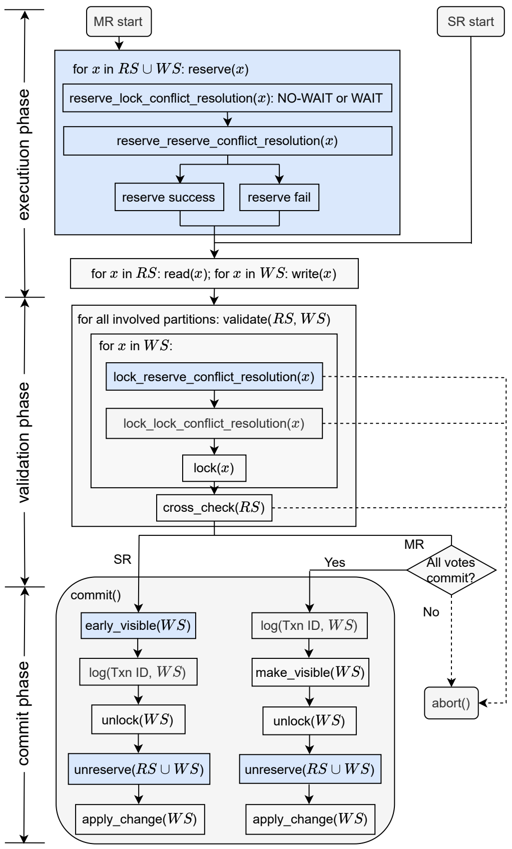

Figure 5: Transaction Lifecycle in Bonspiel.

held until SR's logging is finished in GDCC, this early-visible-write approach ensures serializability and recoverability.

Regarding the second challenge, consider a scenario where (a) SR has locked all records in its write set during validation but has not released the locks, as in the previous case (a), but now (b') the MR transaction M has used NO-WAIT to complete its execution phase and is entering its validation phase, initiating lock acquisition for its own write set. In this situation, we encounter the same issues I1 and I2 as before, but with the difference that now NO-WAIT (proceed) is not an option for issue I3 since M is now in its validation phase, it has to wait for S to release locks for correctness.

To mitigate this, GDCC is meticulously designed to ensure that any unavoidable wait either is on the *non*-critical path of the MR transaction's validation phase (thus not adding to the physical latency of MR) or that the wait time for M to acquire the lock released by S is minimized. The latter is accomplished by carefully overlapping the execution phase of M with the logging process of S, so that by the time the lock acquisition request from M reaches the leader of the required record, the logging of S on the same record is often completed with the locks released. Consequently, the lock acquisition request from MR is effectively wait-free in most circumstances.

3.1.1 Transaction Lifecycle. Figure 5 illustrates the lifecycle of a transaction in Bonspiel, highlighting GDCC's distinctions from standard OCC of geo-distributed databases in blue. During the execution phase, SR transactions execute as in standard OCC, while

MR transactions utilize a `reserve` primitive to reserve records in their read and write sets before accessing them. When an MR transaction attempts to reserve a record x, it may find that x is locked or reserved by another transaction. In such cases, the MR transaction follows a *Reserve-Lock Conflict Resolution* scheme and a *Reserve-Reserve Conflict Resolution* scheme to determine the outcome of the reservation (Section 3.1.2).

After the execution phase, a transaction enters its validation phase (`validate`), validating its read and write sets on the leader of every involved partition. The validation phase is mostly traditional: first attempts to lock all the records in the write set (`lock`), and if all locks are successfully acquired, the transaction then cross-checks the timestamps of the records in the read set against the latest timestamps of those records (`cross_check`). The only distinction here is that when attempting to lock all the records in the write set, the transaction employs a *Lock-Reserve Conflict Resolution* scheme on top of a *Lock-Lock Conflict Resolution* scheme to resolve the conflicts in case the required records are already locked or reserved by concurrent transactions (Section 3.1.3).

Based on the outcomes of Lock-Lock and Lock-Reserve Conflict Resolutions, along with the `cross_check` result, a transaction may either abort or pass the validation and proceed to commit. Note that the MR Abort-Free property in GDCC only ensures that an MR transaction is not aborted by an SR transaction, but does not prevent aborts arising from serializability conflicts between MR transactions.

GDCC admits different commit paths for SR and MR transactions. For MR transactions, they follow a mostly traditional `commit` procedure, which first logs the write sets, then makes the writes visible to other transactions, releases the locks on the write set, removes the reservations on the read and write sets by sending the write sets and unlock/unreserve requests to the respective leader of each record, and finally applies the changes to the durable storage (i.e., a majority of replicas in geo-distributed settings). For SR transactions, they make their write sets early visible (`early_visible`) once after successful validation. Then, they log their write sets, release the locks and reservations, and applies the changes as usual. Write set logging is managed by Raft. Therefore, any logging failures are transparently handled by the Raft protocol (e.g., leader re-election), requiring no additional mechanisms.

3.1.2 *Execution phase: Reservation.*

In GDCC, only MR transactions can do reservations. When reserving a record, an MR transaction first checks whether its intended reservation conflicts with any existing locks on the required records and follows a Reserve-Lock Conflict Resolution scheme to resolve the conflict. Next, it checks whether its intended reservation conflicts with any existing reservations on the required records and follows a Reserve-Reserve Conflict Resolution scheme to determine the reservation result (SUCCESS or FAILURE). It is important to note that a reservation failure for an MR transaction, at worst, represents only a priority inversion, which does not compromise correctness. Consequently, in Bonspiel, a transaction will never be aborted in the execution phase despite the reservation outcome.

Reserve-Lock Conflict Resolution. Table 2 shows the resolution scheme of Bonspiel in case an MR transaction M tries to reserve a record x but the record x is already locked by another SR transaction S or by another MR transaction M'. In simple terms, the Reserve-Lock Conflict Resolution of Bonspiel is *conditional-waiting*: the MR transaction proceeds immediately (NO-WAIT) if the locker holder is an *SR* transaction, but it waits (WAIT) if the locker holder is an *MR* transaction. Of course, if x is not locked, M always proceeds.

Table 2: Execution Phase: Reserve-Lock Conflict Resolution.

M `reserve`(x)	WOUND-WAIT	WAIT-DIE	Polaris	**GDCC**
x is locked by S	Abort S (*WOUND*)	*WAIT* for lock release	*WAIT* for lock release	**Proceed (*NO-WAIT*)**
x is locked by M'	*WAIT* for lock release	Self-abort (*DIE*)	*WAIT* for lock release	***WAIT* for lock release**

Table 3: Exec. Phase: Reserve-Reserve Conflict Resolution

M `reserve`(x)	**GDCC**
x is reserved by S	–
x is reserved by M'	Reserve SUCCESS

The rationale for M choosing NO-WAIT when the locker holder is an SR transaction is clear — for the MR Wait-Minimal principle. The rationale for M choosing WAIT when the locker holder is also an MR transaction M' is that both M and M' hold equal importance. Since the locker holder M' has already acquired the lock – meaning that M' has finished remote data access and is closer to lock release point, it is more beneficial for M to wait for M'.

In fact, there are always other options for M. For example, we can borrow traditional deadlock prevention schemes like WOUND-WAIT and WAIT-DIE from lock-based concurrency control [2–4, 71] and apply them here (Table 2). As discussed in the introduction, applying WOUND-WAIT here would mean wounding the lock holder SR and hurting the overall system throughput and average latency. Applying WAIT-DIE in this context means that if the lock holder is an SR transaction S, M would begin a long wait for S's WAN logging, prolonging its latency and abort penalty. Conversely, if the lock holder is another MR transaction M', M might need to abort itself, increasing its abort rate. The problem of Polaris also becomes clearer here. In Polaris, the MR transaction M that seeks to reserve will WAIT until the lock holder releases the lock, regardless of whether the locker holder is an SR or an MR transaction. This approach exposes M to the same long-waiting issue as WAIT-DIE, since the lock holder is often an SR transaction in real workloads.

Reserve-Reserve Conflict Resolution. After conditional waiting, the MR transaction M will next proceed to resolve the potential conflicts between its intended reservation and the existing reservations made by other MR transactions M'.

Table 3 outlines the resolution scheme of Bonspiel in this scenario. First, SR transactions cannot make reservations, so there is nothing to resolve between M and SR transactions (–). Second, if there are existing reservations on a record x made by other MR transactions M', Bonspiel still considers M's reservation as SUCCESS by allowing multiple reservations on x. The rationale is that a reservation is merely a *performance* primitive, prioritizing MR transactions over SR transactions under the lens of tail latency. Any violation of the reservation would not cause correctness issues. Since the current reservation holder M' may later fail to validate and be aborted, allowing M's reservation to succeed causes no harm but provides M with a higher chance to commit. Real serializability conflicts between M and M' would be resolved later when they reach their validation phase. Of course, when there is no reservation on x, the reservation must succeed.

In summary, the *basic* Reserve-Reserve Conflict Resolution scheme of Bonspiel is ALWAYS SUCCEED. It is considered basic because we will later introduce different priorities among MR transactions (Section 3.1.5) as an optimization, where reservations may FAIL.

3.1.3 Validation Phase. After the execution phase, a transaction enters the validation phase, begins with acquiring the locks for its write set. When no transaction has reserved or locked a record x in its write set, `lock(`x`)` always returns a SUCCESS.

Table 4: Validation Phase: Lock-Reserve Conflict Resolution.

Lock requester	Reservation owner	GDCC	Lock requester	Reservation owner	GDCC
S	S'	–	M	S	–
	M	Self-abort (DIE)		M'	Proceed (NO-WAIT)

Lock-Reserve Conflict Resolution. When a transaction tries to acquire the lock on record x in its write set, but another transaction has already reserved x, Bonspiel follows Table 4 to resolve such lock-reserve conflicts.

First, x can never be reserved by SR transactions (–) because they cannot make reservations. Second, when the transaction that seeks to lock is an SR transaction S and the record of interest x is already reserved by another MR transaction M, S must be aborted (DIE) to yield the commit opportunity to M, in order to comply with the MR Abort-Free principle. Third, when the lock requester is an MR transaction M and the record of interest x is already reserved by another MR transaction M', there is no issue because M' has only reserved x but has not locked it. In other words, M can PROCEED to validate whether it has any lock-lock conflict with other transactions.

Lock-Lock Conflict Resolution. When a transaction T attempts to acquire the lock on x but there is another transaction T' holding the lock, Bonspiel follows the standard OCC [40] to resolve the lock-lock conflict[1] – the lock requester T ALWAYS WAIT until the lock holder T' releases the lock. This applies regardless of the type and priority of the lock requester T and the lock holder T'.

However, one might wonder whether this ALWAYS WAIT policy, while standard (and hence correct), conflicts with the MR Wait-Minimal principle. In the following, we explain why it does not, due to the meticulous design of GDCC.

First, there are four types of lock-lock conflicts between the lock requester T and the lock holder T':

(1) (S-S conflict): both T and T' are SR transactions;
(2) (S-M conflict): Requester T is an SR transaction and lock holder T' is an MR transaction;
(3) (M-M conflict): both T and T' are MR transactions;
(4) (M-S conflict): Requester T is an MR transaction and lock holder T' is an SR transaction.

Among these, only type (4) is relevant to the MR Wait-Minimal principle. Therefore, we focus on explaining why M seldom waits for S to release the lock. We point out that:

[1]In standard OCC implementation, the write sets are sorted to avoid deadlocks [80, 91].

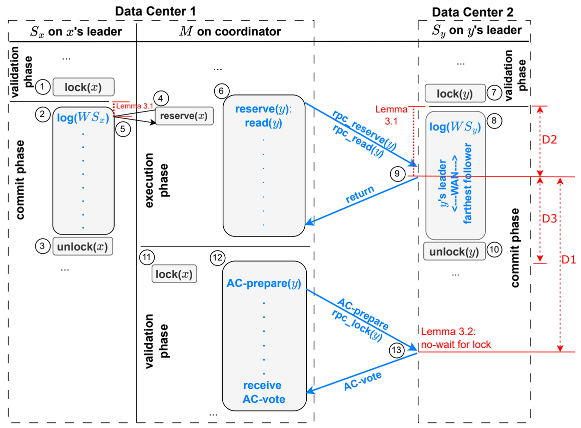

Figure 6: Schedule of an MR transaction M and two SR transactions S_x, S_y. Operations requiring WAN access are in blue.

Lemma 3.1. *When an MR transaction M attempts to lock a record but the record has already been locked by an SR transaction S, it indicates that, on the leader of the record, S must have locked the record before M reserves it.*

Proof. When M attempts to lock a record, the record must have been reserved by M (or some higher-priority MR transactions if the advanced optimization in Section 3.1.5 is enabled). According to GDCC's Lock-Reserve Conflict Resolution scheme (Table 4), if M has reserved the record and S subsequently tries to lock, S has to DIE. Now, S is still alive and has locked the record, indicating that M reserved the record after S had already locked it. $\square$

Now, focusing on M-S conflict (type 4), we can show that:

Lemma 3.2. *For an MR transaction M and a record y locked by an SR transaction S, if the RTT from the leader of y to the farthest follower in its nearest quorum is shorter than the RTT from the leader of y to the coordinator of M, then when M attempts to lock y, S will have already released the lock on y and M will require no waiting.*

Proof. Consider an MR transaction M attempting to lock two records x and y but they are already locked by two SR transactions S_x and S_y, running on the partition leaders of the records, respectively. By Lemma 3.1, both `reserve(`x`)` and `reserve(`y`)` must happen after `lock(`x`)` at x's leader and `lock(`y`)` at y's leader, respectively. Figure 6 gives an illustration of that, in which M's `reserve(`x`)` arrives x's leader at 5 after `lock(`x`)` by S_x at 1; and M's `reserve(`y`)` arrives y's leader at 9 after `lock(`y`)` by S_y at 7.

Now, consider the only two situations, where the leader of a locked item is either (a) co-located or (b) not co-located with M's coordinator. As M is an MR transaction, the leaders of x and y should come from different regions. Without loss of generality, we assume x is co-located with M's coordinator, i.e., case (a); and y is not co-located with M's coordinator, i.e., case (b).

With the presence of case (b), the critical path of M lies on case (b) but not case (a), because the validation of y at 12 requires WAN communication in its atomic commit (AC) process but the validation of x at 11 does not.

Consider the validation of y at 12. It is expected to take one WAN round-trip for the `rpc_lock(`y`)` RPC plus some time to wait

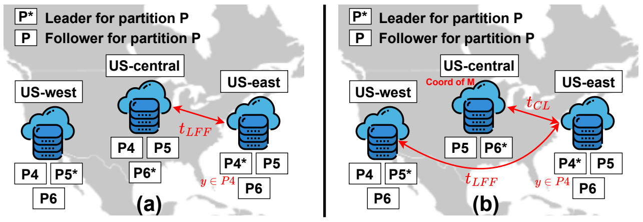

Figure 7: (a) t_{LFF} is a typically short; (b) a contrived setup.

for S_y to release the lock at ⑬ when the RPC arrives at y's leader but S_y has not yet unlocked y. However, we can show that S_y must have unlocked y by then.

Consider the duration from M's `rpc_reserve(y)` arriving at y's leader at ⑨ to M's `rpc_lock(y)` arriving at y's leader at ⑬. We denote this duration as D1. According to Lemma 3.1, M's `rpc_reserve(y)` at ⑨ must arrive after S_y's `lock(y)` at ⑦. We denote the duration between S_y's `lock(y)` at ⑦ and M's `rpc_reserve(y)` at ⑨ as D2. Additionally, we denote the duration between the arrival of M's `rpc_reserve(y)` at y's leader at ⑨ and S_y's `unlock(y)` at ⑩ as D3. Together, D2 and D3 constitute the lock-holding time of S_y on y. D3 alone is strictly shorter than the lock-holding time.

Now it is clear that M does not need to wait for S_y to release the lock (i.e., it is wait-free) as long as D3 is shorter than D1. Note that (1) D3 is strictly shorter than S_y's lock-holding time, which is the duration started from ⑧ for logging the write set WS_y across the WAN, which is roughly the RTT between the leader of y and the farthest follower in its nearest quorum; (2) D1 (from ⑨ to ⑬) is strictly greater than the RTT between M's coordinator and y's leader. Thus, the lemma is proven. □

Lemma 3.2 implies that, under one condition, an MR transaction does *not* need to wait for a conflicting SR transaction to release the lock. The only question is whether that condition – namely, that the RTT t_{LFF} from the leader of y to the farthest follower in its nearest quorum is shorter than the RTT t_{CL} from the coordinator of M to the leader of the remote record y – always holds, or how likely it is that it does not hold.

Typically, t_{LFF} is short, as most geo-distributed deployments are designed to form a fast quorum. In contrast, t_{CL} depends on the proximity of the coordinator of M to the remote leader. However, in most deployments, $t_{LFF} \leq t_{CL}$ should hold. Consider the example in Figure 7(a), which follows the setup in Figure 4. For a record $y \in P4$ with its leader in `US-east`, t_{LFF} (indicated by the red arrow) represents the RTT between y's leader `US-east` (where $P4^*$ resides in) and the farthest follower, `US-central`, in the nearest quorum {`US-central`, `US-east`} from the leader. Consequently, regardless of the location of the coordinator of M, we have $t_{LFF} \leq t_{CL}$.[2]

Of course, one could always contrive a setup that makes t_{LFF} very long, thereby violating the condition in Lemma 3.2. For example, in Figure 7(b), one follower replica of $P4$ is moved from `US-central` to a distant region (such as Asia; outside the picture). When placing the coordinator of M in `US-central`, then the nearest quorum from the leader $P4^*$ becomes {`US-west`, `US-east`}. In this case, t_{LFF} is the RTT between $P4$ in `US-west` and y's leader $P4^*$ in `US-east`, whereas t_{CL} is the RTT between M's coordinator (`US-central`) and y's leader (`US-east`), resulting in $t_{CL} < t_{LFF}$. However, such a scenario is practically avoidable when deciding on the replication plan, and indeed, many actual deployments deliberately avoid so [34, 62, 70].

Another possibility that may violate the condition involves live WAN traffic and workload burstiness. For example, when a bursty workload consumes more bandwidth than the provisioned WAN capacity, congestion in the WAN message queue may result in $t_{CL} < t_{LFF}$. These external factors can be mitigated by over-provisioning or dynamic re-provisioning of WAN bandwidth (e.g., [39]), which is beyond the scope of this paper.

Read Set Cross-Check. After locking the records in the write set, the transaction checks whether the records in the read set are locked or updated by other transactions. If not, validation succeeds. If any record has been updated, the transaction must abort, as it has read a stale version. If any record is locked, the transaction waits for the lock to be released and then checks for updates again. The wait time in this scenario is also minimal, similar to the wait experienced in Lock-Lock Conflict Resolution discussed above.

3.1.4 *Commit phase: Early Visible Write.*

Although MR transactions' reservation scheme enables an MR transaction M to be genuinely wait-free during the execution phase (see Section 3.1.2), it also exposes M to reading records locked by SR transactions awaiting updates, ultimately causing M to abort. In other words, without any remedy, the benefit of a wait-free MR execution phase comes at the cost of losing MR Abort-Free.

In GDCC, we leverage *early visible write* (EVW) [6, 28, 30–32, 36] in SR transactions to solve the problem. Specifically, in traditional OCC systems, the commit phase follows this sequence:

(1) write the log to the durable storage (write-ahead logging);
(2) publish the write set in the memory, visible to other transactions;
(3) release the locks on the write set; and
(4) apply the changes to the durable storage.

In GDCC, we advocate for the sequence of (2) ⟶ (1) ⟶ (3) ⟶ (4) for SR transactions instead, making the write set of SR transactions `early_visible()` on the leader. By doing so, MR transaction M can always read the latest records updated by SR transactions before the latter carries out the slow logging across the WAN.

Note that this early write visibility is both serializable and recoverable because the latest records updated by SR transactions, as seen by M, have passed validation with their locks held. GDCC's EVW is therefore distinct from the EVW in prior work [6, 28, 30–32, 36], which is based on 2PL [12] and not only makes the writes early visible but also releases the locks early (i.e., their EVW follows the sequence (2) ⟶ (3) ⟶ (1) ⟶ (4)).

When locks are released early as in prior work, significant effort is required to ensure serializability and recoverability (e.g., reordering the commit points [28, 30, 31]). In contrast, GDCC does not have these complications because its EVW retains the locks until logging is completed. Consequently, any transaction that reads these fresh updates must commit after the transactions that expose them – both serializability and recoverability are naturally maintained.

Another complication associated with EVW is the potential for cascading aborts. Specifically, any transaction that reads a dirty

[2]Since M is accessing y with `US-east` as y's leader, M's coordinator is not in `US-east` under non-co-locating case (b).

write exposed by a transaction T must abort if T eventually aborts. In prior work that adopts EVW, this issue is either avoided by design (e.g., deterministic databases do not abort transactions unless facing a server failure [28]) or mitigated via various methods (e.g., limiting the length of the cascading abort chain [30, 31]).

GDCC falls into the former category – cascading aborts do not occur by design unless there is a server failure. This is because only SR transactions use EVW, and they expose their dirty writes *only after successful validation*. In other words, EVW-enabled SR transactions generally do not abort, so there is no cascading abort issue. For MR transactions, since they do not use EVW, they would not induce cascading aborts either.

Of course, like any other systems that use EVW (e.g., [28, 32]), SR transactions utilizing EVW would need to be aborted in case of a server failure, even after they have successfully validated. However, using EVW in GDCC has the advantage of limiting the cascading abort chain length to one, even in the face of server failure.

Specifically, if a transaction T reads the early visible write of an SR transaction S, and S must be aborted due to a server failure, the cascading abort chain will stop at T if T is an MR transaction, as MR transactions do not use EVW. If T is an SR transaction, the cascading abort chain will also stop at T. Since S is a successfully validated transaction, it will still hold the lock, preventing T from validating successfully and triggering EVW, thereby stopping the cascading abort chain from extending further.

3.1.5 Multi-Priority Optimization and Summary. GDCC includes an optimization that increases the priority level of an MR transaction M by one whenever it has been aborted a times, where a is a system parameter we empirically set to 2. With multiple priorities among MR transactions, a reservation made by a lower-priority MR transaction M will FAIL if the record has been reserved by a higher-priority MR transaction M' (reserve-reserve conflict), and M will WAIT if it wants to lock instead (lock-reserve conflict). It is important to note that M failing to reserve simply means it does not shield itself from being aborted by a higher-priority MR transaction M'. However, it still prevents SR transactions from aborting M, as the reservation made by M' remains in effect. The complete pseudocode with multiple-priority optimization is available in [5].

In summary, transaction schedules in GDCC are **serializable** because its validation step adheres to the one in standard OCC. These schedules are also **recoverable** because locks are held until the end of the commit phase. Furthermore, there are **no cascading aborts**, as EVW only exposes dirty writes of transactions that will commit, thereby limiting the length of the cascading abort chain to one even if there is a server failure.

MR transactions in GDCC are **abort-free** against SR transactions by using reservations. SR transactions in GDCC are **wound-free**, as others will ALWAYS WAIT for them once they obtain their locks during validation. MR transactions in GDCC are **wait-minimal** against SR transactions. They are completely wait-free during their execution phase. Since their execution and validation phases overlap with the lock-holding window of the SR lock holder, this buys the SR lock holder time to release the lock while the MR transaction is executing and sending the atomic commit request, enabling MR transactions to achieve wait-free performance in most circumstances. GDCC, however, does not guarantee that SR transactions are starvation-free, as priority is given to MR transactions after all. Nevertheless, SR starvation is unlikely because MR transactions are relatively few in real workloads, and SR transactions already have wound-free protection. Our empirical results in Section 4 also support this – the tail latency of SR transactions in Bonspiel consistently remain low in our experiments, indicating that SR starvation is a non-issue. Table 5 summarizes GDCC in relation to existing works, focusing on the MR Abort-Free, SR Would-Free, and MR Wait-Minimal principles that define a good concurrency control protocol for geo-distributed databases. The table demonstrates that GDCC is the only protocol that satisfies all three principles.

Table 5: GDCC and related CC protocols

Protocols	MR Abort-Free	SR Wound-Free	MR Wait-Minimal
Sundial [92], NO-WAIT [71]	✗	✓	✓
WOUND-WAIT [71]	✓	✗	✓
Polaris [89], WAIT-DIE [71]	✓	✓	✗
GDCC (ours)	✓	✓	✓

3.2 Geo-Aware Access Method Selection (GAMS)

Reducing the latency of the **execution phase** of an MR transaction M can further lead to a reduction in tail latency, both in terms of the abort penalty and the execution time of the last successfully committed round of M. As discussed in Section 2.2, there are currently two approaches for reading records in a geo-distributed database: (1) `read-leader` (RL), which always reads a record from the record's leader replica, and (2) `read-nearest` (RN), which always reads a record from the replica closest to the transaction coordinator. When the transaction coordinator is co-located with the record's leader replica, RL is clearly the optimal choice, offering the lowest latency and the freshest data.

However, when the transaction coordinator is *not* co-located with the record's leader replica, RL incurs longer latency although the data it reads remain freshest. In contrast, RN incurs shorter latency but carries the risk of returning stale data, which may result in an abort. Nonetheless, this risk is contingent on the *update frequency* of the required records — if the required record is not frequently updated, RN can achieve optimal latency while still providing fresh data.

To leverage the potential of RN in Bonspiel, we must first address its compatibility with GDCC's *reservation*. Specifically, reservation requests in GDCC must be sent to the remote leader of the record; therefore, RN alone cannot reduce the latency unless GDCC's reservation is disabled. In fact, RN with the NO-RESERVE option is a competitive choice when the required record is not frequently updated, as reserving that record or not does not matter much there.

In light of that, Bonspiel maintains *temperatures* [82] to selectively issue RN for reading cold records (with no reservation) and RL for reading hot records (with reservation). A record's temperature is measured by its update frequency. Temperatures can be maintained at per-record or per-page level — a trade-off between accuracy and space/network overhead. In Bonspiel, we maintain page-level temperatures. In a full replication setting, sites can collect these statistics locally from the leader and follower replicas it hosts. In a partial replication setting, each data center periodically gossips these statistics among one another.

When an MR transaction reads a record x during the execution phase, Bonspiel uses RN with no reservation if the temperature of x is below a threshold t; otherwise, it employs RL with reservation. Bonspiel adopts an experimental-driven approach [81] to determine the value of t. Specifically, the abort rate of MR transactions effectively reflects workload and network conditions (e.g., workload drifts, network congestion). Hence, t is lowered if MR transactions experience a high abort rate, encouraging more MR transactions to read records from the leader and use reservations for abort protection. Otherwise, t is increased to allow more MR transactions to attempt RN without reservation.

GAMS enables transactions to select the most suitable access method for each record, thereby improving latency and system throughput. Take TPC-C as an example, records in the `WAREHOUSE` table are frequently updated by `Payment` transactions – typically exhibiting high update frequency, so GAMS would let transactions access them via RL with reservation to minimize aborts. In contrast, the `ITEM` table is read-only; thus, records in this table are accessed via RN without reservation for shorter execution latency. Besides, by dynamically adapting to live workload and network conditions, GAMS excels across diverse workloads and maintains robustness under workload drift.

4 EVALUATION

We evaluate Bonspiel using industrial-strength TPC-C [1]. We utilize YCSB [21] when we need to vary certain parameters that TPC-C does not allow for modification (e.g., the MR transaction ratio). In both cases, each client issues one transaction and waits for its commit before issuing the next [41, 42, 91, 92]. Aborted transactions are retried using the exponential backoff strategy [35].

In TPC-C, we run `NEW-ORDER` and `PAYMENT` transactions, following the specification with 10% and 15% of them as MR transactions, respectively. We use 300 warehouses for *low contention* and 50 warehouses for *high contention*. In YCSB, each transaction accesses 10 keys, selected based on a Zipfian distribution. We use $\theta = 0.2$ for *low contention* and $\theta = 0.8$ for *high contention* in YCSB.

All programs were implemented using the DBx1000 framework in C++ [90]. The experiments were conducted in a simulated WAN setting comprising five data centers: Virginia (VA), Washington (WA), Paris (PR), New South Wales (NSW), and Singapore (SG).

Table 6: WAN round-trip latency (in ms).

	WA	PR	NSW	SG
VA	67	80	196	214
WA	-	136	175	163
PR	-	-	234	149
NSW	-	-	-	87

The average network round-trip latencies between these data centers are based on [87] and presented in Table 6. Each data center runs on Debian 11 servers with two Intel Xeon E5-2620V4 CPUs (8 cores) and 252 GB of DRAM. The WAN bandwidth is 1 Gbps, along with 10 Gbps in-data-center bandwidth. In the experiments, the data is sharded into five partitions (P) and replicated (R) five times across the five data centers (5P5R).

4.1 System-Level Evaluation

We first examine the p999 tail latency, p50 latency, average latency, and system throughput of Bonspiel, comparing it to state-of-the-art general geo-distributed databases: Spanner [22], TAPIR [93], GPAC [55], and R4 [41]. Details of all systems are given in Table 7. We do not compare with geo-distributed databases that require prior knowledge about the read and write sets of transactions (e.g., deterministic databases [29, 34, 43, 61, 62, 62, 70, 79] and some other non-deterministic ones [85, 87, 94]) since TPC-C includes dependent operations that they cannot natively support. Besides, we also exclude Replicated Commit [53] since it has no liveness – the system blocks upon single-data-center outages [41] and cannot support partial replication.

Table 7: Bonspiel and system competitors.

System	Concurrency control	Access method	Atomic commit with replication
Spanner	WAIT-DIE	Always RL	2PC&Paxos (3 RTTs)
TAPIR	OCC	Always RN	Optimized 2PC&Fast Paxos (1-2 RTTs)
GPAC	WAIT-DIE	Always RL	Optimized 2PC&Paxos (2 RTTs)
R4	OCC	Always RL	Optimized 2PC&Paxos (1-1.5 RTTs)
Bonspiel	**GDCC**	**GAMS**	Same as R4

4.1.1 *Overall Performance.*

Figure 8 illustrates the results of scaling the number of clients in TPC-C. In terms of p999 tail latency, Bonspiel achieves a speedup ranging from 1.8× to 2.2×. Notably, Bonspiel reduces the tail latency from around 3–4 seconds to around 1.7 seconds, significantly enhancing the user experience and the product image. More importantly, Bonspiel improves tail latency without compromising average latency, p50 latency or system throughput. In fact, it exhibits the best average latency and system throughput, primarily due to significant improvements in the performance of MR transactions. Figure 9 illustrates the performance of SR transactions in TPC-C, clearly indicating that Bonspiel maintains top-tier performance for SR transactions. Figures 9(a) and (b) also confirm that Bonspiel's design does not result in any starvation issues for SR transactions, as evidenced by the consistently low p999 tail latency for SR transactions in Bonspiel.

In Figure 8(b), we observe that the tail latencies of the other systems surprisingly decrease as the number of clients (system load) increases under high contention. In fact, the average latency of their MR transactions worsens by about 28% to 35% (figures omitted due to space limit) with an increasing number of clients. However, this increased latency results in fewer, and indeed too few, MR transactions being committed during each experimental run. Consequently, when calculating the tail latency, they fall beyond the 99.9 percentile. As a result, that tail latency reports either the latency of an SR transaction or the latency of MR transactions that incur no/few conflict, which explains the decrease in tail latency observed in those systems.

To further investigate the impact of Bonspiel's prioritization of MR transactions over SR transactions, we present the abort rates[3] and the number of aborts for the transactions in Figure 10. Specifically, Bonspiel significantly reduces the abort rate of MR transactions while causing only a marginal, expected increase in the abort rate of SR transactions and the overall abort rate. Importantly, the SR abort rate remains well-controlled and comparable to

[3]TPC-C abort rates are generally higher than those of YCSB. However, TPC-C abort rates remain high even under low contention in geo-distributed settings because WAN latency significantly extends transaction contention footprint, creating large temporal windows for transaction conflicts, while `PAYMENT` and `NEW-ORDER` transactions both contend for the small `WAREHOUSE` table, resulting in frequent conflicts and thus aborts.

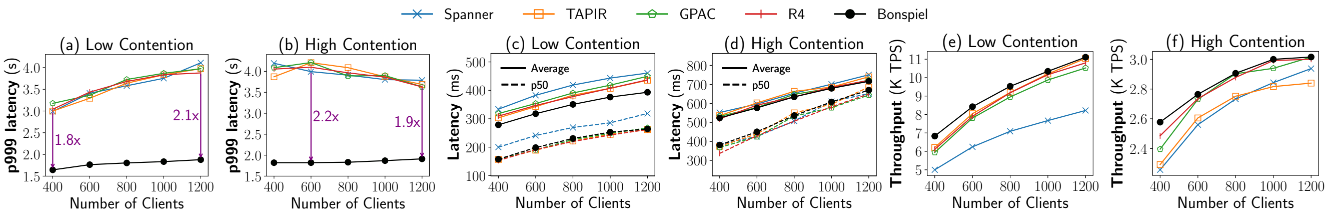

Figure 8: Overall performance (TPC-C).

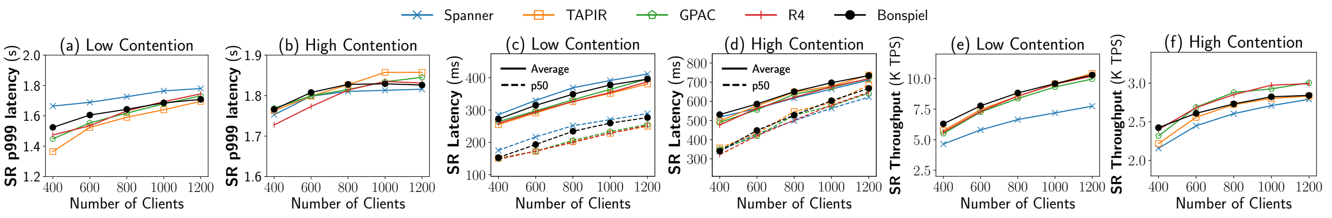

Figure 9: SR transaction performance (TPC-C).

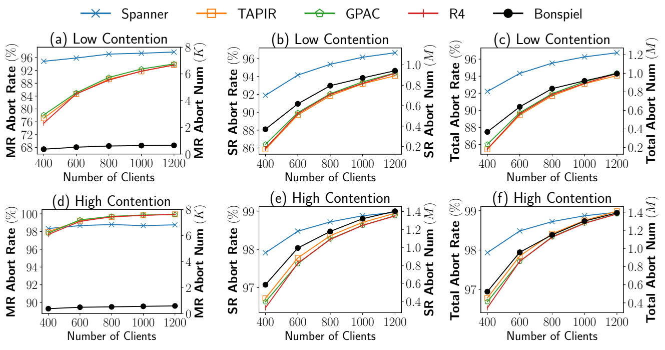

Figure 10: Abort analysis (TPC-C).

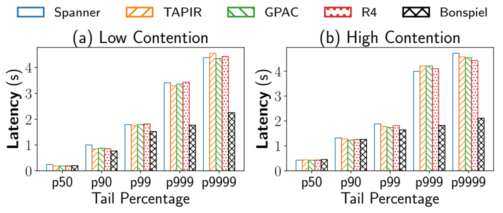

Figure 11: Various tail latency results (TPC-C).

other systems, thanks to Bonspiel's enforcement of the SR Wound-Free principle – Bonspiel does not unconditionally prioritize MR transactions; once an SR transaction has successfully validated its read-write set, it is guaranteed not to be aborted by any other transaction. Additionally, because MR transactions represent only a small fraction of realistic workloads, prioritizing them has a limited impact on the abort rate of SR transactions or the overall abort rate.

4.1.2 Other Tail Latency Metric. Next, we report the performance of Bonspiel in terms of additional tail latency metrics: p50, p90, p99, p999, and p9999. As shown in Figure 11, Bonspiel delivers the lowest tail latency across varying tail percentages, with improvements becoming more pronounced as the tail percentage increases. This is because more stringent tail percentages report the latency of MR transactions that suffer from higher abort rates and penalties. These transactions particularly benefit more from Bonspiel's MR-transaction-friendly design.

4.1.3 Varying MR Transaction Ratio. TPC-C has fixed the MR transaction ratio at around 10% to 15%. Therefore, we use the YCSB workload in this experiment to study the performance of Bonspiel under different MR transaction ratios. The ratio of read to write operations in each transaction is 50:50 (i.e., YCSB-A). Figure 12 presents the results, showing that Bonspiel consistently delivers the best tail latency, average latency, and system throughput across all cases. Interestingly, Bonspiel demonstrates significant improvement over the other systems even when the workload consists of 100% MR transactions. In the absence of SR transactions, none of the three GDCC properties (MR Abort-Free, SR Wound-Free, and MR Wait-Minimal) are relevant. Consequently, the improvement is contributed by (1) Bonspiel's geo-aware access method selection, which effectively reduces the execution phase latency of MR transactions, and (2) the multi-priority optimization in GDCC (Section 3.1.5), which balances the abort rate among MR transactions.

4.2 Ablation Study

After demonstrating that Bonspiel outperforms other geo-distributed databases across various settings, and given that we have already shown in Section 1 that the use of the atomic commit protocol is not the primary factor in tail latency, we now present an ablation study to examine the effectiveness of its two major components: GDCC and GAMS.

Figure 13 illustrates results of the ablation study. We compare the performance of Bonspiel under three scenarios: when only GDCC is enabled (`GDCC-only`), with GAMS disabled; when only GAMS is enabled (`GAMS-only`), with GDCC disabled; and when both GDCC and GAMS are disabled (`w/o both`), which is essentially equivalent to R4's performance. Our findings indicate that both GDCC and GAMS significantly enhance the system's tail latency when used individually, demonstrating their individual effectiveness.

The inclusion of GDCC has a positive effect on average latency under low contention (Figure 13c) but minimal impact under high contention (Figure 13d). Under low contention, average latency improves due to the decreased latency of MR transactions. In contrast, during high contention, average latency remains unchanged, despite the reduction in MR transaction latency. This occurs because

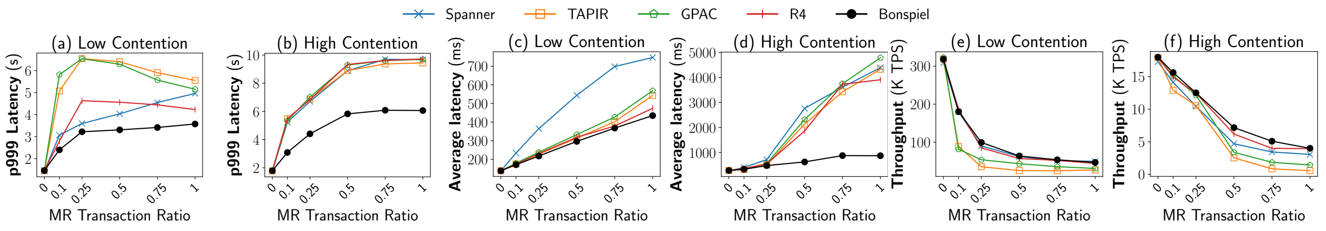

Figure 12: Performance with varying MR transaction ratios (YCSB-A).

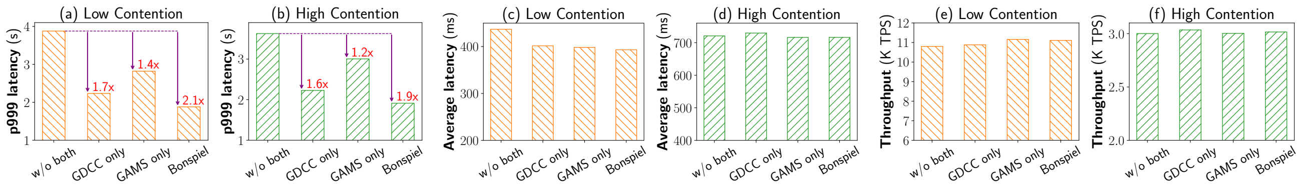

Figure 13: Ablation study (TPC-C).

the latency of SR transactions slightly increases as a result of prioritizing MR transactions. The small increase in latency for many SR transactions offsets the significant improvement in latency from the fewer MR transactions, leading to limited overall improvement in average latency.

4.3 More Results

For space reasons, we have included the results of additional experiments in [5]. These encompass:

- A detailed evaluation of GDCC: (i) comparisons with other concurrency control schemes (Sundial [92], NO-WAIT, WAIT-DIE, WOUND-WAIT, Polaris), showing GDCC achieves the best balance; (ii) analysis of the wait time of MR transactions for SR lock releases, indicating that 99% of MR transactions are wait-free; and (iii) a detailed breakdown of GDCC's key components: (a) priority scheme, (b) conditional waiting, and (c) early visible write, demonstrating their synergy.
- A detailed evaluation of GAMS: (i) comparisons with always RL and always RN, showing that GAMS is able to select the best-performing access method in all cases; (ii) positive results under unstable network, and (iii) the empirical choice of the priority boosting threshold.
- Further results from TPC-C and YCSB, including results using a set up with 3 partitions and 2-way replication (3P2R) and results of varying the number of partitions under full replication. Those results show that Bonspiel consistently outperforms the others in terms of tail latency while maintaining top-tier average latency and system throughput.

5 RELATED WORK

The community has long pursued a **general** geo-distributed database that offers low tail latency, low average latency, high throughput, and high availability, without imposing any workload limitations. However, apart from Spanner [22], TAPIR [93], GPAC [55], and the recent R4 [41], most advancements have focused on solutions that entail certain compromises. For instance, many systems sacrifice strong consistency [9, 13, 17, 20, 25, 44, 49, 50, 75, 78, 83, 84]. Deterministic databases [29, 34, 61, 62, 62, 70, 79] achieve strong consistency and performance by minimizing coordination overhead and avoiding aborts induced by serializability. However, these databases require prior knowledge of the read-write sets and void SQL workloads with non-deterministic elements (e.g., branches or DATE). Recent deterministic databases [42, 43, 51] have lifted these workload restrictions but have reintroduced serializability-induced aborts, which are a primary contributor to high latency in geo-distributed environments. Additionally, they rely on batching. While batching may enhance throughput, it is unacceptable in geo-distributed settings, as it further prolongs tail latency, which is already very high. Other non-deterministic geo-distributed databases also enhance performance by limiting their supported workloads [7, 23, 60, 85, 87, 94, 97]. For example, Carousel [85] and Natto [87] only support 2FI transactions. In contrast, Bonspiel accommodates general SQL workloads while maintaining strong consistency and high performance.

Low tail latency is a concern not only for geo-distributed databases but for any system in general. Numerous studies have aimed to reduce tail latency through core scheduling [16, 18, 37, 48, 64, 68, 69, 76, 87], queue management [26], and caching [11]. These approaches are orthogonal to Bonspiel.

6 CONCLUSION

This paper presents Bonspiel, a fully SQL-compliant geo-distributed database with low tail and average latency as well as high system throughput. As state-of-the-art protocols can now perform atomic commit in 1 WAN RTT, Bonspiel focuses on geo-aware concurrency control and access method selection. Experimental results demonstrate that Bonspiel achieves up to 2.2× improvement in tail latency without any compromise.

ACKNOWLEDGMENTS

This work is partially supported by Hong Kong General Research Fund (14208023), Hong Kong AoE/P-404/18, the Center for Perceptual and Interactive Intelligence (CPII) Ltd under InnoHK supported by the Innovation and Technology Commission, Alibaba Cloud (Innovative Research Program and Research Intern Program) and Google Cloud Platform (Research Credits Program).

REFERENCES

[1] 2010. TPC-C BENCHMARK Revision 5.11.

[2] 2024. Cockroach DB: transaction priority. https://www.cockroachlabs.com/docs/v21.2/transactions#transaction-priorities.

[3] 2024. Oracle: Berkeley DB C++ API Reference. https://docs.oracle.com/database/bdb181/html/api_reference/CXX/txnset_priority.html.

[4] 2024. SQL Server: SET DEADLOCK_PRIORITY. https://learn.microsoft.com/en-us/sql/t-sql/statements/set-deadlock-priority-transact-sql?view=sql-server-ver15.

[5] [Online]. *Technical Report for Bonspiel.* https://github.com/Bonspiel-Project/Bonspiel.git

[6] Divyakant Agrawal and Amr El Abbadi. 1990. Locks with constrained sharing. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.* 85–93.

[7] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: a new paradigm for building scalable distributed systems. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 159–174.

[8] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. (2011).

[9] Valter Balegas, Nuno Preguiça, Sérgio Duarte, Carla Ferreira, and Rodrigo Rodrigues. 2018. IPA: Invariant-preserving applications for weakly-consistent replicated databases. *arXiv preprint arXiv:1802.08474* (2018).

[10] Catalonia-Spain Barcelona. 2008. Mencius: building efficient replicated state machines for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08).*

[11] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. {RobinHood}: Tail Latency Aware Caching–Dynamic Reallocation from {Cache-Rich} to {Cache-Poor}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* 195–212.

[12] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. 1979. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering* 3 (1979), 203–216.

[13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}: Facebook's distributed data store for the social graph. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13).* 49–60.

[14] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, et al. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE).* IEEE, 2859–2872.

[15] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 209–223.

[16] Michael J Carey, Rajiv Jauhari, and Miron Livny. 1989. *Priority in DBMS resource scheduling.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[18] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 210–227. https://doi.org/10.1145/3447786.3456238

[19] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jiwu Shu. 2022. Plor: General Transactions with Predictable, Low Tail Latency. In *Proceedings of the 2022 International Conference on Management of Data.* 19–33.

[20] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1277–1288.

[21] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing.* 143–154.

[22] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[23] James A Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *USENIX Annual Technical Conference*, Vol. 12.

[24] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. 2010. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1 (2010), 48–57. https://doi.org/10.14778/1920841.1920853

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[26] Diego Didona and Willy Zwaenepoel. 2019. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19).* 79–94.

[27] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems.* 1–15.

[28] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[29] Hua Fan and Wojciech Golab. 2019. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1471–1484.

[30] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. 2013. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.* 85–96.

[31] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *Proceedings of the 2021 International Conference on Management of Data.* 658–670.

[32] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham. 1997. Revisiting commit processing in distributed database systems. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data.* 486–497.

[33] Pat Helland. 2022. Decoupled Transactions: Low Tail Latency Online Transactions Atop Jittery Servers. In *12th Conference on Innovative Data Systems Research.*

[34] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. 2023. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *Proceedings of the VLDB Endowment* 17, 3 (2023), 469–482.

[35] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for optimism in contended main-memory multicore transactions. (2020).

[36] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: a scalable approach to logging. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 681–692. https://doi.org/10.14778/1920841.1920928

[37] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for {μsecond-scale} Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19).* 345–360.

[38] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems.* 113–126.

[39] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. 2015. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication.* 1–14.

[40] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[41] Ziliang Lai, Fan Cui, Hua Fan, Eric Lo, Wenchao Zhou, and Feifei Li. 2024. Occam's Razor for Distributed Protocols. In *Proceedings of the 2024 ACM Symposium on Cloud Computing.* 618–636. https://doi.org/10.1145/3698038.3698514

[42] Ziliang Lai, Hua Fan, Wenchao Zhou, Zhanfeng Ma, Xiang Peng, Feifei Li, and Eric Lo. 2023. Knock Out 2PC with Practicality Intact: a High-performance and General Distributed Transaction Protocol. *arXiv preprint arXiv:2302.12517* (2023).

[43] Ziliang Lai, Chris Liu, and Eric Lo. 2023. When private blockchain meets deterministic database. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–28.

[44] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[45] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[46] Leslie Lamport. 2005. Generalized consensus and Paxos. (2005).

[47] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19 (2006), 79–103.

[48] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1091–1104.

[49] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage

with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 401–416.

[50] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Symposium on Networked Systems Design and Implementation*.

[51] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2047–2060.

[52] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-based commit and replication in distributed OLTP databases. (2021).

[53] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment* 6, 9 (2013), 661–672.

[54] Hatem A Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proceedings of the VLDB Endowment* 7, 5 (2014), 329–340.

[55] Sujaya Maiyya, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2019. Unifying consensus and atomic commitment for effective cloud data management. *Proceedings of the VLDB Endowment* 12, 5 (2019), 611–623.

[56] Puya Memarzia, Huaxin Zhang, Kelvin Ho, Ronen Grosman, and Jiang Wang. 2024. GaussDB-Global: A Geographically Distributed Database System. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 5111–5118.

[57] Olivier Michaelis, Stefan Schmid, and Habib Mostafaei. 2024. L3: Latency-aware Load Balancing in Multi-Cluster Service Mesh. In *Proceedings of the 25th International Middleware Conference*. 49–61.

[58] C Mohan, Bruce Lindsay, and Ron Obermarck. 1986. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 378–396.

[59] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 358–372.

[60] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting more concurrency from distributed transactions. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 479–494.

[61] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts.. In *OSDI*. 517–532.

[62] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[63] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 305–319.

[64] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[65] Stacy Patterson, Aaron J Elmore, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2012. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *arXiv preprint arXiv:1208.0270* (2012).

[66] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, Scottsdale Arizona USA, 61–72. https://doi.org/10.1145/2213836.2213844

[67] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, Portland OR USA, 527–542. https://doi.org/10.1145/3318464.3389764

[68] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 325–341.

[69] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne:{Core-Aware} thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 145–160.

[70] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019).

[71] Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis. 1978. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS)* 3, 2 (1978), 178–198.

[72] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 445–456. https://doi.org/10.14778/3025111.3025125

[73] Ji-Yong Shin, Jieung Kim, Wolf Honoré, Hernán Vanzetto, Srihari Radhakrishnan, Mahesh Balakrishnan, and Zhong Shao. 2019. Wormspace: a modular foundation for simple, verifiable distributed systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 299–311.

[74] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. 2013. F1: A distributed SQL database that scales. (2013).

[75] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 385–400.

[76] P. Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 513–527. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/suresh

[77] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[78] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 172–182.

[79] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.

[80] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. 18–32.

[81] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 198–216. https://www.usenix.org/conference/osdi21/presentation/wang-jiachen

[82] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proc. VLDB Endow.* 10, 2 (2016), 49–60. https://doi.org/10.14778/3015274.3015276

[83] Chenggang Wu, Jose M Faleiro, Yihan Lin, and Joseph M Hellerstein. 2019. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering* 33, 2 (2019), 344–358.

[84] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. 2019. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment* 12, 6 (2019), 624–638.

[85] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*. 231–243.

[86] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Trans. Storage* 17, 3 (2021), 17:1–17:35. https://doi.org/10.1145/3468521

[87] Linguan Yang, Xinan Yan, and Bernard Wong. 2022. Natto: Providing distributed transaction prioritization for high-contention workloads. In *Proceedings of the 2022 International Conference on Management of Data*. 715–729.

[88] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397.

[89] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. 2023. Polaris: Enabling Transaction Priority in Optimistic Concurrency Control. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–24.

[90] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. (2014).

[91] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. Tictoc: Time traveling optimistic concurrency control. In *SIGMOD*. 1629–1642.

[92] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: harmonizing concurrency control and caching in a distributed oltp database management system. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1289–1302.

[93] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 263–278.

[94] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. 2013. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 276–291.

[95] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. 2023. Scalable tail latency estimation for data center networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 685–702.

[96] Weixing Zhou, Qi Peng, Zijie Zhang, Yanfeng Zhang, Yang Ren, Sihao Li, Guo Fu, Yulong Cui, Qiang Li, Caiyi Wu, et al. 2023. GeoGauss: Strongly Consistent and Light-Coordinated OLTP for Geo-Replicated SQL Database. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[97] Qiyu Zhuang, Xinyue Shi, Shuang Liu, Wei Lu, Zhanhao Zhao, Yuxing Chen, Tong Li, Anqun Pan, and Xiaoyong Du. 2024. GeoTP: Latency-aware Geo-Distributed Transaction Processing in Database Middlewares (Extended Version). *arXiv preprint arXiv:2412.01213* (2024).

Appendix A GDCC PSEUDOCODE

Figure 14 and Figure 15 provide illustrations of the full GDCC protocol, including the multiple-priority optimization discussed in Section 3.1.5. In GDCC, each transaction is assigned a priority level, initially set to 0 (Figure 14, line 4). SR transactions consistently maintain a priority level of 0, whereas MR transactions have their priority level elevated after every k aborts (Figure 14, lines 6-8). The value of k is tunable; in our implementation, we set k to 2. This approach ensures that MR transactions, which are aborted multiple times, receive higher priority to mitigate potential starvation.

During the execution phase, transactions perform read/write operations (Figure 14, lines 9-20). If a transaction accesses multiple regions, it is classified as an MR transaction, and its priority level is raised to at least 1 (Figure 14, lines 11-13). To execute a read/write operation on x, the transaction sends RPC calls to reserve and access x (Figure 14, lines 14-16). For clarity, we assume all writes are read-modify-writes in the pseudocode; blind writes do not require access during the execution phase but still necessitate reservation.

The `access_method_selection()` function in Figure 14 (line 14) pertains to geo-aware access method selection (GAMS), as discussed in Section 3.2. Notably, since SR transactions consistently maintain the lowest priority level of 0, their reservations are unnecessary. Thus, SR transactions do not reserve records.

The destination node `dest` executes the `access` RPC call locally. If a reservation is required for x, the RPC call waits until x is unlocked or its value is *validated* (i.e., early writes from validated SR transactions), after which it attempts to reserve x using the transaction's priority level `prio` (Figure 15, lines 7-12).

If x is not reserved or is reserved by a same-priority or lower-priority transaction, the reservation succeeds. Conversely, if x is reserved by a higher-priority transaction, the reservation fails (Figure 15, lines 15-33). In either case, the `access` RPC call returns the value of x (Figure 15, line 14). Upon receiving the return value, the transaction includes it in the read set `RS` and in the write set `WS` if it is a write operation (Figure 14, lines 17-20).

After executing all operations, the transaction enters the validation phase and makes a commit-vs-abort decision via validation (Figure 14, line 22). For SR transactions, this process is equivalent to performing local validation; for MR transactions, it follows the atomic commit protocol. If the decision is to commit, MR transactions follow a standard procedure to first log the write set `WS`, then make the write set visible to other transactions; afterwards, they yield all locks and reservations and apply the changes to follower replicas (Figure 15, lines 24-30). In contrast, SR transactions employ Early Write Install (c.f., Section 3.1.4), immediately making the write set visible before logging (Figure 15, line 33). This includes marking records within `WS` as *validated*. After logging (Figure 15, line 34), locks and reservations are released and changes are applied on follower replicas (Figure 15, lines 35-38).

On the contrary, if the decision is to abort, the transaction discards its write set and increments `abort_num` (Figure 15, line 40).

Appendix B PROOF OF CORRECTNESS

In this appendix, we formally prove the correctness of Bonspiel in terms of serializability. Specifically, we present and prove the following theorem.

```
 1: function PROCESS_TXN(tid)
 2:     // tid: txn id
 3:     if this is a new txn then
 4:         prio = 0
 5:         is_multi_region = false
 6:     else if is_multi_region then
 7:         // Priority of MP txns is lifted every k aborts
 8:         prio = abort_num / k + 1
 9:     while txn has more operations to execute do
10:         get next operation as <$x$, type>
11:         if txn accesses multiple partitions then
12:             is_multi_region = true
13:             prio = max(prio, 1)
14:         dest, to_reserve = access_method_selection($x$, prio)
15:         // Send rpc call to dest to access $x$
16:         val = access(tid, $x$, prio, type, dest, to_reserve)
17:         RS.add(<x, val>)
18:         if type == write then
19:             compute new value of $x$ as new_val
20:             WS.add(<$x$, new_val>)
21:     // For MR transactions, this is done by AC
22:     decision = validate(tid, RS, WS)
23:     if decision == commit then
24:         if is_multi_region then
25:             log(tid, WS)
26:             make_visible(tid, WS)
27:             unlock(tid, WS)
28:             unreserve(tid, RS, WS)
29:             // Apply change on follower replicas
30:             apply_change(tid, WS)
31:         else
32:             // Early Visible Write
33:             early_visible(tid, WS)
34:             log(tid, WS)
35:             unlock(tid, WS)
36:             unreserve(tid, RS, WS)
37:             // Apply change on follower replicas
38:             apply_change(tid, WS)
39:     else
40:         abort_num += 1
```

Figure 14: Pseudocode for transaction processing in GDCC.

THEOREM B.1. *Every transaction schedule generated by Bonspiel is serializable.*

We prove Theorem B.1 by showing that, each committed transaction in Bonspiel can be thought to execute in its entirety at the instant it takes locks on all records in its write set. Thus, for each transaction schedule S generated by Bonspiel, there is a conflict-equivalent serial schedule S': *the one in which the transactions appear in the same order as they take locks on all records in their write sets.* We first present and prove two useful lemmas.

LEMMA B.2. *Let T_1 and T_2 be two committed transactions in schedule S generated by Bonspiel. If T_1 takes locks on all records in its write set before T_2 does, then T_2 must see all updates made by T_1.*

```
function ACCESS(tid, $x$, prio, type, dest, to_reserve)
    // This rpc is executed on dest
    // tid: txn id
    // prio: priority level of the txn
    // type: read/write
    // to_reserve: whether to reserve $x$
    if to_reserve then
        // Wait for lock release or $x$ becomes validated
        while $x$.locked and NOT $x$.validated do
            wait
        // Try to reserve $x$
        try_reserve(tid, $x$, prio)
    // Return from dest to the rpc caller
    return $x$.value

function TRY_RESERVE(tid, $x$, prio)
    if $x$.reserved then
        if $x$.reserve_prio > prio then
            // Reserved by a higher-priority txn, fail
            return false
        if $x$.reserve_prio == prio then
            // Reserved by a same-priority txn, succeed
            $x$.reserve_list.add(tid)
            return true
        if $x$.reserve_prio < prio then
            // Reserved by a lower-priority txn, succeed
            $x$.reserve_list = [tid]
            $x$.reserve_prio = prio
            return true
    else
        $x$.reserved = true
        $x$.reserve_prio = prio
        $x$.reserve_list.add(tid)
        return true
```

Figure 15: Pseudocode for record accessing in GDCC.

PROOF. This is trivially true if the write set of T_1 does not intersect with the read set of T_2.

Let record x exist both in the write set of T_1 and the read set of T_2, i.e., T_1 updates x and T_2 reads x. In this case, we prove by contradiction that T_2 must see T_1's update on x. First, assume that T_2 does **not** see T_1's update on x.

Denote the moments when T_1 and T_2 take locks on all records in their write sets as t_1 and t_2. According to the condition in Lemma B.2, we have $t_1 < t_2$. Denote the moment when T_2 cross-checks its read set as t_2^*. In Bonspiel's GDCC, read set cross-checking happens after the transaction locks its write set. hence, we have $t_1 < t_2 < t_2^*$, meaning that when T_2 cross-checks x at t_2^*, T_1 must have already locked x. In this case, T_2 needs to wait for T_1 to apply its change, release the lock. Then T_2 cross-checks x.

Since T_1 has already applied its change when T_2 cross-checks record x, and T_2 does not see T_1's update on x, T_2's cross-check will fail and it will be aborted. However, T_2 is a committed transaction, and this creates a contradiction. Hence, T_2 must see T_1's updates on x. □

LEMMA B.3. *Let T_1 and T_2 be two committed transactions in schedule S generated by Bonspiel. If T_1 takes locks on all records in its write set before T_2 does, then T_1 must* **not** *see any updates made by T_2.*

PROOF. This is trivially true if the write set of T_2 does not intersect with the read set of T_1.

Let record y exist both in the write set of T_2 and the read set of T_1, i.e., T_2 updates y and T_1 reads y. In this case, we prove by contradiction that T_1 cannot see T_2's update on y. First, assume that T_1 reads T_2's update on y at time t_1^*.

Denote the moments when T_1 and T_2 take locks on all records in their write sets as t_1 and t_2. According to the condition in Lemma B.3, we have $t_1 < t_2$. In Bonspiel's GDCC, read operations of a transaction happen during the execution phase, which is strictly earlier than the time a transaction takes locks during the validation phase. Hence, for T_1, we have $t_1^* < t_1$. Besides, for T_1 to read T_2's update at t_1^*, T_2 must have made its updates visible before t_1^*, say, at t_2'. Consequently, we have $t_2' < t_1^*$.

In Bonspiel, even with Early Visible Write, a transaction can only make its updates visible to other transactions before it takes all locks and successfully cross-checks its read set. Hence, for T_2, we have that $t_2 < t_2' < t_1^*$. This, combined with $t_2 < t_1^*$, leads us to $t_2 < t_1$, which creates a contradiction! Hence, T_1 must **not** see T_2's update on x. □

PROOF OF THEOREM B.1. For any transaction schedule S generated by Bonspiel, we can construct a serial transaction schedule S': *the one in which the transactions appear in the same order as they take locks on all records in their write sets.*

From Lemma B.2 and Lemma B.3, we can conclude that any committed transaction T in S can see updates from all transactions preceding it in S', and cannot see any updates from transactions following it in S'. This is equivalent to executing transactions serially according to S', so S is conflict-equivalent to S'. □

Appendix C GDCC EVALUATION

C.1 Comparisons with other Concurrency Control Protocols

System evaluation results clearly show that Bonspiel's GDCC effectively reduces tail latency while maintaining high performance in SR transactions, average latency, and system throughput. Now, we show that this success is due to the meticulous design of GDCC, whereas adaptations of related concurrency control schemes in the literature do not yield comparable results.

First, we include Polaris [89] in our study. Polaris is an optimistic concurrency control (OCC) scheme that supports transaction priority but is designed for single-node databases. In our adaptation, we assign higher priority to MR transactions over SR transactions in Polaris, same as GDCC. Next, we include Sundial [92], a state-of-the-art distributed concurrency control protocol designed for distributed transactions across multiple servers within the same data center. We have adapted it to function in a geo-distributed and replicated setting. Finally, we also include three classical two-phase locking (2PL) schemes: WOUND-WAIT (WW), NO-WAIT (NW), and WAIT-DIE (WD). WW and WD utilize transaction timestamps to resolve conflicts – MR transactions naturally possess higher priority due to their long-running nature with older timestamps.

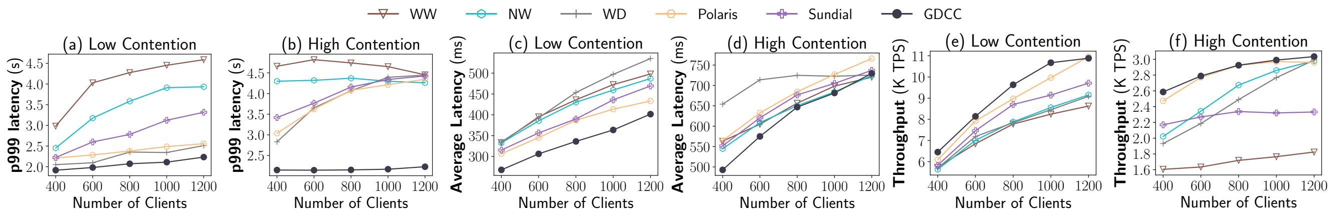

Figure 16: Performance of GDCC compared to state-of-the-art concurrency control protocols (TPC-C).

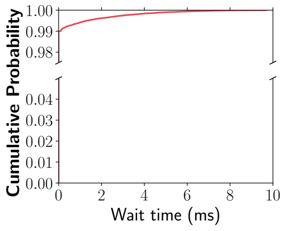

Figure 17: Wait time CDF of GDCC (TPC-C).

To ensure fair comparisons, all schemes use Raft [63] as the replication protocol, just like Bonspiel. They also employ the same state-of-the-art geo-distributed atomic commit protocol as Bonspiel, R4 [41]. All schemes utilize only `read-leader` (RL) for remote access, and we disable Bonspiel's GAMS to ensure it also uses only RL for remote access in this experiment.

Figure 16 presents the results. GDCC achieves better tail latency compared to other adaptations of existing concurrency control schemes, particularly in high contention settings. WAIT-DIE and Polaris perform well in low contention scenarios because they also ensure that MR transactions are abort-free with respect to SR transactions. However, since they are not MR Wait-Minimal, there remains a noticeable gap in tail latency between them and Bonspiel. Furthermore, they become less competitive as contention levels increase. WOUND-WAIT, despite also being abort-free, performs poorly. This is because wounding a transaction that holds locks is especially wasteful in a geo-distributed setting.

C.2 MR Wait-Minimal of GDCC

We also conducted an additional experiment to examine the time of MR transactions waiting for SR transactions to release their locks. Figure 17 displays the results. We found that 99% of MR transactions are indeed wait-free, while the remaining 1% wait for only a very short duration, less than 8 ms. This corroborates the MR Wait-Minimal property of GDCC.

C.3 Ablating GDCC

From previous discussions, we understand that GDCC outperforms state-of-the-art concurrency control protocols, including Sundial [92] and Polaris [89], by achieving significantly better tail latency while maintaining strong average latency and throughput. However, it remains unclear which component of GDCC contributes to this performance. Specifically, compared to vanilla optimistic concurrency control, GDCC features three major components:

(1) A priority scheme between MR and SR transactions.
(2) A conditional waiting reserve-lock conflict resolution scheme for MR transactions.
(3) Early visible write (EVW).

Now, we further ablate these three components to investigate the source of GDCC's performance improvement. In these experiments, GAMS is always disabled to allow us to focus on the effectiveness of these components within GDCC. Additionally, since both components (2) and (3) rely on (1), it is unnecessary to investigate their individual effectiveness independently.

Figure 18 presents the ablation study of GDCC. Specifically, the `w/o all` configuration indicates that none of the components in GDCC are activated, effectively yielding the performance of vanilla OCC. Under this setup, MR transactions can be aborted by SR transactions and may need to wait for SR lock holders during validation, meaning they are neither abort-free nor wait-minimal. The `prio only` configuration activates only the priority scheme. Consequently, MR transactions will not aborted by SR ones, but they can still be forced to wait for them. Thus, MR transactions are abort-free but **not** wait-minimal. The `prio&cond wait` configuration enables both the priority scheme and the conditional waiting reserve-lock conflict resolution scheme. In this case, MR transactions are wait-minimal. However, they are **not** abort-free because they might read stale data and be aborted eventually. Finally, the `prio&EVW` configuration activates both the priority scheme and EVW. This setup is essentially equivalent to `prio only`, as enabling EVW without conditional waiting does not introduce any differences. For reference, we also include the performance of full GDCC, where MR transactions are both abort-free and wait-minimal. Note that in all setups, SR transactions are wound-free, and the abort-free and wait-minimal principles for MR transactions apply only with respect to SR transactions.

As illustrated in the figure, the priority scheme significantly reduces tail latency under low contention, highlighting the importance of the MR Abort-Free principle. However, surprisingly, under high contention, the priority scheme appears to worsen tail latency. This is because the vanilla OCC (`w/o all`) results in fewer MR transactions being committed under high contention, and the p999 latency may capture the latency of fast SR transactions or the latency of MR transactions that encounter few conflicts, a problem discussed earlier in the paper. Building on the priority scheme, conditional waiting makes MR transactions wait-minimal, albeit at the cost of compromising the MR Abort-Free principle. This yields slightly better performance than vanilla OCC, which offers neither MR Abort-Free nor MR Wait-Minimal principles. The `prio&EVW` configuration is equivalent to `prio` alone, as EVW merely allows SR

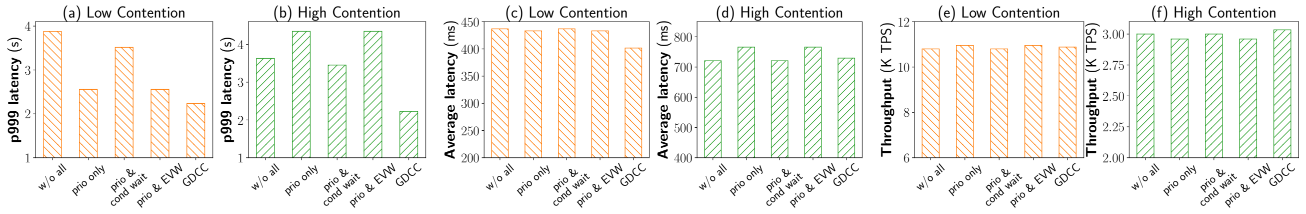

Figure 18: GDCC ablation study (TPC-C).

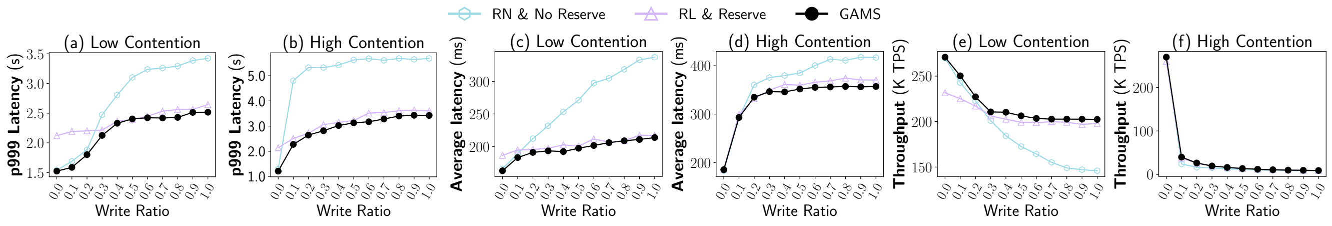

Figure 19: Effectiveness of GAMS (YCSB).

transactions to publish their dirty writes early, but no one is permitted to read them, resulting in no discernible difference. Moreover, this figure clearly demonstrates the strong synergy between these components. When combined, the full GDCC exhibits significantly better tail latency than any other setups, attributed to the fact that MR transactions in GDCC are both abort-free and wait-minimal.

As the components in GDCC are primarily designed to improve MR transactions' performance, which constitute a minority of the system, the differences between these configurations in terms of average latency and throughput are largely negligible.

Appendix D GAMS EVALUATION

D.1 Effectiveness of GMAS

We evaluate the effectiveness of Bonspiel's GAMS and compare it with always reading the nearest replica (RN) without reservation, and always reading the leader (RL) with reservation. We vary the ratio of write operations in the workload to determine whether GAMS can consistently select the best-performing access method based on the update frequency of records. Since TPC-C fixes this parameter, we resort to the YCSB workload. Figure 19 presents the results. The figure illustrates that always using RN without reservation yields better latency and throughput at low write ratios, whereas always using RL with reservation excels at high write ratios. GAMS is able to select the best-performing access method in all cases, and at times, it can outperform both competitors by employing different access methods for different records within the same transaction.

D.2 GAMS under Unstable Networks

We evaluate the performance of GAMS under unstable networks. To realistically simulate network instability, we follow the methodology of [33, 41] by introducing a packet loss rate of 2%, a level commonly observed in congested data centers and emulated using the Linux `tc` command. Additionally, we model random network delays using a Log-Normal distribution, as suggested in [57, 95], to capture the variability typical of real-world network latency.

Figure 20 summarizes the results. As illustrated in the figure, despite these challenging conditions, GAMS consistently outperforms both `RN&No-Reserve` and `RL&Reserve`, demonstrating its robustness and effectiveness in unstable network environments.

D.3 Determining the Priority Boosting Threshold

Figure 21 presents the results of our empirical study to determine the optimal threshold parameter a (number of aborts) for increasing priority (Section 3.1.5). As shown, setting $a = 2$ consistently achieves the best tail latency while maintaining strong average latency and throughput under both low and high contention. Therefore, we set a to 2.

Appendix E MORE TPC-C RESULTS

E.1 Under Partially-Replicated Setups

In this section, we present evaluation results based on a 3P2R setup, there are three partitions (regions), each replicated twice. Each region hosts the leader replica of the local partition, as well as the follower replica of a remote partition, as illustrated by Figure 22. The setup is deployed as two network configurations, one with uniform round-trip latency (Table 8), one with non-uniform latency (Table 9).

Figure 23 and Figure 24 illustrate the overall performance of Bonspiel and others on those setups. We observe that Bonspiel reduces the p999 latency significantly while maintaining top-tier average latency and system throughput, consistent with the 5P5R full replication results presented in the main body of the paper.

E.2 Varying Numbers of Partitions

In this section, we further evaluate the performance of Bonspiel using the TPC-C workload, by varying the number of partitions

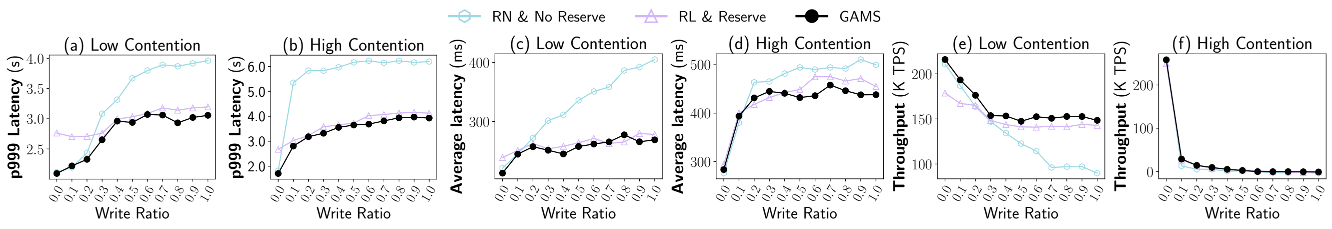
Figure 20: GAMS evaluation under unstable networks (YCSB).

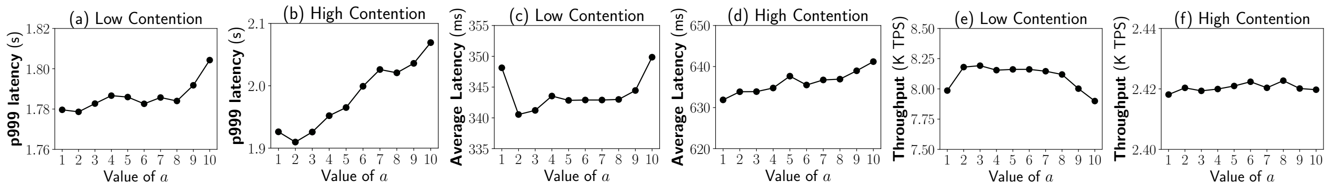
Figure 21: Determining the priority boosting threshold (TPC-C).

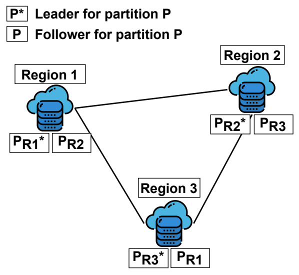

Figure 22: Partially-replicated setup; two-way replicated with three partitions.

Table 8: Partially-replicated setup with *uniform* round-trip latency; two-way-replicated with three partitions.

	Region 2	Region 3
Region 1	100	100
Region 2	-	100

Table 9: Partially-replicated setup with *non-uniform* round-trip latency; two-way-replicated with three partitions.

	Region 2	Region 3
Region 1	0.2	100
Region 2	-	100

in the network. Specifically, we adjust the number of partitions from 2 to 5. In our experiments, each partition represents data from a specific region, holds a fixed number of warehouses, and is associated with a set of local clients. All partitions operate in full replication mode – each region maintains a full replica of all partitions. Thus, the network setups range from 2P2R to 5P5R, with the total number of nodes used in the experiments increasing from 4 (2P2R) to 25 (5P5R). As we increase the number of partitions, both the system load and capacity expand as well. In all experiments, the WAN round-trip latency between any two regions is fixed at 100 ms.

Figure 25 presents the results, with all metrics collected at the point when the system saturates. As more partitions are added across regions, the overhead increases, leading to a rise in the average latency of all systems. Throughput also increases, as both the system load and capacity grow with the number of partitions. As demonstrated throughout the paper, tail latency is primarily influenced by the abort rate and levels of contention, with the atomic commit time being secondary. Since the workload remains unchanged in this experiment, the abort rate and contention levels stay stable across different numbers of partitions. Consequently, the tail latency of Bonspiel remains low and consistently outperforms the others, while also maintaining top-tier performance in terms of average latency and system throughput.

Appendix F MORE YCSB-A RESULTS

This appendix presents supplementary experimental results on the YCSB-A workload. Specifically, we set the ratio of MR transactions to 10%, with the remaining 90% being SR transactions.

F.1 System-Level Evaluation

Figure 26 illustrates the performance of Bonspiel in comparison to other state-of-the-art geo-distributed databases. We observe that the results align with our discussion based on the TPC-C results: Bonspiel significantly reduces tail latency while maintaining top-notch performance in terms of average latency and system throughput.

F.2 Other Tail Latency Metric

We report Bonspiel's performance using additional tail latency metrics – p50, p90, p99, p999, p9999 – as shown in Figure 27. The findings are consistent with the TPC-C experiments: Bonspiel consistently delivers the lowest tail latency across varying tail percentages.

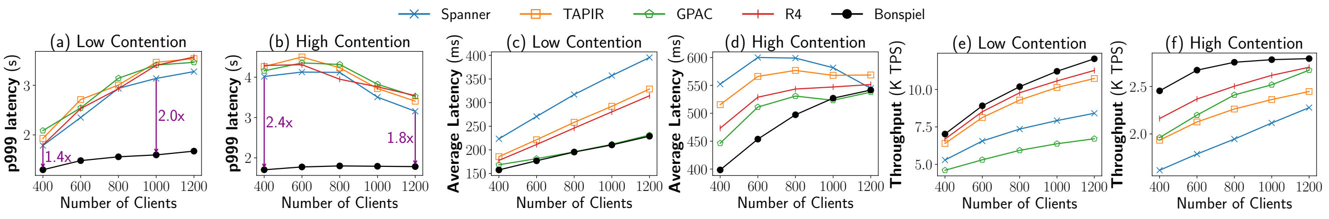
Figure 23: Partially-replicated setup (3P2R) with uniform round-trip latency (TPC-C).

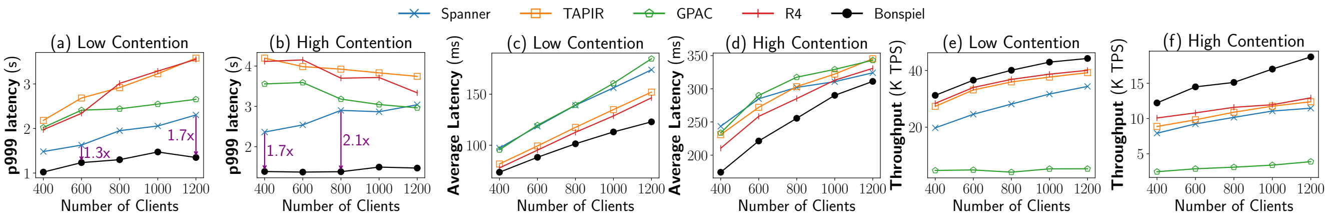
Figure 24: Partially-replicated setup (3P2R) with non-uniform round-trip latency (TPC-C).

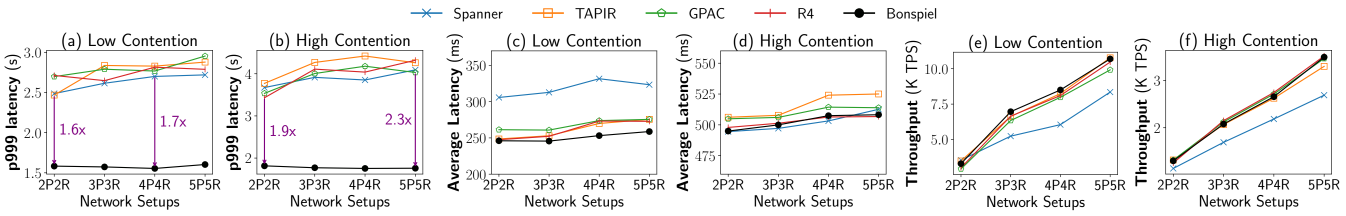
Figure 25: Varying numbers of partitions under full replication (TPC-C).

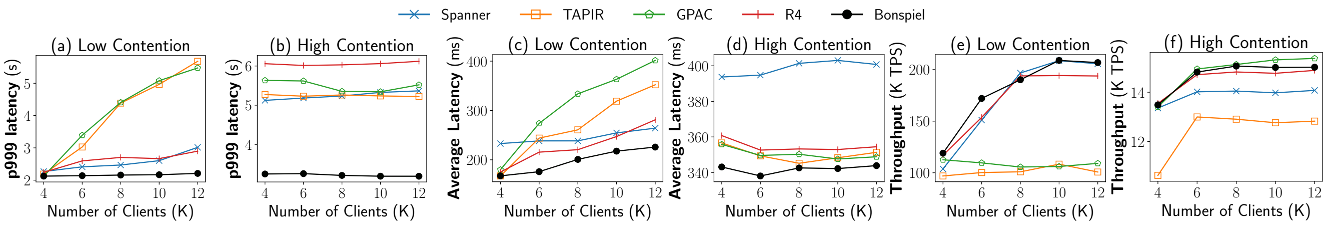
Figure 26: Overall performance (YCSB-A).

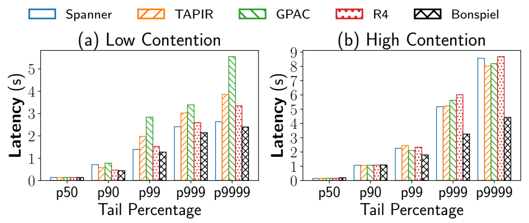
Figure 27: Various tail latency results (YCSB-A).