Caracal: Contention Management with Deterministic Concurrency Control

Dai Qin
University of Toronto
mike@eecg.toronto.edu

Angela Demke Brown
University of Toronto
demke@cs.toronto.edu

Ashvin Goel
University of Toronto
ashvin@eecg.toronto.edu

Abstract

Deterministic databases offer several benefits: they ensure serializable execution while avoiding concurrency-control related aborts, and they scale well in distributed environments. Today, most deterministic database designs use partitioning to scale up and avoid contention. However, partitioning requires significant programmer effort, leads to poor performance under skewed workloads, and incurs unnecessary overheads in certain uncontended workloads.

We present the design of Caracal, a novel shared-memory, deterministic database that performs well under both skew and contention. Our deterministic scheme batches transactions in epochs and executes the transactions in an epoch in a predetermined order. Our scheme enables reducing contention by batching concurrency control operations. It also allows analyzing the transactions in the epoch to determine contended keys accurately. Certain transactions can then be split into independent contended and uncontended pieces and run deterministically and in parallel, further reducing contention. Based on these ideas, we present two novel optimizations, batch append and split-on-demand, for managing contention. With these optimizations, Caracal scales well and outperforms existing deterministic schemes in most workloads by 1.9x to 9.7x.

CCS Concepts: • **Software and its engineering → Concurrency control**; • **Information systems → Main memory engines**.

Keywords: deterministic concurrency control, contention, main-memory databases

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483591

1 Introduction

Deterministic databases are attractive for several reasons. Their predetermined ordering of transactions ensures serializable execution while avoiding concurrency-control related deadlocks and aborts. The lack of aborts ensures opacity [19] because transactions always read consistent data. Furthermore, deterministic execution reduces the need for two-phase commit, helping scale distributed transaction throughput [21]. It also simplifies replication and failure recovery since transactions can be replayed deterministically [17].

Unfortunately, current deterministic databases scale poorly under skewed and contended workloads. Such workloads are common because modern web applications frequently encounter unpredictable demand spikes. These applications rely critically on high-performance databases but are subject to events such as holiday sales or social reviews that make specific data items highly popular, leading to skewed and contended accesses in the database.

One strategy for dealing with contended data accesses is to partition the workloads so that contended accesses are serialized on each core. Existing deterministic databases [5, 7, 8] employ this approach for scaling and for handling contended accesses. However, partitioning suffers from load imbalance under skewed accesses, which can make it difficult to partition the data evenly. As a result, transactions accessing the popular items can degrade throughput dramatically. Long-term load spikes and imbalance requires repartitioning data, which is both expensive and may require rewriting the application code to ensure locality under the new partitioning.

To manage skewed accesses, databases can employ a shared-memory architecture and distribute transactions across cores. This approach allows load balancing transaction execution among all cores, but its performance can degrade significantly with contended accesses. Recently, much recent work on main-memory databases has focused on managing contention in OCC-based systems [9, 12], but these schemes are designed for non-deterministic databases.

This work presents Caracal, a novel shared-memory (i.e., non-partitioned), deterministic database that performs well under both skewed and contended workloads. Transactions running on different cores in Caracal access shared data and thus concurrency control is required to ensure deterministic ordering. To achieve parallelism, Caracal batches transactions into epochs; similar to previous deterministic concurrency control schemes [7, 21], it initializes the concurrency

	Versioning	Initialization	Execution	Handle Skew?	Handle Contention?	Scaling Challenge
Calvin	Single	Single-Thread	Shared	Yes	No	Single threaded initialization. Contention during execution.
Bohm	Multi	Partitioned	Shared	No	No	Skewed accesses. Contention during execution.
Granola	Single	N/A	Partitioned	No	Yes	Skewed accesses.
PWV	Single	Partitioned	Partitioned	No	Yes	Skewed accesses.
Caracal	Multi	Shared	Shared	Yes	Yes	Contention during initialization and execution. Addressed by batch append and split-on-demand.

Table 1. Comparison with Existing Deterministic Schemes

control operations for the batch before transaction execution, allowing transactions to be executed concurrently while still maintaining deterministic output. This shared-memory approach enables maintaining load balance across cores, but it can lead to poor performance with contended workloads.

We propose two novel optimizations called *batch append* and *split-on-demand* for managing contention. Both these optimizations are enabled by deterministic execution. Batch append executes the concurrency control initialization operations in arbitrary order, which allows batching them and running them scalably. Split-on-demand detects contended transactions without depending on accurate historical data. Then it splits contended transactions into contended and uncontended pieces. These pieces are run independently and without requiring synchronization for conflicts or aborts, which reduces the serial component of contended execution.

We evaluate Caracal using YCSB and TPC-C-like benchmarks and compare it with three deterministic concurrency control schemes that use various partitioning methods, Granola [5], Bohm [7] and PWV [8]. Caracal's optimizations enable it to outperform these schemes for most workloads. For uniform uncontended YCSB, Caracal's throughput at 32 cores is 1.64× higher than the best-performing alternative. For skewed YCSB, Caracal's advantage increases to 2.7× with contention, and 9.7× without contention. For a contended single-warehouse TPC-C-like workload, Caracal outperforms the best alternative by up to 1.92×.

This work makes several contributions. Caracal is the first deterministic database that scales well under skewed and contended workloads. We provide a detailed comparison of our approach against previous deterministic schemes. We show that by leveraging determinism, our optimizations can provide significant benefits compared to these schemes.

The rest of the paper describes our approach in more detail. Section 2 provides background on existing deterministic concurrency control schemes. Section 3 presents the design of Caracal system. Section 4 describes the Caracal implementation. Section 5 evaluates Caracal by comparing its performance with existing deterministic schemes. Section 6 provides our conclusions.

2 Existing Deterministic Schemes

We provide background on existing deterministic concurrency control schemes to motivate the design of Caracal. In a deterministic database, the serial ordering of transactions is established *before* transactions are executed and the outcome of the database is consistent with serial execution in that order. Thus the database ensures that a transaction's outcome is uniquely determined by the database's initial state and an ordered set of known previous transactions. The simplest implementation of a deterministic database executes all transactions serially in the predefined order. This is clearly correct but unable to utilize the parallelism available on modern multi-core systems. To enable parallelism, various deterministic concurrency control schemes have been proposed for single and multi-versioned systems and for shared-memory and partitioned architectures. Table 1 summarizes the differences between these schemes and Caracal.

Calvin is an early, single-version, shared-memory deterministic database [21]. Calvin ensures deterministic execution by batching transactions and executing them in two phases, initialization and execution. The initialization phase establishes a lock order for each row that will be accessed by transactions in the batch, thereby ensuring that rows are accessed according to the predefined serial order in the transaction execution phase. Calvin uses a centralized lock manager during initialization, which can become a performance bottleneck for high-performance, main-memory databases.

Bohm [7] is a multi-versioned deterministic database that develops on Orthrus [18]. Orthrus [18] shows that performance degrades under contention in the shared-memory architecture due to processor state pollution and thus proposes decoupling the lock manager from transaction execution. The lock manager is partitioned but transactions are executed in shared memory. Similarly, Bohm partitions the keys that need to be initialized across cores, thus eliminating synchronization costs during initialization. These systems use a hybrid approach: a shared-nothing model for concurrency control and a shared-memory execution architecture. As a result, they suffer under skewed workloads during initialization and contention during execution.

Deterministic databases can scale by partitioning data across cores. Granola [5] is a single-versioned, partitioned database that uses a timestamp-based scheme to assign a serial order to deterministic transactions. Transactions are split into pieces based on the partitioning scheme, with each piece accessing a single partition, and run in parallel across cores (intra-transaction parallelism). Granola does not require batching transactions or any concurrency control initialization since each partition executes its pieces serially in the predefined serial order. This makes Granola especially suited for independent transactions, whose pieces do not have intra-transaction dependencies [5, 11].

The piece-wise visibility (PWV) scheme [8] takes advantage of deterministic execution to make transaction writes visible early, after the transaction commit status is determined rather than after the transaction finishes execution. PWV supports partitioned execution, similar to Granola, while adding support for intra-transaction cross-partition dependencies via rendezvous points. These data dependencies limit parallelism in Granola since a partition needs to wait until a piece's dependencies are satisfied. PWV enables concurrent transaction execution within a partition by building a dependency graph per partition during the initialization phase. This graph tracks conflicts among pieces within a partition, allowing non-conflicting pieces, whose data dependencies have been satisfied, to be scheduled.

Granola and PWV's shared-nothing (partitioned) architecture provides good performance for contended workloads since data accesses do not need to be synchronized [2]. However, this architecture leads to load imbalance and poor scaling with skewed or hard-to-partition workloads. Also, programming with a shared-nothing database is non-trivial because developers need to partition the data, handle data-flow dependencies, and possibly repartition data to fully exploit the ever-increasing number of cores on modern systems.

Caracal is a multi-versioned, shared-memory database that can easily load balance skewed workloads. However, the shared initialization and execution phases can lead to performance degradation under contention. As discussed in the next section, our insight is that batching and determinism enable reducing contention because together they allow parallelizing operations with low overhead.

3 The Caracal Design

We now describe our transaction model and our shared-memory, deterministic concurrency control scheme in the Caracal database, and then present our optimizations for managing skewed and contended workloads.

3.1 Transaction Model

Our design requires batching transactions, which increases transaction latency. In practice, many web requests are issued by end users over the wide-area internet or mobile networks. Our approach targets these use cases and provides latencies comparable to wide-area network latencies.

Caracal assumes a one-shot transaction model in which transaction inputs are available at the start of a transaction. Thus requests do not require any further client input, which avoids latencies due to client communication. This model is commonly used in main-memory databases [9, 15, 20, 22].

Our approach requires the write-set keys of transactions before transaction execution, but unlike most existing deterministic designs [8, 21], we do not require the transaction's read-set keys and we support range updates. We rely on manual effort to infer the write set, which is feasible for many applications. Handling range updates is more difficult, but this support in Caracal makes it much easier to program transactions that need to update all records in a one-to-many relation, for example, marking all items from an order as "delivered" in TPC-C Delivery. We discuss how we find the write set for range updates in Section 3.2. We discuss methods for handling unknown write sets in Section 3.7.

Caracal supports get, scan, insert, delete and update operations for stored procedures. In non-deterministic databases, the insert operation fails if the row already exists, and both delete and update operations fail if the row does not exist, resulting in transaction aborts. In Caracal, none of these operations cause aborts. When a row exists, the insert operation updates the row with the newer value; when a row does not exist, the delete and the update operations do nothing and return false. These semantics are consistent with the `INSERT REPLACE`, `UPDATE WHERE` and `DELETE WHERE` statements in SQL. If the application needs to abort the transaction, it should check for the existence of the row via get operations at the beginning of the stored procedure before issuing any write operations, as described in Section 3.7.

Caracal's transaction API allows developers to annotate rows that may be contended. Caracal may dynamically split an update to such a row into a piece separate from the rest of the transaction and run the piece on a separate core when the row is contended. Currently, developers need to use C/C++ for writing stored-procedure transactions.

3.2 Concurrency Control in Caracal

The key challenge with deterministic concurrency control is supporting concurrent execution while ensuring the predetermined serial ordering of transactions. Caracal uses multi-versioning and transaction batching to support concurrent execution. With multi-versioning, transactions can read and write different versions of rows concurrently, allowing writers to create new row versions safely while readers are accessing old versions. Transaction batching enables determining dependencies across transactions in a batch and parallelizing independent data accesses while ensuring the serial order.

Caracal batches transactions into epochs and executes these transactions in two phases, *initialization* and *execution*, as shown in Figure 1. The initialization phase performs

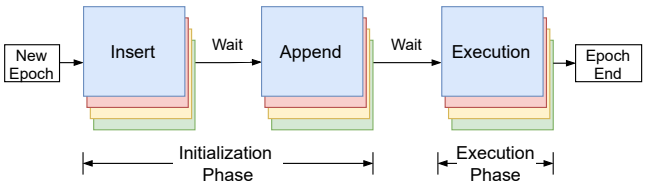


Figure 1. The Caracal Architecture

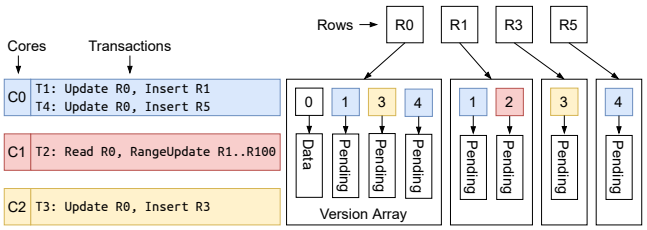


Figure 2. Transaction Processing in Caracal

concurrency control for all the transactions in the epoch, while the execution phase runs these transactions. By default, Caracal assigns transactions to cores in round-robin order and each core initializes and executes the transactions assigned to it. Transactions can be assigned to cores using other policies, e.g., to improve load balance or locality.

Figure 2 shows an example in which Caracal batches four transactions and processes them on three cores in an epoch. The initialization phase uses the write sets of the transactions to create corresponding row versions with a PENDING value that indicates that this version is a placeholder whose data has not been produced yet. The write set of a transaction contains the keys of all the rows that are to be written (inserted, updated or deleted) by the transaction. For example, Transactions T_1, T_3 and T_4 update Row R_0 and so the initialization phase creates three PENDING versions for Row R_0. Unlike Bohm [7], which partitions the initialization phase by keys, Caracal reduces the impact of skewed accesses by performing shared initialization, i.e., transactions running on different cores may create versions for the same row, e.g., for R_0 and R_1 in Figure 2.

During the execution phase, writers update PENDING row versions without any synchronization. Caracal makes writes visible before transactions complete, similar to PWV [8]. This approach is safe because transactions in Caracal do not abort after issuing their first write, as explained in Section 3.7, ensuring that transactions read committed data. Readers can observe all the row versions that will be created in the epoch (due to the initialization phase) and thus can determine the correct previous version to read based on the serial order. This guarantees that readers always read the latest committed version. For example, Transaction T_2 will read the version of Row R_0 that is written by Transaction T_1 in Figure 2. Readers synchronize with writers by waiting when this version is PENDING, until it is written. No further synchronization is needed since the execution is deterministic, deadlock-free, and has no concurrency-control related aborts.

For most transactions, the write set is provided by the developer, but inferring the write-set keys accurately for transactions that perform range updates is difficult, especially when transactions are initialized concurrently. Consider the range update in the row range R_1 to R_{100} performed by Transaction T_2 in Figure 2. Transaction T_1 inserts R_1, T_3 inserts R_3 and T_4 inserts row R_5, and these rows lie within the update range. Since T_2 is serialized after T_1 but before T_3 and T_4, T_2 should update Row R_1 but ignore Rows R_3 and R_5. Thus, we require that T_2's write set contain R_1 but not R_3 or R_5. One method for handling range updates is to run the initialization phase serially so that we can perform a range scan during initialization. This scan would determine that T_2's write set contains R_1, since the insert of Rows R_3 and R_5 would not have been initialized yet. However, this serial initialization, such as in Calvin [21], will not scale well. Bohm [7] partitions the initialization phase and runs initialization serially in each partition. However, as we show in our evaluation, this approach only scales when the workload is partitionable and can be load balanced.

We handle range update operations in Caracal by splitting the concurrent, shared initialization phase into two steps, *insert* and *append*, as shown in Figure 1. During insert, we create rows and their initial versions for all newly inserted (but not updated) keys (e.g., create Rows R_1, R_3 and R_5 in Figure 2). This step ensures that all rows that are inserted in the epoch will be visible in the next step. During append, we perform a range scan to acquire the write-set keys for a range update. Then, we insert row versions for the keys in the write set. If a key is present in the range, we check the first version of the key, and only append a new version if the first version of the key is smaller than the serial id of the range update. In the example in Figure 2, this step ensures that T_2's write set contains R_1 but not R_3 or R_5.

As an example, the TPC-C Delivery transaction needs to update all the purchase items for a given order id within a warehouse district. Since these items are not known in advance, we need to scan the key range [warehouse, district, order_id, 0] to [warehouse, district, order_id, infinity] in the `OrderLine` table to determine the items. This scan is performed during the append step and it determines the items correctly because they have already been inserted in the previous insert step.

A side effect of splitting the initialization phase into separate insert and append steps is that we can speed up read-only operations when the read set or the read range is known. In this case, the index read and scan operations can be performed during the append step. We show in Section 5.5 that this optimization can help reduce contention by reducing index operations during execution.

During the execution phase, a transaction needs to find the latest version to read that satisfies its serial order. The typical way to implement a multi-versioned database [6, 10, 16, 19, 24] uses a linked list of versions for each row object. This

design assumes that each row has a small number of versions, so searching the list will be inexpensive. This assumption is usually true for non-deterministic databases, but with batching in deterministic systems, some hot keys may have many versions, all of which must be accessed by transactions in the epoch, making linked-list traversal expensive.

We observe that epoch-based concurrency control allows Caracal to use arrays for efficient versioning since all row versions are created during the initialization phase, and no versions are created or deleted during execution. As shown in Figure 2, each element in the version array is a version id (i.e., the serial id of the transaction that created this version) and the version pointer. To accelerate lookups during execution, the version array is kept sorted by version ids. Instead of traversing all versions during execution, a transaction does a binary search to find the latest version to read with respect to its serial order. To further optimize this procedure, Caracal first searches around the latest updated version since readers are more likely to read the latest version. If we find the required version around the latest update, we can avoid the cost of a full binary search.

In the append step, Caracal adds new row versions to the sorted version arrays using insertion sort. Since transactions are processed in roughly increasing transaction serial id order across cores, each append is usually close to the end of the version array, and insertion sort is efficient in this case.

Next, we describe two optimizations, batch append and split-on-demand, that Caracal uses for managing contention during the initialization and execution phases, respectively.

3.3 Initialization Phase: Batch Append

Caracal supports concurrent initialization by synchronizing access to the shared row version arrays with per-row locks. For uncontended workloads, acquiring the row lock before inserting a new version scales well. However, under contention, this locking operation limits scalability, similar to other shared-memory databases.

Our batch append optimization helps manage contention during the initialization phase. We observe that in our deterministic scheme, newly added versions do not need to be immediately visible. Within an epoch, as long as the row versions are created before the execution phase, transactions will execute deterministically and the final outcome will follow the serial order. Thus row locks can be acquired in any order, which enables batching the append operations for the row versions and running them scalably with fewer locking operations. Also, locks are acquired one row at a time and so lock reordering doesn't cause any deadlocks.

Instead of waiting to acquire a row lock for each version that is inserted, we can simply append the new row version to a per-row, per-core, fixed-size buffer, without any locking. When this buffer is full, or at the end of the initialization phase, each version in the buffer is appended to the corresponding rows' version array using insertion sort, which lowers overall lock contention. The buffer is then cleared and reused. Note that the versions in a row buffer may not be at the end of the version array, which could increase the cost of insertion sort. We evaluate this cost in Section 5.5. As an example, in Figure 2, the two updates to Row R_0 on Core 0 can be batched, which reduces the number of locking operations on R_0 from 3 to 2.

A naive implementation of this approach would require an excessive amount of memory for the per-core buffers. To use memory more efficiently, we only assign per-core row buffers to contended rows. If a row doesn't have a row buffer, Caracal tries to acquire the row lock to directly insert the row version into the version array. However, if acquiring the row lock fails, indicating contention, a per-core row buffer is created. Thereafter, versions are appended to the slots in the row buffer. We describe our memory-efficient implementation of the batch append optimization in Section 4.1.

3.4 Execution Phase: Split-on-Demand

The execution phase synchronizes data accesses by ensuring that a transaction reading a row version waits until the version is written. Under heavy contention, this wait synchronization can occur frequently leading to significant cache coherence traffic among processors. We aim to reduce this contention during the execution phase.

We observe that a small number of rows are updated repeatedly during contention. To reduce contention due to cache coherence, we would like to cluster and schedule these updates on a small set of cores. However, along with performing these contended updates, transactions also perform other uncontended operations that should be scheduled on all cores for scaling. We resolve this tension by introducing a fine-grained transaction piece interface in Caracal that allows an application developer to annotate rows that *may* contend. When such a row is identified as contended, its update operation is dynamically split from the transaction into a separate piece that is run independently from the rest of the transaction code on a core that will minimize contention while maintaining load balance.

To illustrate the idea, suppose accesses to Row R_0 in Figure 2 are contended during execution. We can split Transaction T_3's update to Row R_0 into a separate piece and execute it on Core 0 to reduce contention, while still executing the rest of Transaction T_3 on Core 2.

This approach works well in a deterministic database because all the transaction pieces will eventually commit, and commit in an order that is consistent with the predefined serial order. In contrast, in a non-deterministic database the main transaction would need to synchronize with its split pieces to determine a serial order. For example, with two-phase locking, the main transaction would need to wait for its split pieces to finish before it can release locks.

3.4.1 Specifying Contended Operations.

Application developers specify that a row may be contended by using the `ApplyRowUpdate(Row, CallbackFunction, Weight)` row update function. This interface takes three arguments: 1. the row to update, 2. a callback function that specifies how the row values should be updated, and 3. an estimate of the amount of work needed by the callback function for scheduling purposes. When the row is likely to be contended, Caracal splits the callback in a separate piece and invokes the piece independently of the transaction. Otherwise, the callback function is invoked serially within the transaction since splitting uncontended operations provides no performance benefits. Instead, it introduces memory and cache miss overheads associated with running an extra transaction piece. Note that developers do not specify the core on which to run the piece, since Caracal automatically schedules it. Currently, Caracal does not allow specifying contention for reads for two reasons: 1) reads are less contended due to multi-versioning, and 2) we do not require the read sets of transactions and so cannot estimate contention due to reads.

The callback function must handle data dependencies with the row update in the split piece. For example, say an update to a contended row B depends on row A's value. The callback function for row B can track this dependency by waiting for the access to row A to complete in the main transaction, an approach similar to rendezvous points in PWV [8]. However, this dependency tracking requires expensive synchronization. In our shared-memory approach, a more efficient solution is to access row A and update row B within the callback function so that the data dependency is preserved. Currently, we assume that the callback function will update at most one contended row, and so the programmer needs to specify `ApplyRowUpdate()` for each potentially contended row. Note that the callback function performs an update and cannot issue aborts (See Section 3.7).

We normally assign a callback weight based on the number of row updates in the callback function. However, if the callback function performs read-heavy operations, such as scanning many rows, then this weight is increased.

Caracal's split-on-demand introduces a parallel programming model similar to PWV and Granola for reducing contention. However, while PWV splits transactions and executes them in parallel to speed up execution, Caracal splits transactions and executes the split pieces on *fewer* cores to reduce contention. Unlike partitioning, Caracal can assign pieces to arbitrary cores, which provides more flexibility for scheduling and load balancing. For example, Caracal can avoid splitting parts of a transaction that have data dependencies, avoiding any synchronization costs. Also, the annotations do not need to be accurate because Caracal only splits rows that it determines are contended, thus reducing the overhead of splitting uncontended rows. Finally, the split-on-demand programming model is optional: developers are encouraged to use it when contention is a concern.

```
OnInitializationPhaseFinish():
  sum = 0
  for row in contention_set:
    # contended if row.nr_pending_versions >= threshold
    row.start = sum
    sum += row.sum_weights

ApplyRowUpdate(row, callback):
  if row not in contention_set:
    # do not split...
  rand = RandomNumberFrom(row.start,
      row.start + row.sum_weights)
  core = NR_CORES * rand / sum
  RunPieceOnCore(core, new Piece(row, callback))
```

Listing 1. Probabilistic Pieces Assignment in Caracal

3.4.2 Scheduling Contended Operations.

Our approach requires efficiently identifying contended rows so that transactions can be split dynamically when they access these rows. Fortunately, in our batched deterministic model, we know the number of row versions that are created in an epoch by the end of the initialization phase. When the number of versions exceeds a certain threshold, we predict the row updates will contend during execution.

During execution, Caracal needs to schedule all the contended pieces across all cores. To reduce contention, all the pieces updating a contended row should be *clustered* to run on a small set of cores. Furthermore, to *balance load* among all cores, each core should be assigned pieces based on their weights, so that the piece weights are balanced. Unlike partitioning, Caracal does not require that all the pieces accessing a contended row be assigned to a single core. As shown below, this simplifies load balancing, and it also allows leveraging parallelism when running these contended pieces since each core is also running other uncontended pieces.

Caracal fulfills our clustering and load balancing requirements described above by probabilistically packing pieces to one or more cores, while ensuring that cores process similar amounts of work. Listing 1 shows our piece assignment algorithm that runs at the end of the initialization phase. It operates on a set of rows that are marked contended and have at least as many pending versions as the threshold, which we call the *contention set*. For each row in the contention set, `row.sum_weights` is the sum of the callback weights of all the pieces associated with this row. We sum up the weights from all the rows in the contention set and keep the partial sum in `row.start`. When issuing transactions in the execution phase, `ApplyRowUpdate` detects if the row is in the contention set, and if so, we assign the piece to a core according to a random number between `row.start` and `row.start + row.sum_weights`. The probabilistic assignment algorithm is inspired by lottery scheduling [23].

Figure 3 shows an example assignment. Four rows are contended and have a total of 24,000 versions, so each core is assigned 6,000 versions. All updates to Row 1 and Row 4

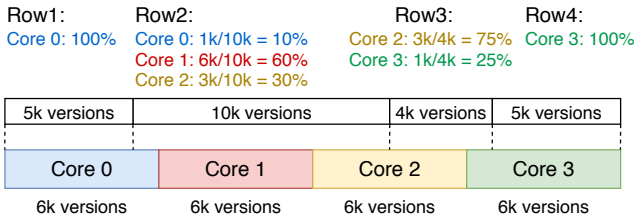


Figure 3. Load Balancing Split Pieces

are assigned to Core 0 and Core 3, respectively. However, updates to Row 2 are probabilistically split across Cores 0, 1 and 2, in the ratio 10%, 60% and 30%. Similarly, updates to Row 3 are split across Cores 2 and 3. Our evaluation shows that our probabilistic partitioning scheme has performance comparable to or better than an offline optimal partitioning scheme that assigns each contended row to a single core.

3.5 Garbage Collection

Caracal is multi-versioned and so it needs to perform garbage collection (GC) to reclaim unused versions in the version array. We use a *major* collector at the end of each epoch to reclaim versions for rows that have been updated in the epoch. This collector is expensive because it operates on older versions that are likely to have poor cache locality. It also pollutes the CPU cache with cold rows and version arrays, which affects the performance of the initialization phase in the next epoch. We improve GC performance in two ways. First, we use a *minor* collector that operates during the initialization phase and frees unused versions from the rows that are being initialized. Since the database has to access these rows during initialization anyway, collecting unused versions at this point has better cache locality. Second, we modify the major collector to only collect row versions that have not been updated in the last K epochs. The intuition is that the rows that have been recently updated are likely to be updated again and will be collected by the minor GC. By default, K is set to 8 epochs. Currently, our garbage collector does not support collecting old versions within an epoch, which can improve performance significantly [3, 12].

3.6 Logging and Recovery

Non-deterministic databases log the outputs of committed transactions to durable storage for recovery. With determinism, transactions executing in the pre-determined serial order always have the same outcome. Thus, Calvin [21] proposed logging the transaction inputs before executing the transactions. When the database fails and restarts, it replays the log by re-executing the transactions deterministically. Caracal uses a similar logging method. At each epoch, Caracal logs transaction inputs to storage concurrently with transaction processing. However, transaction results are returned to application clients only after all the transaction inputs in the epoch are logged and persisted. During recovery, Caracal replays transactions until the last persisted epoch.

To reduce the replay time during recovery, Caracal needs to perform periodic checkpointing. Since our logging format is logical, our checkpointing needs to be atomic. When a failure occurs, Caracal's checkpoint image contains consistent and complete data until a certain epoch. Then, during recovery, Caracal only needs to replay transactions from later persisted epochs. Checkpointing can be implemented efficiently using copy-on-write features in modern file systems that allow sharing storage across files [1]. Currently, Caracal implements logging but not checkpointing.

3.7 Limitations of Determinism

Caracal inherits certain known limitations from deterministic concurrency control. Next, we discuss these limitations and how we address some of them.

Aborts. There are two types of aborts in databases, system-level aborts due to concurrency control, and application-level aborts in transaction logic. As an example of the latter abort, a transaction placing an order should abort if there is not enough stock left. Caracal's deterministic concurrency control eliminates all system aborts. For application aborts, similar to PWV [8] and Granola [5], Caracal requires the application developer to issue any aborts before the transaction issues any writes. This limitation can often be handled by a transaction performing initial validating reads and aborting before performing updates. Other recent systems, such as Rococo [13] and Janus [14], also have similar restrictions.

To support application aborts before any writes have been issued, the transaction writes an IGNORE marker in a PENDING version. When a transaction reads an IGNORE marker, Caracal skips to a previous version that does not have an IGNORE marker. Similarly, conditional updates in which some rows in the write set are only updated when a certain condition is met are handled using the IGNORE marker.

Write Sets. Caracal requires transactions to declare write set or write ranges before execution. For some transactions, it may not be possible to infer the write set from the transaction inputs before execution. Calvin [21] proposes using *reconnaissance transactions* to deal with this issue. First, the transaction is run as a read-only transaction that performs the reads needed to infer the write set. Then, in the next epoch, the transaction declares its write set and executes as a normal transaction. During this execution phase, a transaction first checks if the rows read in the read-only phase have changed, and if so, the transaction aborts using the mechanism described above. This approach generally works well since reconnaissance transactions are often used to perform secondary index lookups on fields that change infrequently.

Currently, Caracal relies on manual effort to declare the write set of a transaction. Our API only allows transactions to perform writes that have been declared. However, if a user declares the write set but does not update the value, read

transactions may wait indefinitely. Our current implementation flags a write transaction when a read waits too long. A more robust implementation would check for missing writes on transaction completion. A query compiler framework that determines the write set would make it easier to use Caracal.

4 Implementation

Caracal is a main-memory, multi-core database that supports efficient single node operation and scaling out to multiple nodes. Our implementation uses C++17 and leverages language features such as lambda functions and templates to provide a developer-friendly API for writing stored procedures, including distributed transactions. Transactions are split into pieces represented by lambda functions; pieces can be dispatched to multiple cores or multiple machines.

Caracal assigns a globally unique 64-bit serial id (consisting of the epoch number, a per-node sequence counter, and a node id) and a core id (generated in a round-robin manner) to each transaction in an epoch. Caracal uses the core id to assign transactions to cores, ensuring that load is roughly balanced. In a real deployment, these operations would be carried out by the event scheduler that processes incoming network requests, but for our experiments in Section 5 we do them while generating transactions.

Next, we describe our implementation strategy and several optimizations that we have applied to our epoch-based, deterministic, multi-versioned design.

4.1 Initialization of the Version Array

Caracal uses sorted arrays for row versions, and cores insert PENDING versions during epoch initialization using insertion sort (Section 3.2). If the version array fills up, we re-allocate it with a larger capacity. During garbage collection, we free the version data pointed to by the unused array entries, clear the version number, and set the pointer to NULL to distinguish a free entry from a PENDING version. Garbage collection does not shrink the version array itself to reduce repeated reallocations in subsequent epochs. We expect that the version array for a row will quickly grow to the appropriate capacity for a particular workload, and then remain at that size.

Our batch append optimization requires a per-core buffer object that is associated with each row to accumulate PENDING versions when there is contention on the row-level lock. The simplest design is to add N buffer pointers to the row object (where N is the number of cores), each of which points to a pre-allocated per-core buffer. This design impacts cache utilization since the row object has many buffer pointers (e.g., with 32 cores each row object will need an extra 256 bytes for these pointers). Also, creating per-core buffers for every row in the database would increase memory footprint dramatically with little benefit since most rows are uncontended and these buffers would never be used. In Caracal, each row object only needs a single index value to represent

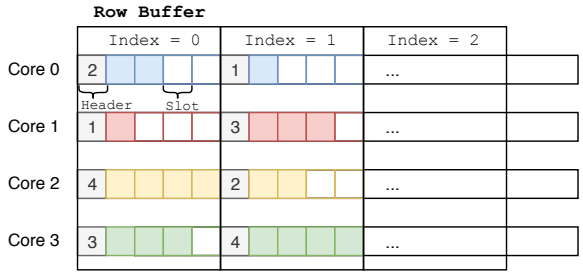


Figure 4. Per-core Buffers for Batched Initialization

the per-core buffers. The key idea is to pre-allocate memory for the per-core buffers, but assign buffers only to contended rows during initialization.

On database startup, Caracal pre-allocates memory for each core from the local NUMA zone (see Figure 4; each core's chunk is shown with a different color). An index value represents a row buffer, consisting of per-core buffer objects. Each buffer object has a header and a fixed number of slots for the row versions (e.g., 4 slots in Fig. 4). The header tracks the number of filled slots (i.e., the colored boxes in Fig. 4).

If a row doesn't have a row buffer (indicated by a special PENDING index value), Caracal tries to acquire the row-level lock and add the PENDING version to the row object's version array. Acquiring the row lock will fail if there is contention on the row, which triggers the assignment of a pre-allocated per-core row buffer by incrementing the global index value. The index of the assigned row buffer is stored in the row object, and PENDING versions are subsequently appended to the slots in the buffer objects.

At the end of the initialization phase, each core batch-appends all versions in its own slice of each row buffer to the version arrays of the corresponding rows. A core's batch-append operations can occur while other cores are still running the initialization phase and possibly allocating new row buffers. This is safe because the newly allocated row buffers will not contain any row versions for cores that have finished the initialization phase. For example, in Figure 4, say Core 1 finishes initialization and batch-appends the versions in the red buffers until `Index 1`. If another core allocates a row buffer at `Index 2`, that row buffer will not have any versions for Core 1, since Core 1 has already finished initialization. If all cores finish initialization at exactly the same time, they may contend on some rows during batch-append. If this happens, Caracal will delay batch-appending the contending row and process other rows first.

4.2 Transaction Scheduler

Caracal executes transactions using per-core kernel threads. Each kernel thread implements a user-level thread scheduler for dispatching and executing transactions on the core (to avoid confusion, we refer to the kernel thread as a core). Each scheduler runs two types of threads: a dispatch thread and one or more worker threads. The scheduler also maintains

a piece queue, consisting of transaction pieces that worker threads execute on that core.

The dispatch thread splits incoming transaction code into multiple pieces (e.g., for our split-on-demand optimization), and adds these pieces to the piece queue. Due to on-demand splitting, pieces may be added to the piece queues of other cores. To reduce contention, each core keeps a small buffer for each per-core piece queue, and appends pieces in a batch.

Each time a worker thread chooses a piece to run, it will pick the one with the smallest serial id from its piece queue. This approach may deadlock since pieces may be dispatched from other cores concurrently. For example, say the worker thread on Core 2 is running a piece of transaction T_2 since it has the smallest serial id. While this transaction is running, the dispatch thread on Core 1 could add a piece of Transaction T_1 (serialized before T_2) to Core 2's piece queue. If T_2's read depends on T_1's write, then T_2 will wait forever even though T_1 is in the piece queue.

We address this issue by using a preemptive scheduler. While a worker thread waits for a read during the execution of a piece, it spins and periodically checks the piece queue for a piece with a smaller serial id. If such a piece exists then Caracal suspends the current worker thread and creates a new worker thread to run the piece with the smallest serial id. This worker thread executes pieces with serial ids smaller than the suspended thread's serial id. When it finishes, it resumes the suspended thread. When a worker thread waits, we do not preempt it to run transaction pieces with a larger serial id since these pieces may depend on the preempted piece, causing repeated thread creation and preemption and degrading performance.

Caracal runs the dispatch and worker threads in parallel in each of the phases shown in Figure 1 and uses a barrier to synchronize phases. Each dispatch thread tracks the number of pieces it has dispatched using a global piece counter. When a worker thread finds that its piece queue is empty, it decrements the piece counter with the number of pieces it has executed. While the piece counter is larger than zero, the worker thread waits for pieces to be added to the piece queue. Otherwise, the phase is complete and the worker threads exit. Then a single, light-weight, control thread restarts the dispatch threads on all the cores for the next phase. We use `epoll` to coordinate the user-level threads, which also enables multiplexing network IO for multi-node Caracal.

The `ApplyRowUpdate(Row, CallbackFunction)` call dynamically splits the callback function into a separate piece if the row was marked as contended during the initialization phase. It returns a `Future` object representing the callback result. If the application depends on this result, it uses the future's `wait` function to wait for callback completion. When the callback function is split, the `wait` function is implemented in the same way as a piece waiting on a read. Otherwise, the `wait` function invokes the callback directly.

4.3 Other Optimizations

Caracal inlines the version array and version data for tables with small rows and infrequent updates, similar to Cicada [12]. Caracal's two-phase execution model causes the same keys to be searched twice, once in each phase. To address this issue we allocate a row cache for each transaction. During initialization, after searching the index for a row, we store a pointer to the row in the row cache. During execution, transactions first check the row cache to read or write data. All accesses to the write set are found in the row cache.

5 Evaluation

We compare Caracal against three deterministic approaches: Granola [5], Bohm [7], and PWV [8]. All these baselines use partitioning to improve performance and to eliminate contention. They are all designed for uniform, partitionable workloads and don't tolerate skewed workloads well. Our evaluation shows that Caracal outperforms all these previous deterministic databases when the workload is both contended and skewed. We do not compare against Calvin [21] since its single-threaded initialization phase makes it non-scalable, and because Bohm already improves on Calvin.

We use the YCSB benchmark and a TPC-C like benchmark to evaluate all databases. For both benchmarks, we first evaluate how Caracal performs against the baseline partitioned systems for both uncontended and contended workloads. Then, we show how Caracal's optimizations benefit these workloads.

5.1 Hardware and Software Platform

We evaluate Caracal's single-node performance on an HP ProLiant DL560 Gen8 with four Xeon E5-4620 processors. Each processor has eight physical cores with 16MB last level cache and one NUMA zone, for a total of four NUMA zones on the machine. Each NUMA zone has eight 16GB DIMMs of DDR3 DRAM, for a total of 512GB DRAM. Each DIMM has two ranks and operates at 1333 MT/s.

For the distributed experiments in Section 5.8, we use 8 HP ProLiant DL160 Gen8 machines connected using commodity 10Gb Ethernet. Each machine has two Xeon E5-2650 processors with eight physical cores, 20MB last level cache and 32GB DRAM.

All machines run CentOS 7.6 with 1062.el7 kernel (released on Aug 7, 2019). We compile Caracal, Granola, Bohm, and PWV using Clang 9, all with `-O3` optimization.

5.2 Comparison Databases

We implement the Granola, Bohm, and PWV deterministic databases using our code-base to make the comparison as fair as possible. We process batches of transactions in epochs for all implementations so that they have bounded latencies for transactions, similar to Caracal.

Bohm is multi-versioned. It partitions the initialization phase, so our implementation does not acquire locks on rows during initialization. We also include several optimizations compared to original Bohm, such as using a version array and binary search to speed up searching versions.

The original Bohm partitioning implementation uses filtering to assign work to partitions during initialization. When using 32 cores and assuming a uniformly distributed workload, each core would need to filter out 31/32 of the write-set keys, so 97% of the CPU cycles would be wasted during initialization. To reduce this overhead, our Bohm implementation reuses our piece infrastructure to implement partitioning. We split the initialization task into pieces and dispatch these pieces to the relevant cores. This splitting and dispatch must be done online since it depends on the input. We optimize this operation by dispatching pieces in small batches, which reduces contention on the piece queue.

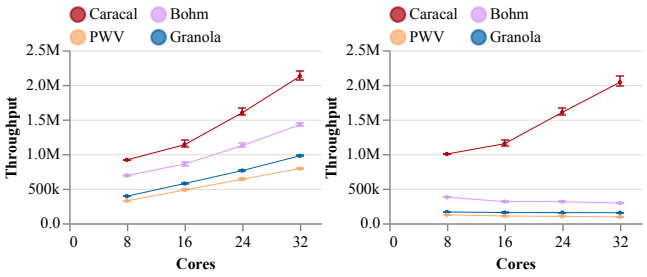
Granola is single-versioned. We implement it by eliminating the initialization phase so each epoch contains just the execution phase. Granola does not perform any concurrency control and so each partition executes transaction pieces in serial order. Our Granola implementation supports dependent transactions via PWV-style rendezvous points [8] but these transactions may stall the serial execution.

PWV is single-versioned. It uses a per-partition dependency graph to track conflicts within the partition. If a piece depends on a piece that hasn't finished execution on another partition, the dependency graph allows PWV to schedule other non-conflicting pieces in the partition (unlike Granola). Our implementation constructs the dependency graph using row or table access information during the initialization phase. Then, we schedule pieces based on the dependency graph in the execution phase and support early write visibility. No synchronization is needed between the initialization and execution phases since both are partitioned.

All experiments with multi-versioning are performed with garbage collection. We use a statically generated trace to avoid network artifacts, similar to other high-performance database evaluation. We run all experiments for ~ 5M transactions, and for each workload, we batch the same number of transactions per epoch. This batch size is chosen to ensure that in Caracal an epoch lasts roughly 40-60 ms and so the maximum transaction latency is roughly 100 ms. We discuss the throughput at maximum scale (32 cores) when comparing systems unless otherwise stated.

5.3 YCSB

We use a transactional YCSB microbenchmark. The original YCSB is a key-value store benchmark [4] that does not specify transactions. In our evaluation, we group 10 unique key accesses (either read or write) in a single transaction. We use a total of 10M keys, and each row is 1000 bytes. A read operation reads the entire row, while a write operation updates the first 100 bytes of the row, representing an update



(a) YCSB Uniform **(b)** YCSB Skew, $s = 0.9$

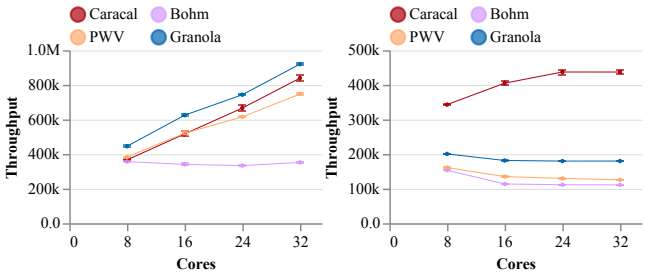
Figure 5. YCSB Performance (8:2 read/write ratio)

to a subset of the columns in a row. There are no data flow dependencies within a transaction. For Caracal, we specify that all update operations in YCSB may be contended and thus eligible for our split-on-demand optimization. Granola, Bohm's initialization phase, and PWV evenly partition the keys among all of the given cores.

We compare Caracal's performance with other databases under low contention by using a uniform distribution to choose keys in a transaction and use a read-write ratio of 8:2. Figure 5a shows that Caracal achieves 2.12 MTxn/s under low contention. Bohm, Granola, and PWV achieve 1.42 MTxn/s, 975 KTxn/s, and 791KTxn/s. Caracal outperforms partitioning because Caracal does not need to split transactions into pieces, while the other three databases need to split each transaction into pieces, based on the key accesses. With the uniform key distribution, most transactions have 10 pieces. The mean time to create pieces and schedule them is roughly 0.8 µs for Caracal, 6 µs for Bohm, 17 µs for Granola, and 20 µs for PWV. The rest of the transaction execution time lies between 13~16 µs for all databases, showing that the cost of partitioning the transaction has the most significant impact on performance in this workload. Bohm outperforms Granola because Granola splits pieces during execution. Granola outperforms PWV because PWV constructs the dependency graph, but YCSB does not have any rendezvous points, and so the dependency graph has no benefits in this workload.

We evaluate performance under skew by using a Zipfian distribution with $s = 0.9$ to choose the keys in a transaction. Figure 5b shows that Caracal achieves 2.04 MTxn/s throughput. It schedules transactions in round-robin order and thus is minimally affected by key skew. The other approaches use partitioning to eliminate contention but suffer under this skewed workload. Bohm's partitioned initialization takes 306 ms while Caracal's shared initialization takes 17 ms per epoch. Bohm performs better than PWV and Granola because its uses shared execution. Both Bohm and Caracal spend roughly 28 ms in the execution phase per epoch.

To evaluate Caracal's performance under higher contention, we create a highly contended YCSB workload. First, we set the read-write ratio to 0:10 so all 10 key accesses are update operations. Second, 3 of the rows are chosen from the entire database and the remaining 7 rows are chosen from a small



(a) YCSB Uniform Contention **(b)** YCSB Contention+Skew $s = 0.9$

Figure 6. YCSB Contention Performance (7/10 contended keys)

set of 77 rows that are spaced 2^{17} apart in the 10M key space. The 3 keys and the 7 keys are chosen using either a uniform or a skewed distribution from their respective set. These workloads trigger Caracal's contention optimizations.

Figure 6a shows performance under contention for the uniform distribution. Bohm does not scale because its shared execution is contended. It spends 30 ms for its partitioned initialization and 105 ms for execution, while Caracal spends 14 ms for initialization and 39 ms for execution each epoch. As shown in Section 5.5, Caracal's execution time improvement occurs due to the split-on-demand optimization. This uniformly contended workload, with no data dependencies across keys, is the best case for a partitioned database like Granola (920 KTxn/s), which outperforms Caracal (839 KTxn/s) by 9.7%. PWV's dependency graph construction and scheduling impose significant overheads for this workload.

Figure 6b shows the results when the contended keys are chosen using the Zipfian distribution. Even with this challenging skewed and contended workload, Caracal's performance still scales up to 24 cores. With 32 cores, Caracal (437 KTxn/s) achieves ~2.5× the performance of Granola (180 KTxn/s). Bohm spends 148 ms for initialization and 72 ms for execution while Caracal spends 20 ms for initialization and 33 ms for execution each epoch. In this workload, our split-on-demand optimization is applied to 80~82 contended keys that each have more than 512 versions. One key is heavily contended with over 30,000 versions. Our piece scheduling mechanism ensures that all cores are load balanced and updates this key on multiple cores, achieving higher throughput than using a single core for this key.

We also experimented with moderately skewed Zipfian distributions. At $s = 0.12$, Caracal matches Granola's performance (724 KTxn/s vs. 714 KTxn/s).

5.4 TPC-C Like

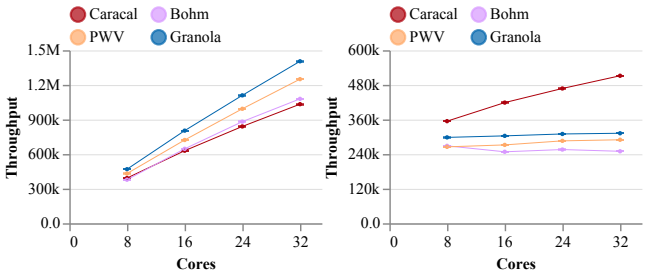
Next, we use the TPC-C OLTP benchmark to evaluate Caracal. Unlike YCSB, TPC-C specifies transaction behavior in detail and simulates the activities of a wholesale supplier that stocks items in multiple warehouses. Customers place new orders, pay for the orders, and items are eventually delivered to the customers. TPC-C models this process using 3 read-write transactions (NewOrder – 45%, Payment – 43%, Delivery – 4%) and 2 read-only transactions (OrderStatus – 4%, StockLevel – 4%). Although Delivery is only 4% of the transaction mix, it is heavy-weight compared to NewOrder and Payment and it creates conflicts with these transaction types. By default, the number of warehouses in TPC-C is the same as the number of cores in our experiments.

Our TPC-C like benchmark is based on TPC-C, with a few modifications because Caracal requires the write-sets of transactions. First, the TPC-C NewOrder transaction inserts new order items with an order id, generated by reading and incrementing a field in the `District` table. This prevents us from inferring the write-set of the NewOrder transaction before execution. Instead, we use auto-increment to generate the order id: when the NewOrder transaction inserts new order items, it fetches and increments an atomic counter inside the database. Silo refers to this optimization as "FastIds" [22]. Second, we modify the Payment transaction to remove the customer name lookup, which provides a customer ID by scanning a read-only index when the customer only provides a last name for payment. We remove the read-only index and limit the Payment transaction to only use the customer ID. For the two read-only transactions in TPC-C, we perform index lookup in the `OrderLine` table and its secondary index during initialization, which helps reduce contention. Finally, we ignore the scaling requirement in TPC-C, similar to all the baselines in our evaluation, as it imposes a throughput limit at each warehouse (Section 4.1.3 in the TPC-C spec.).

We could use reconnaissance transactions (Section 3.7) to run TPC-C's Payment transaction. The reconnaissance transaction would read the `CustomerName` index, which is read-only, and so the validation would always succeed, and the rest of transaction could then be performed using Caracal's deterministic protocol. However, for TPC-C's NewOrder, the reconnaissance transaction would need to read the `next_o_id` field, which is updated heavily and thus would cause validation failure. Our order id auto-increment avoids this issue.

Granola, Bohm and PWV partition TPC-C by warehouse. Caracal does not need partitioning but we pin transactions to cores based on their home warehouses for better performance, similar to other shared-memory databases such Silo [22], Cicada [12], STO[9] etc.

Default TPC-C has low contention. For higher contention, we use a single warehouse TPC-C workload [8, 9], which causes contention on the `Warehouse`, `District`, `Customer` and `Stock` tables. For this workload, the PWV paper [8] proposes partitioning all tables by `district_id`, except the `StockLevel` and the `Warehouse` tables. The latter two tables are assigned their own cores. Since there are 10 districts, the `District`, `NewOrder`, `Customer`, `Orders` and `OrderLine` tables each have 10 partitions. These 50 partitions are assigned to the rest of the cores in round-robin order. We have rendezvous points in the Delivery and the StockLevel transactions but not in the NewOrder transaction since we use auto-increment for the order id.



(a) TPC-C **(b)** TPC-C Single Warehouse

Figure 7. TPC-C and TPC-C Single Warehouse Performance

In Caracal, the NewOrder, Payment and Delivery transactions can cause contention in the `Stock`, `Warehouse`, `District` and `Customer` tables in the single warehouse TPC-C workload. We update these tables using our `ApplyRowUpdate` API. The Delivery transaction contains a data dependency. It scans the `OrderLine` table for items in an order to calculate the total and to mark these items as delivered. Then, it uses the calculated total to increment the customer balance. To handle contended updates to the `Customer` table, we merge the `OrderLine` range update and the `Customer` update in the same callback function.

Our PWV implementation uses coarse-grained dependencies [8], in which accesses to the `Stock` table are treated as a single edge in the graph, instead of an edge per row. This coarse-grained tracking reduces the graph size and the scheduling overheads significantly. Also, it avoids the need to declare the `Stock` table read-set in PWV, which is not feasible for the StockLevel transaction.

Figure 7a shows Caracal's throughput on default TPC-C compared to the other databases. Granola outperforms Caracal by 36% for three reasons. First, Granola is single-versioned, which provides better performance compared to using version arrays in Caracal and Bohm on this workload [9]. Second, default TPC-C is easily partitionable and its transactions can be run as independent transactions, making it well suited for Granola [5]. Finally, most TPC-C transactions are processed in a single piece, so unlike with YCSB (Figure 5a), there is minimal overhead of partitioning. Thus, PWV and Bohm also benefit from partitioning. PWV performs better than Bohm since it partitions transaction execution but its dependency tracking and scheduling add overheads compared to Granola. Both Bohm and Caracal are multi-versioned but Bohm partitions its initialization and is 4.4% faster than Caracal. This difference is due to locking operations during the initialization phase in Caracal.

Figure 7b shows the result for single warehouse TPC-C. Granola, PWV and Bohm do not scale for this workload, while Caracal shows much better scaling. Caracal outperforms Granola by 64% at 32 cores. Due to skew, Bohm takes 78 ms while Caracal takes 12 ms for initialization, while both take roughly 19 ms for execution per epoch. With partitioned execution in Granola and PWV, the `Stock` table partition becomes the bottleneck at 8 cores. The `Warehouse` table partition has 50% utilization, while the rest of the cores are 60-70% utilized. Compared to Granola, PWV's dependency-based scheduling reduces wait times due to a rendezvous point on the `Customer` table, but not the `Stock` table, which has coarse-grained dependencies, and so accesses to the table cannot be scheduled out-of-order.

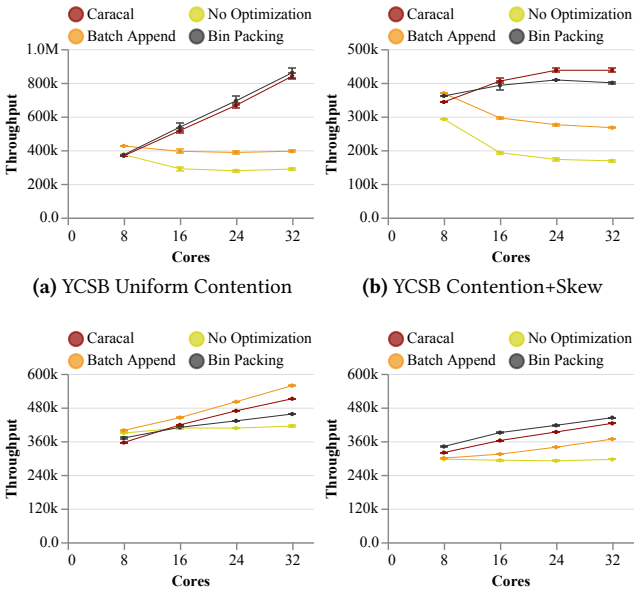
5.5 Impact of Optimizations

We have demonstrated Caracal's ability to manage contention. In this section, we show the impact of Caracal's contention optimizations by comparing Caracal's performance with three variations: 1) Caracal with both optimizations disabled, 2) Caracal with only batch append, and 3) Caracal with best-effort partitioning. The last variant enables Caracal's batch append and splits transactions based on the number of versions of the row. However, rather than probabilistically placing pieces onto cores, this variant performs best-effort partitioning using an offline bin packing algorithm; the rows are partitioned and the pieces updating a row are placed on one core exclusively. Note that we do not show the cost of bin packing, which is roughly 1-2 seconds for a 50 ms epoch.

For the uniformly contended YCSB workload, Figure 8a shows that Caracal with no optimizations experiences contention in both the initialization and execution phases; batch append improves throughput by 33% and when combined with split-on-demand, Caracal achieves 2.9× higher performance compared to no optimization. batch append reduces initialization time from 60 ms to 15 ms, and split-on-demand reduces the execution time from 105 ms to 39 ms per epoch at 32 cores. Our probabilistic placement of pieces has no visible overhead compared to using offline bin-packing partitioning (which is infeasible to use in a high-performance database).

For the skewed and contended YCSB workload, Figure 8b shows that batch append achieves 58.5% higher throughput (267 KTxn/s) compared to no optimizations (168 KTxn/s). Among all the contended rows, the distribution of the number of versions is also highly skewed. To maintain load balance, Caracal assigns a few highly contended rows to multiple cores. Bin packing performs 8.5% worse than Caracal because it is hard to partition the skewed keys while maintaining load balance. The two YCSB workloads have significant contention in both the initialization and execution phases and so both are our optimizations are required to achieve throughput scaling.

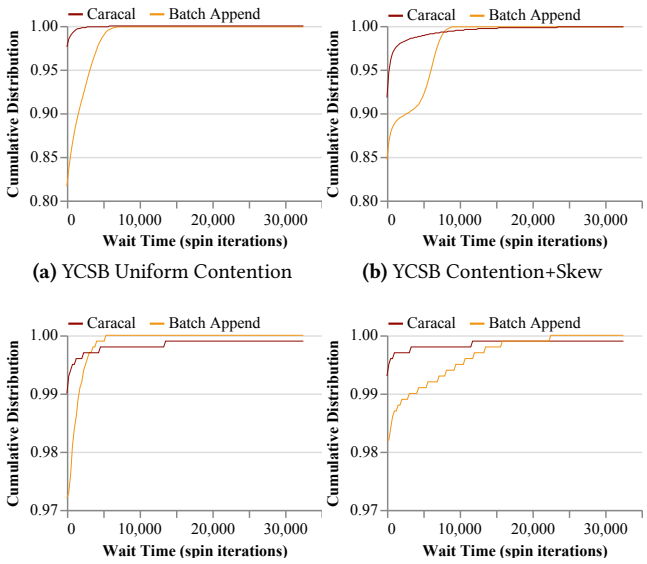
Figure 8c shows the impact of our contention optimizations for single warehouse TPC-C. In this case, batch append provides good scalability and reduces the initialization time from 31 ms to 12 ms per epoch at 32 cores. To understand why this optimization works, we measured lock wait times. Without batch append, the average and the 99.9 percentile lock wait times are 1.3 µs and 61 µs. With batch append, the corresponding numbers are 43 ns and 0.5 µs. The median number of element move operations in insertion sort



(a) YCSB Uniform Contention **(b)** YCSB Contention+Skew

(c) Single Warehouse TPC-C **(d)** Single Warehouse TPC-C (index lookup during execution)

Figure 8. Impact of Optimizations on Different Workloads



(a) YCSB Uniform Contention **(b)** YCSB Contention+Skew

(c) Single Warehouse TPC-C **(d)** Single Warehouse TPC-C (index lookup during execution)

Figure 9. CDF of Read Wait Time for Different Workloads

is 0 both without or with batch append, but surprisingly, batch append reduces the average number of element move operations in insertion sort from 4.9 to 1.4. Without batch append, the long tail in lock wait times blocks thread progress, causing more insertions to happen out of order.

In Figure 8c, the split-on-demand optimization adds overhead due to splitting pieces while its contention reduction benefits are small and so the execution time increases from 17 ms to 21 ms per epoch. We found that this workload stresses index lookup and concurrency control during initialization but transaction conflicts do not cause much contention during the execution phase. We illustrate this by performing index lookup for the two read-only transactions in the execution phase (instead of the initialization phase), which increases contention in the execution phase. In this case, Figure 8d shows that the batch append optimization improves performance by 24% and reduces initialization time from 30 ms to 10 ms. The split-on-demand optimization improves performance compared to no optimization by another 15% and reduces the execution time from 47 ms to 35 ms.

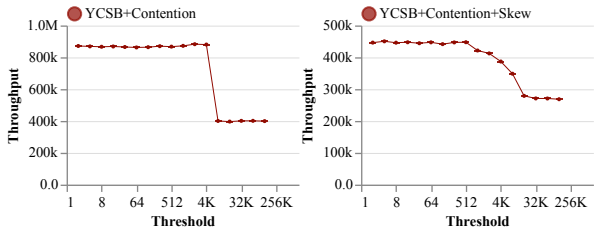
To further understand the behavior of the split-on-demand optimization, Figure 9 shows the CDF of the time (in spin iterations) that a read spins waiting on a pending version, for the workloads shown in Figure 8. Figure 9 shows that the split-on-demand optimization reduces the waiting time significantly for all the workloads except single warehouse TPC-C, which is consistent with Caracal's performance improvements over Batch Append shown in Figure 8. Single warehouse TPC-C has one heavily contended key and split-on-demand spreads accesses to this key across several cores, similar to the skewed and contended YCSB workload. However, in TPC-C, the different transaction types cause higher variance in transaction progress across cores. This increases tail wait times significantly because the pieces are run in the pre-determined serial order.

5.6 Tuning

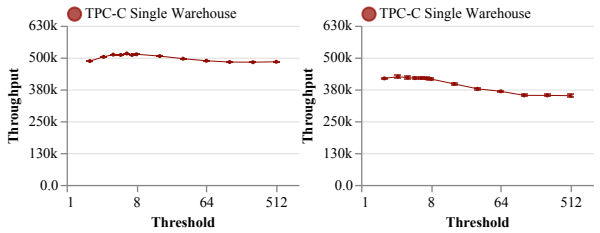
In this section, we show the performance impact of tuning the split-on-demand threshold. Figure 10 shows the throughput for 4 workloads for the different thresholds values shown on the X-axis. For YCSB Contention (Figure 10a) and YCSB Contention+Skew (Figure 10b), there is a large range for the optimal threshold. Starting from 1 version to 512 versions, Caracal always maintains optimal performance. For thresholds larger than 16K versions, performance decreases because split-on-demand is deactivated. With TPC-C Single Warehouse (Figure 10c), the range for optimal thresholds is relatively narrow: from 3 to 8 versions. For thresholds larger than 16 versions, Caracal splits fewer update operations and the performance drops by up to 6%. If we instead perform the index search for read-only transactions in the execution phase (Figure 10d), the range for optimal thresholds remains 3 to 8 versions. For thresholds larger than 8 versions, Caracal's performance drops by 17% in the worst case. We conclude that it is best to set a relatively low threshold when the database administrator expects contention. In that case, the worst performance penalty is 10% ~ 20%.

5.7 Latency

Caracal uses epoch-based concurrency control and so the commit latency (the time that a transaction stays inside the

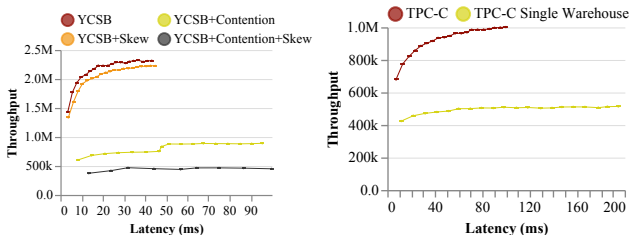


(a) YCSB Uniform Contention **(b)** YCSB Contention+Skew



(c) Single Warehouse TPC-C **(d)** Single Warehouse TPC-C (index lookup during execution)

Figure 10. Tuning Thresholds for split-on-demand



(a) YCSB **(b)** TPC-C

Figure 11. Transaction Latency and Throughput

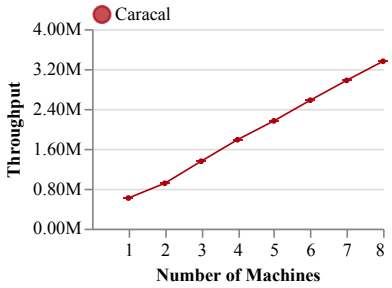


Figure 12. Distributed TPC-C Throughput

database) is the execution time of the entire epoch. In this section, we evaluate the latency-throughput trade-off in Caracal by varying the epoch size. Figure 11 shows Caracal throughput versus latency graph. For this experiment, we vary the epoch size from 5,000 transactions to 100,000 transactions and the corresponding epoch length is shown on the X-axis.

For sufficiently large epochs, changing the epoch size has a minor effect on throughput. Specifically, when the latency increases above 50 ms, the throughput increases by at most 6%. When the epoch size is decreased so that the latency is below 50 ms, Caracal's throughput begins to drop. Uncontended workloads like YCSB, YCSB Skew and TPC-C are affected the most. With the smallest epoch size, the latency drops to 8 ms but the throughput drops by 30% to 40%.

Contended workloads are less sensitive to smaller epochs. However, we see that YCSB Contention has a sharp throughput drop when the latency is reduced to less than 50 ms. Our investigation suggests the major garbage collector's overhead increases for YCSB Contention as we shrink the epoch size. This may indicate that we are running the major garbage collection too frequently.

5.8 Scaling Out

Caracal leverages determinism to scale out to multiple nodes. We evaluate Caracal's throughput on a cluster of 8 machines running TPC-C. In this experiment, we shard the data by warehouse. Each machine has 16 warehouses (one warehouse per-core). Roughly 14% of NewOrder transactions and 15% of Payment transactions access remote warehouses. With 8 machines, roughly 12% of NewOrder transactions and 13% of the Payment transactions are cross-machine transactions.

Figure 12 shows that Caracal achieves 610 KTxn/s using a single machine. As we add more machines, Caracal's throughput scales linearly, with no drop-off. With 2 nodes, Caracal achieves 900 KTxn/s, and with all 8 nodes, Caracal reaches 3.35 MTxn/s. As a rough comparison with a state-of-art distributed database, FaRMv2 [19] reports 12 MTxn/s for TPC-C using 90 machines and an expensive, high-speed Infiniband network. Caracal achieves a quarter of FaRMv2's performance with just 8 (slightly slower) machines using a commodity 10Gb Ethernet network.

6 Conclusions

Caracal is a high-performance, main-memory database designed for handling skewed and contended workloads. It uses a shared-memory, deterministic concurrency control scheme that enables reordering and parallelizing transaction execution with low overhead. We present two optimizations that allow scaling under highly contended workloads. The batch append optimization groups and reorders concurrency control operations because these operations are commutative. The split-on-demand optimization splits contending updates, which reduces the serial component of contended execution, and performs these updates on fewer cores, thus reducing contention. Compared to previous deterministic databases, these schemes enable Caracal to scale well under contention and skew.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Natacha Crooks, for their valuable feedback. We specially thank Michael Stumm, Ding Yuan, and the members of the Caracal group, including Zhiqi He and Shirley Wang, for their insightful suggestions. This work was supported by NSERC Discovery.

References

[1] [n.d.]. ioctl_ficlonerange(2). Linux Manual Pages. share some the data of one file with another file.

[2] Raja Appuswamy, Angelos C. Anadiotis, Danica Porobic, Mustafa K. Iman, and Anastasia Ailamaki. 2017. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads. *Proceedings of the VLDB Endowment* 11, 2 (Oct. 2017), 121–134. https://doi.org/10.14778/3149193.3149194

[3] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proceedings of the VLDB Endowment* 13, 2 (Oct. 2019), 128–141. https://doi.org/10.14778/3364324.3364328

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the Symposium on Cloud Computing, SoCC*. ACM, Indianapolis, Indiana, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[5] James Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *USENIX Annual Technical Conference - ATC*. USENIX Association, 21–33.

[6] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the International Conference on Management of Data - SIGMOD*. ACM, 1243–1254. https://doi.org/10.1145/2463676.2463710

[7] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proceedings of the VLDB Endowment* 8, 11 (July 2015), 1190–1201. https://doi.org/10.14778/2809974.2809981

[8] 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 (Jan. 2017), 613–624. https://doi.org/10.14778/3055540.3055553

[9] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proceedings of the VLDB Endowment* 13, 5 (Jan. 2020), 629–642. https://doi.org/10.14778/3377369.3377373

[10] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the ACM International Conference on Management of Data - SIGMOD*. 1675–1687. https://doi.org/10.1145/2882903.2882905

[11] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the Symposium on Operating Systems Principles - SOSP*. ACM, Shanghai, China, 104–120. https://doi.org/10.1145/3132747.3132751

[12] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the ACM International Conference on Management of Data - SIGMOD*. ACM, Chicago, Illinois, USA, 21–35. https://doi.org/10.1145/3035918.3064015

[13] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *Operating Systems Design and Implementation '14*. 479–494.

[14] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Operating Systems Design and Implementation '16*.

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

[16] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the ACM International Conference on Management of Data - SIGMOD*. ACM, Melbourne, Victoria, Australia, 677–689. https://doi.org/10.1145/2723372.2749436

[17] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2017. Scalable Replay-Based Replication For Fast Databases. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2025–2036.

[18] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2015. Design Principles for Scaling Multi-Core OLTP Under High Contention. In *Proceedings of the ACM International Conference on Management of Data - SIGMOD*. 1583–1598. https://doi.org/10.1145/2882903.2882958

[19] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *Proceedings of the ACM International Conference on Management of Data - SIGMOD*. ACM, Amsterdam, Netherlands, 433–448. https://doi.org/10.1145/3299869.3300069

[20] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the VLDB Endowment*. 1150–1160.

[21] 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 ACM International Conference on Management of Data - SIGMOD*. 1–12. https://doi.org/10.1145/2213836.2213838

[22] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the ACM Symposium on Operating Systems Principles - SOSP*. 18–32. https://doi.org/10.1145/2517349.2522713

[23] Carl A Waldspurger and William E Weihl. 1994. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*. 1–11.

[24] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proceedings of the VLDB Endowment* 10, 7 (March 2017), 781–792. https://doi.org/10.14778/3067421.3067427