Appendix of Warbler

A Formal proof of correctness

This is an extended version of the Warbler, containing comprehensive proof of its strict serializability.

Definition of dependency: If transaction T_0 writes to a key and subsequently transaction T_1 reads from the same key, we refer to this as T_1 being read-dependent on T_0. In Warbler, read dependency matters for correctness and the later replay. Throughout this proof, the term "dependency" specifically refers to read-dependency, while "conflict" refers to "Write-Write," "Read-Write," and "Write-Read" conflicts, which are resolved in the speculative execution phase.

Lemma 1. Distributed OCC can ensure serializability in the absence of failures.

Proof. Warbler's distributed OCC protocol shares similarities with widely used OCC and 2PC protocols in system designs, and its correctness is straightforward; hence, we do not elaborate on this lemma. The core principle of our distributed OCC protocol is that a transaction is considered to have reached a serializable point if it can satisfy two key conditions during the commit phase, consistent with other OCC protocols: (1) All write locks are successfully acquired, and (2) No read items have been modified by other concurrent transactions. Vector clocks of transactions would not be used in this execution.

Lemma 2. The serialization point of a transaction always occurs between the start of its execution and the completion being returned to the client.

Proof. The serialization point occurs after all locks of a transaction have been acquired in the commit phase. The shard leader delays the release of the result to clients until the replication and progress-checking of transactions have been completed. This mechanism guarantees that the serialization point falls within the specified window, maintaining the correctness of the stated lemma.

Lemma 3. If transaction T_1 depends on T_0, then the vector clock of T_1 is pairwise larger than that of T_0.

Proof. According to Lemma 1, if two transactions conflict, they cannot both enter the system's serialization point and will be scheduled to execute sequentially. If the transaction T_1 depends on T_0, in *GetClock* phase, the vector clock of T_0 must be included in T_1's ReadSet. This allows the Warbler to determine T_1's vector clock by selecting the maximum of shard clocks among all vector clocks in its ReadSet. In addition, local clocks of shards involved in the WriteSet are incremented (via piggybacking during the *Install* phase) to ensure that all subsequent transactions after T_1 observe a monotonically increasing local clock. Therefore, T_1 is pairwise larger than T_0.

Formally, let the vector clock for T_0 be denoted as (p_0, p_1, ..., p_{n-1}), and for T_1 as (q_0, q_1, ..., q_{n-1}), where n represents the number of shards. We have $p_i \leq q_i$ where $i \in [0, n-1]$. Notably, the purpose of the vector clock is to support correct replay and recovery. If two transactions are read-only, their vector clocks may be identical in Warbler, making it impossible to distinguish them based solely on their vector clocks. However, this does not pose an issue, as read-only transactions do not alter the database state and, therefore, do not affect replay or state recovery during failures.

Lemma 4. If transaction T_1 depends on transaction T_0, and T_2 depends on T_1, then T_2 *transitively* depends on T_0; the vector clock of T_2 is larger than that of T_0 pairwise.

Proof. According to Lemma 3, if T_1 depends on transaction T_0, then T_1 is pairwise larger than T_0; if T_2 depends on T_1, then T_2 is pairwise larger than T_1; so we can conclude that T_2 is also pairwise larger than T_0 transitively.

Lemma 5. For a Paxos stream on shard leader-i, the shard clock-i of Paxos entries within this Paxos stream is monotonically increasing within the same epoch.

Proof. According to Lemma 1, if two transactions, T_0 and T_1, conflict during the speculative transaction execution, they must contend for a specific key from shard leader-i. Consequently, T_0 and T_1 will be scheduled sequentially at the serialization point, as guaranteed by our distributed OCC. As a result, the shard clock is obtained sequentially. The transaction logs for T_0 and T_1 on shard leader-i will also be generated sequentially, with a monotonically increasing shard clock. It is crucial to note that our worker threads operate in a blocking manner, progressing to the next transaction only after the previous transaction within the same worker thread has been speculatively executed and its log has been generated.

In our implementation, multiple helper threads are designated to wait for RPCs from other shard leaders. These threads do not have their own dedicated Paxos streams. Instead, a worker thread and several helper threads will share a same Paxos stream. When a worker thread decides to encode a batch of transactions into a new Paxos log entry, it will wait for the ongoing transactions in its corresponding helper threads to complete to avoid violating this lemma. If there is a shard failure, the new leader will always execute transactions in the new epoch using its own local monotonic clock. Combined with the guarantees of Lemma 3, we have successfully proved this lemma.

Lamma 6. For each shard leader/follower, w_i in its local view of the vector watermark (w_0, w_1, ..., w_{n-1}) consistently increments to represent the replication progress of shard leader-i.

Proof. According to Lemma 5, the shard clock-i within the same Paxos stream consistently increases on shard-i. Combining this with the fact that w_i is picking up the *minimum* among all shard clock-i values from the vector clocks of *replicated* transactions across all Paxos streams on shard-i, we can infer the following. (1) The replication progress (w_i) of shard-i is advancing, and (2) If a transaction updates at least one key on shard-i and its shard clock-i smaller than w_i, the piece of this transaction has been successfully replicated by one Paxos stream on the shard-i.

Lemma 7. The final vector watermark (FVW) will eventually become identical and constant across all shard leaders for the old epoch e in the presence of failures.

Proof. FVW is computed during a failure recovery, but shard failures introduces more complexity. When a healthy shard-i increments its epoch number from the previous epoch e to $e+1$ during failures. First, it rejects all transactions from the old epoch e. This ensures that no new transactions are executed for the outdated epoch e during recovery on this healthy shard.

Second, new transactions executed in the new epoch defer reading potential rollback versions from the previous epoch, relying on the local view of the vector watermark until the FVW is received. For ongoing transactions from the old epoch, those without associations to failed shards will proceed as usual and finish very quickly. However, a critical situation arises if a shard failure occurs, and healthy shards are unaware of this fact yet: if a worker thread on a healthy shard attempts to perform an *Install* RPC to the failed shard or waits for an *Install* RPC from it, the thread will block further transaction execution until the new epoch RPC arrives (though rare). This precaution ensures that transaction replication is consistent to avoid a corner case, in which the vector clock of a transaction is below FVW, but some pieces of this transaction are lost.

The FVW computation is deferred until all shards have properly closed epoch e. Even if a new failure occurs during recovery, it will not affect correctness because no FVW guarantee has been made to other shards. This allows the FVW computation for epoch e to safely restart if needed.

By enforcing these strategies, Warbler guarantees that the FVW calculation remains consistent and deterministic.

Lemma 8. Within an old epoch e, the FVW is the max vector watermark.

Proof. By Lemma 6 and Lemma 7, we know that (1) the vector watermark consistently advances; (2) when all Paxos streams on a healthy shard leader-i are closed, it ensures that all *completed* speculative transactions have been fully replicated, marked by the use of "INF" in no-ops. For failed shards, no transactions are executed before the FVW is computed. These collectively ensure that the FVW represents the maximum vector watermark for epoch e.

Lemma 9. If one piece of a transaction T_0 is replayed on one shard follower, then 1) all other pieces of this transaction will be replayed on all relevant shard followers; 2) all transactions that T_0 depends on will also be replayed on all relevant shard followers.

Proof. If one piece of a transaction T_0 is replayed on one shard follower, it indicates that its vector clock is below the advancing vector watermark (or FVW in the case of failure) pairwise. Since all pieces of T_0 share the same vector clock, this applies to all its pieces. We can conclude this lemma by combining Lemma 6 and 7, which states that w_i represents that the pieces of a transaction on shard-i that are below w_i have been successfully replicated. Additionally, when combined with Lemma 3, we observe that all transactions on which transaction T_0 depends on are also pairwise below the advancing vector watermark. Consequently, all those dependent transactions will be replayed on all relevant shards.

It is possible for a transaction to be partially below the FVW. For instance, a transaction might have been speculatively executed but only partially replicated successfully on some shards. In such cases, Warbler will prevent any pieces of the transaction from being replayed. The reason is that there is no certainty that the transaction will be fully replicated across all relevant shards during a failure.

Lemma 10. A transaction executed in the new epoch $e+1$ does not depend on any transactions that are potentially rolled back in the old epoch e.

Proof. In most cases, it is straightforward that a transaction cannot depend on any potentially rolled-back transactions because the coordinator already knows the (e,FVW). Consequently, it can always read the latest committed version below the (e,FVW) if necessary.

However, this is not always the case. During a shard failure, a healthy shard is not blocked, and continues executing transactions in the new epoch. It may find itself executing a transaction without yet knowing the exact value of (e,FVW). At this moment, the health shard lacks certainty about which version of values to read, as described in Section 5.1. Instead, it relies on its local view of the vector watermark, which might be somewhat outdated but is safe to use because it is guaranteed to be pairwise smaller than (e,FVW).

According to Lemma 8, if the latest version in the old epoch e is below the local vector watermark, it will certainly be below the future computed FVW (since the FVW represents the maximum in the old epoch e). In such cases, there is no risk in reading it immediately. However, if the version a transaction in the new old reads from the old epoch is not below the local vector watermark, this transaction is enqueued for future retry.

Lemma 11. Rollback only needs to be handled within the same epoch; there is no need for cascading aborts processing across epochs.

Proof. According to Lemma 10, a transaction in the new epoch $e+1$ never observes any states released by rolled-back transactions from the previous epoch e. Consequently, we can easily conclude the validity of this lemma.

Lemma 12. If a transaction is rolled back in one shard leader, it will be rolled back in all relevant shard leaders as well.

Proof. If transaction T_0 is rolled back in one shard leader, it implies that its vector clock is not pairwise below the tuple (e, FVW). According to Lemma 7, which asserts that the tuple (e,FVW) is identical across different shard leaders, we can conclude that the other pieces of T_0 on other shard leaders must also not be pairwise below the tuple (e,FVW) and therefore need to be rolled back as well.

Lemma 13. All transactions below the (e,FVW) *never* depend on any transactions beyond or incomparable with (e,FVW).

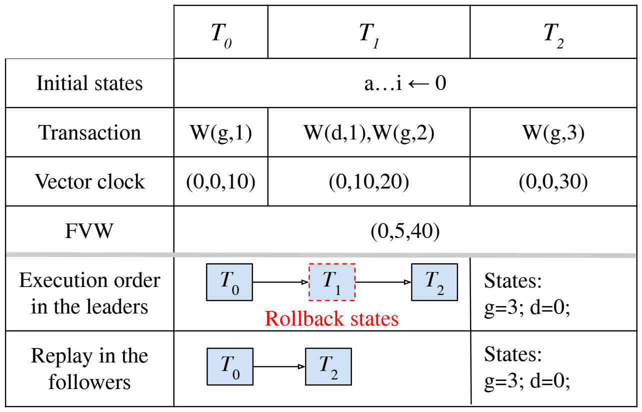
Proof of contradiction. Suppose there exists a transaction T_1 below the (e,FVW) that depends on another transaction T_0 beyond or incomparable with (e,FVW). Intuitively, if a transaction T_1 with vector clock (p_0, p_1, ..., p_n-1) is below the latest vector watermark (w_0, w_1, ..., w_n-1), then $p_i <= w_i$ for all i in $[0, n-1]$. By Lemma 3, if T_1 depends on transaction T_0 with vector clock (q_0, q_1, ..., q_n-1), then $p_i >= q_i$ for all i in $[0, n-1]$. Therefore, we can conclude that $q_i <= w_i$ for all i in $[0, n-1]$. However, T_0 is incomparable or beyond (e,FVW); it implies that there must exist an i that $q_i > w_i$. This yields a contradiction, so T_1 cannot exist. Therefore, the Lemma is proved.

Lemma 14. A transaction T_0 in the epoch e never depends on any transaction T_1 from a higher epoch number (e.g., e+1).

Proof. Similar to Lemma 13, if transaction T_1 is executed in the shard leader-i with a new epoch $e+1$. At this moment, any read from transaction T_0 with the old epoch e will be rejected. Combining with the fact from Lemma 10, we can conclude this lemma.

Lemma 15. Replay on the shard followers obeys the execution order on the shard leaders and is always consistent with the shard leaders.

Proof. Replay threads on the shard followers replay Paxos logs concurrently under the control of the vector watermark using Thomas write rule. If failures occur, shard followers will replay transactions up to the (e,FVW), while the shard leader will roll back transactions above or incomparable with (e,FVW). According to Lemma 11, Lemma 13, and Lemma 14, the (e, FVW) represents a globally consistent max cut. This, combined with Lemma 9 and Lemma 12, guarantees that the rollbacks performed by the shard leader and the replays executed by the shard followers will always result in the same final state.



	T_0	T_1	T_2
Initial states	a…i ← 0		
Transaction	W(g,1)	W(d,1),W(g,2)	W(g,3)
Vector clock	(0,0,10)	(0,10,20)	(0,0,30)
FVW	(0,5,40)		
Execution order in the leaders	$T_0 \rightarrow T_1 \rightarrow T_2$ Rollback states		States: g=3; d=0;
Replay in the followers	$T_0 \rightarrow T_2$		States: g=3; d=0;

Figure 13: Incomparable transactions example: T_1 and T_2 are incomparable in pairwise while T_2 has a blind write on key g. These transactions exhibit no dependency on each other.

Case study: As mentioned above, it's not necessary for Warbler to always guarantee the pairwise comparison property due to conflicts (e.g., a blind write-write conflict), but it does not hurt correctness. Assume we have three transactions, T_0-T_2, as in Figure 13. The keys are distributed across the shards as follows: shard-0 contains keys (a, b, c), shard-1 contains keys (d, e, f), and shard-2 contains keys (g, h, i). Initially, all key values are 0. In this example, the vector clocks of T_1 and T_2 are incomparable, and the possible FVW is (0,5,40) if there is a failure on shard leader S_0. The followers will replay T_0 and T_2, and ignore T_1. Meanwhile, the healthy shard leaders S_1 and S_2 have to roll back the effects made by T_1. The final states on the shard leader and follower are still consistent.

A question was raised: is it really safe to only roll back T_1, not all transactions (e.g., T_2) executed after it in the leaders? The first thing we should be aware of is that if T_2 is executed after T_1 and we only roll back T_1, then the vector clocks of T_1 and T_2 must not be pairwise comparable. It is safe to roll back T_1 because (1) T_1 has not sent an acknowledgment back to the client, and (2) transactions executed after T_1 in the serializable order (e.g., T_2) do not observe any states released by T_1 (or transitively), otherwise T_1 and T_2 would be pairwise comparable.

As discussed above (Lemma 1, Lemma 2, and Lemma 15), we can conclude that Warbler guarantees strict serializability.

B Pseudocode: Lifecycle of a Transaction Execution with Compression Optimization

```
// Execute a transaction by a worker thread from one
    shard server.
// Omit the logic for the failure recovery.

/**** Variable definitions **********/
int nShards;  // Number of shards
int nThreads; // Worker threads per shard

// A counter using fetch and add
int localCounter;
// local shard leader index
int locShardID = getShardID();
int localW; // Shard watermark
// Replication progress per worker thread
vector<int> replicateProgress(nThreads);

// Size of compressed vector clock, e.g., 10 fixed
int compSize;
// local view of compressed vector watermarks
vector<int> compVW(compSize);

// k-v store in map for simplicity (masstree)
map<int,ValueType> masstree;


// Get compressed shard clock index according to a
    defined mapping
int GetCompSIdBySId(int shardId) {
  // We assume a simple mapping here
  return shardId/(nShards / compSize);
}


/**** Execute a transaction in Warbler ****/

// Execution phase
{
  // Read() optimistically, and buffer write
  // ...
}

// Commit phase
{
  {
    // Lock keys in writeset
    // ...
  }

  vector<int> compVC = getClock();

  {
    // Validate keys in readset
    // ...
  }

  int threadID = getThreadID();
  // Replicate transaction logs in batch
  asyncPaxosRep(/*resultant values of transaction*/,
                /*compVC*/, /*threadID*/,
                /*locShardID*/,
                callbackAsyncPaxosRep);

  // Install & Release locks
  Install(writeSet, currentEpoch, compVC);
}

void Install(writeSet, currentEpoch, compVC) {
  for (auto k,v: writeSet) {
    {// NOTE: Execute this code block on all
        associated shards !!!
      masstree[K].append((currentEpoch, compVC, v));
      compSId = GetCompSIdBySId(locShardID);
      // Ensure that all following transactions
          observe a larger local counter
      int delta = compVC[compSId] - localCounter;
      if (delta > 0)
        localCounter.fetch_add(delta);
    }
  }
}

// NOTE: this func can be executed in different
    associated shards
tuple<int,vector<int>,ValueType> Read(k, currentEpoch)
    {
  // Txn always reads latest value if no failures,
      otherwise uses FVW to read value from the old
      epoch
  _epoch, _compVC, _value = masstree[k];
  return (_epoch, _compVC, _value);
}


// Get compressed vector clock for the transaction
vector<int> getClock() {
  vector<int> compVC(compSize);

  // Merge compressed vector clocks in ReadSet
  for ((_, _cvc): readSet) {
    for (int i=0; i<compSize; i++) {
      compVC[i] = max(compVC[i], _cvc[i])
    }
  }

  // Broadcast a remoteGetClock RPC to remote shards
      in WriteSet to increment remote shard's local
      counter
  for (auto shardId: writeSet) {
    compSId = GetCompSIdBySId(shardId);
    sclock = remoteGetClock(shardId);
    compVC[compSId]= max(compVC[compSId], sclock);
  }

  compSId = GetCompSIdBySId(locShardID);
  localCounter.fetch_add(1);
  compVC[compSId] = max(compVC[compSId],localCounter);
  return compVC;
}

// If a transaction log is replicated to a majority,
// then invokes this callback function on shard
    followers / leaders.
void callbackAsyncPaxosRep(int threadID, vector<int>
    compVC, int locShardID, const string&
    resultantValue) {
  compSId = GetCompSIdBySId(locShardID);
  replicateProgress[threadID] = compVC[compSId];
  localW = min(replicateProgress);
  compVW[compSId] = min(compVW[compSId], localW);

  {
    // Exchange shard watermark periodically in
        background to update compVW
    // ...
  }

  receivedQueues.push_back((resultantValue, compVC));

  // Replay on shard followers
  while (!receivedQueues.empty()) {
    for (int i=0; i<compSize; i++) {
      if (receivedQueues.front().compVC[i] <= compVW[i
          ]) { }
      else { break; }
    }

    // It is safe to return back the client
    replay(receivedQueues.front());
    receivedQueues.pop_front();
  }
}
```

C Shard management in detail

This section provides a detailed explanation of our shard management approach. To enable operations such as shard deletion, addition, and key partition reassignment, we use a management configuration that encapsulates crucial details, such as whether keys remain on the same server (compared to the previous configuration), the total number of shards in the new setup, and the identifiers of those shards.

Two key observations shape our approach: (1) cross-epoch vector clock and vector watermark computations and comparisons are unnecessary in Warbler, and (2) transactions in the old epoch are not dependent on those in the new epoch. Once Warbler obtains the FVW for the old epoch, the entire database can be treated as a read-only snapshot, serving as the initial state with all vector clocks reset to zero (vector clocks from higher epochs have a higher priority). This still ensures that dependency relationships are preserved, as vector clocks only track dependencies, and vector watermarks track replication progress to prevent data loss.

Building on these observations, our solution treats shard management similarly to failure recovery. The Configuration Manager (CM) oversees the management process by broadcasting a *MIGRATE-START* message containing the new configuration to all shards, including those being deleted or newly added. Upon receiving this message, shard leaders transition to the new epoch and reject any RPCs from the old epoch or from invalid shards (e.g., those deleted in the new configuration).

Shard leaders then begin executing most transactions speculatively, with two constraints: (1) keys must be below their current view of vector watermarks to avoid potential data loss, and (2) keys scheduled for migration cannot be accessed to ensure safety in the event of a shard failure during the shard management. Since there is no need for all vector clocks from the old epoch to compare with new epoch vector clocks, eliminating concerns about mismatches between vector clocks across epochs.

In the background, shards in the old configuration close the old epoch and exchange shard watermarks to compute a FVW. This process removes the first constraint, and the computed FVW is persisted in the CM for future retrieval, eliminating the need for independent recomputation.

Simultaneously, each shard migrates key-value pairs to other configured shards while retaining the original copies. Once a shard completes its migration task, it reports back to the CM. A background thread reclaims the original copies only *after* the migration tasks for all shards are complete. Shards marked for deletion are only responsible for closing Paxos streams in the old epoch.

When *all* migration tasks are completed, the CM issues a *MIGRATE-END* RPC, lifting the second constraint and enabling full transaction execution under the new configuration. In the event of a shard failure during the migration procedure, the CM initiates a failure recovery procedure, assigning a new epoch. After recovery, the CM can safely restart migration procedure from the beginning.

D Implementations

Multi-version and lazy memory de-allocation. As described in the paper, Warbler stores multiple versions of each key to support rollbacks in shard failures. This is implemented as a linked list. Each version contains a pointer to the previous version on this key. The key index stores the pointer to the latest version. We choose this implementation for the multi-version to enable more efficient rollbacks.

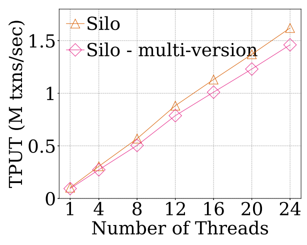
When rolling back versions because of shard failures, Warbler will remove the version from the linked list. As the rollback often needs to discard all versions above a *FVW* in the old epoch, Warbler only needs to unlink a single pointer in the middle of the linked list, minimizing the cost in such a case. Because memory (de)allocation is a very costly operation, Warbler does not immediately free the memory of the discarded versions, but defer it to the future, usually the next time the key is accessed. In our experience, we find this is an engineering effort that can greatly reduce the rollback time.

Event-driven helper threads. In Warbler, the limitation of available CPUs makes it impractical to maintain a large number of busy-polling server threads simultaneously for receiving DPDK messages. As a solution, Warbler adopts an event-driven strategy for managing *helper threads* responsible for processing RPC requests from other shard leaders, thereby saving CPU resources. This approach entails temporarily suspending threads and resuming them when required, which regrettably results in additional latency. Despite this trade-off, the event-driven method can efficiently allocate CPU resources and preserve the overall system performance. In our implementation, we use two polling server threads that continuously receive DPDK messages and delegate those messages to the corresponding threads for execution. Warbler's RPC relies on eRPC [44] and a private RPC library.

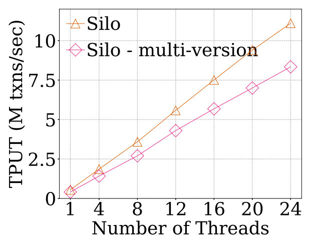
E Impact of the versioned values

In this experiment, we analyze the impact of the versioned values introduced in Warbler. We run the Silo instance by increasing the number of worker threads on a single shard, with and without the multi-version feature. Across all configurations, the performance scales well over threads. Silo is able to achieve up to 1.62M TPUT on TPC-C in Figure 14a and up to 11.1M TPUT on the Microbenchmark in Figure 14b, both with 24 threads. The versioned values introduce a reasonable overhead of 9.9% for the TPC-C benchmark. This overhead is primarily due to the additional memory copy operations.

Notably, the overhead of the versioned values is 24.9% on the microbenchmark, which is higher than in TPC-C. The



(a) TPC-C



(b) Microbenchmark

Figure 14: The impact of the introduced versioned values on TPC-C and Microbenchmark over threads.

reason for this difference is that the microbenchmark uses fixed-length values, while TPC-C uses variable-length values. On the microbenchmark (without versioned values), the value node only needs to be allocated once and then overwritten (*without new allocation*) by the following writes. However, Warbler always has to allocate new space for new writes for the versioned values, regardless of whether we are running TPC-C or microbenchmark. As a result, the microbenchmark suffers more throughput drop compared to the TPC-C benchmark.