Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks

Erfan Zamanian
Brown University
erfanz@alumni.brown.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Carsten Binnig
TU Darmstadt
cbinnig@cs.tu-darmstadt.de

Tim Kraska
MIT CSAIL
kraska@csail.mit.edu

ABSTRACT

Distributed transactions on high-overhead TCP/IP-based networks were conventionally considered to be prohibitively expensive. In fact, the primary goal of existing partitioning schemes is to minimize the number of cross-partition transactions. However, with the new generation of fast RDMA-enabled networks, this assumption is no longer valid.

In this paper, we first make the case that the new bottleneck which hinders truly scalable transaction processing in modern RDMA-enabled databases is *data contention*, and that optimizing for data contention leads to different partitioning layouts than optimizing for the number of distributed transactions. We then present Chiller, a new approach to data partitioning and transaction execution, which aims to minimize data contention for both local and distributed transactions.

1. INTRODUCTION

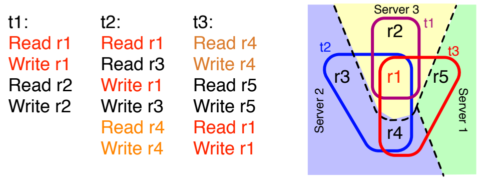
The common wisdom is to avoid distributed transactions at almost all costs as they represent the dominating bottleneck in distributed database systems. As a result, many partitioning schemes have been proposed with the goal of minimizing the number of cross-partition transactions (e.g., [2, 10]). Yet, a recent result [16] has shown that with the advances of high-bandwidth RDMA-enabled networks, neither the message overhead nor the network bandwidth are limiting factors anymore, significantly mitigating the scalability issues of traditional systems. This raises the fundamental question of how data should be partitioned across machines given high-bandwidth low-latency networks. We argue that the new optimization goal should be to minimize contention rather than distributed transactions.

In this paper, we present Chiller, a new partitioning scheme and execution model based on two-phase locking which aims to minimize contention. Chiller is based on two complementary ideas: **(1) a novel commit protocol** based on re-ordering transaction operations with the goal of minimizing the lock duration for contended records through committing such records early, and **(2) contention-aware partitioning** so that the most critical records can be updated without additional coordination, which is different from existing partitioning algorithms that aim to minimize the number of distributed transactions. For example, assume a simple scenario with three servers in which each server can store up to two records, and a workload consisting of three transactions t_1, t_2, and t_3 (Figure 1a). All transactions update r_1. In addition, t_1 updates r_2, t_2 updates r_3 and r_4, and t_3 updates r_4 and r_5. The common wisdom would dictate partitioning the data in a way that the number of cross-cutting transactions is minimized; in our example, this would mean co-locating all data for t_1 on a single server as shown in Figure 1b, and having distributed transactions for t_2 and t_3.

However, as shown in Figure 2a, if we re-order each transaction's operations such that the updates to the most contended items (r_1 and r_4) are done last, we argue that it is better to place r_1 and r_4 on the same machine, as in Figure 2b. At first this might seem counter-intuitive as it increases the total number of distributed transactions. However, this partitioning scheme decreases the likelihood of conflicts and therefore increases the total transaction throughput. The idea is that re-ordering the transaction operations minimizes the lock duration for the "hot" items. More importantly, after the re-ordering, the transaction commit relies entirely on the success of acquiring the lock for the most contended records. That is, if a distributed transaction has already acquired the locks for all non-contended records (referred to as the *outer region*), the commit outcome will only depend on the contended records (referred to as the *inner region*). This allows us to make all updates to the records in the *inner region* without any further coordination. Note that this partitioning technique primarily targets high-bandwidth low-latency networks, which mitigates the two most common bottlenecks for distributed transactions: message overhead and limited network bandwidth.

To provide such a contention-aware scheme, Chiller is based on two complementary ideas that go hand-in-hand: a contention-aware data partitioning algorithm and an operation-reordering execution scheme. First, different from existing partitioning algorithms that aim to minimize the number of distributed transactions (such as Schism [2]), Chiller's partitioning algorithm explicitly takes record contention into account to co-locate hot records. Second, at runtime, Chiller uses a novel execution scheme which goes beyond existing work on re-ordering operations (e.g., QURO [15]). By taking advantage of the co-location of hot records, Chiller's execution scheme reorders operations such that it can release locks

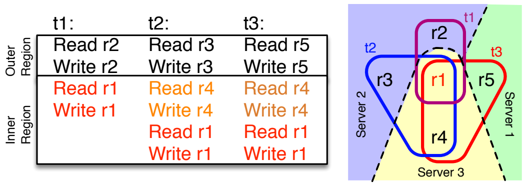
*© ACM 2021. This is a minor revision of the work published in SIGMOD'20, ISBN978-1-4503-6735-6/20/06, June 14–19, 2020, Portland, OR, USA. DOI: https://doi.org/10.1145/3318464.3389724.



t1:	t2:	t3:
Read r1	Read r1	Read r4
Write r1	Read r3	Write r4
Read r2	Write r1	Read r5
Write r2	Write r3	Write r5
	Read r4	Read r1
	Write r4	Write r1

(a) Original Txn. Execution (b) Distr. Txn. Avoiding Partitioning

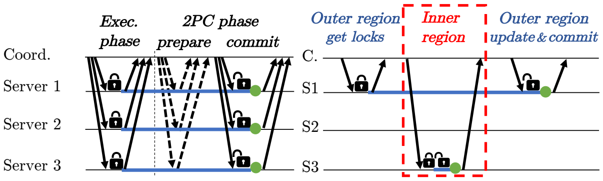
Figure 1: Traditional Execution and Partitioning.



	t1:	t2:	t3:
Outer Region	Read r2	Read r3	Read r5
	Write r2	Write r3	Write r5
Inner Region	Read r1	Read r4	Read r4
	Write r1	Write r4	Write r4
		Read r1	Read r1
		Write r1	Write r1

(a) Re-ordered Txn. Execution (b) Contention-aware Partitioning

Figure 2: Chiller Execution and Partitioning.



(a) 2PC + 2PL (b) Two-region execution

Figure 3: The lifetime of a distributed transaction. The green dots denote when each server releases its locks. The blue lines represent the contention span for each server.

on hot records early and thus reduce the overall contention span on those records.

In summary, we make the following contributions:

(**1**) We propose a new contention-centric partitioning scheme.

(**2**) We present a new distributed transaction execution technique, which aims to update highly-contended records without additional coordination.

(**3**) We show that Chiller outperforms existing techniques by up to a factor of 2 on various workloads.

2. OVERVIEW

2.1 Transaction Processing with 2PL & 2PC

To understand the impact of contention in distributed transactions, let us consider a traditional two-phase locking (2PL) protocol with two-phase commit (2PC). Here, we use transaction t_3 from Figure 1, and further assume that its coordinator is on Server 1, as shown in Figure 3a. The green circle on each partition's timeline shows when it releases its locks and commits. We refer to the time span between acquisition and release of a record lock as the record's *contention span* (depicted by thick blue lines), during which all concurrent accesses to the record would be conflicting. In this example, the contention span for all records is 2 messages long with piggybacking optimization (when merging the last step of execution with the prepare phase) and 4 without it.

2.2 Contention-Aware Transactions

We propose a new partition and execution scheme that aims to minimize the contention span for contended records. The partitioning layout shown in Figure 2b opens new possibilities. As shown in Figure 3b, the coordinator requests locks for all the non-contended records in t_3, which is r_5. If successful, it will send a request to the partition hosting the hot records, Server 3, to perform the remaining part of the transaction. Server 3 will attempt to acquire the lock for its two records, complete the read-set, and perform the transaction logic to check if the transaction can commit. If so, it **commits** the changes to its records.

The reason that Server 3 can unilaterally commit or abort before the other involved partitions receive the commit decision is that Server 3 contains all necessary data to perform the transaction logic. Therefore, the part of the transaction which deals with the hottest records is treated as if it were an independent *local* transaction. This effectively makes the contention span of r_1 and r_4 much shorter (just local memory access, as opposed to at least one network roundtrip).

2.3 Discussion

There are multiple details hidden in the execution scheme presented above. First, after sending the request to Server 3, neither the coordinator nor the other partitions is allowed to abort the transaction; this decision is only up to Server 3. This requirement is very similar to that of H-Store [6], VoltDB [12], and Calvin [13].

Second, for a given transaction, the number of partitions for the inner region has to be *at most* one. Otherwise, multiple partitions cannot commit independently without coordination. This is why executing transactions in this manner requires a new partitioning scheme to ensure that contended records that are likely to be accessed together are co-located.

Finally, our execution model needs to have access to the transaction logic in its entirety to be able to re-order its operations. Our prototype achieves this by running transactions through invoking stored procedures, though it can be realized by other means such as implementing it as a query compiler (similar to Quro [15]). The main alternative model, namely interactive transactions, in which there may be multiple back-and-forth rounds of network communication between a client application and the database is extremely unsuitable for applications that deal with contended data, because all locks and latches have to be held for the entire scope of the client interaction [14].

3. TWO-REGION EXECUTION

The goal of the two-region execution is to minimize the duration of locks on contended records. To achieve this, the execution engine re-orders operations into cold (outer region) and hot operations (inner region); the outer region is executed as normal. If successful, the records in the inner region are accessed. The inner region *commits* upon completion without coordinating with the other participants.

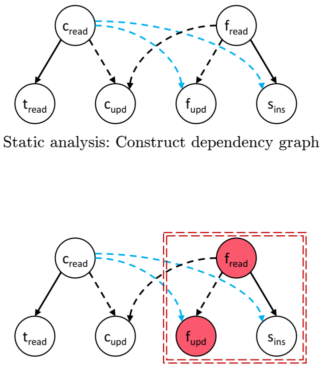
To explain the concepts, we will use an imaginary flight-booking transaction shown in Figure 4a. Here, there are four tables: flight, customer, tax, and seats. In this example, if the customer has enough balance and the flight has an available seat (line 12), a seat is booked and the ticket fee plus state-tax is deducted from their account (lines 14–16). Otherwise, the transaction aborts (line 19).

3.1 Constructing a Dependency Graph

We now describe how we extract the constraints in re-ordering operations from the transaction logic and model it as a dependency graph.

```
1  // input: flight_id, cust_id
2  // desc: reserve a seat in the flight
3           and deduct the ticket fee from
4           customer's balance.
5  Begin transaction
6    f = read("flight", key:flight_id)
7    c = read("customer", key:cust_id)
8    t = read("tax", key:c.state)
9
10   cost = calculate_cost(f.price, t)
11
12   if (c.balance >= cost AND f.seats > 0){
13     seat_id = f.seats
14     update(f, f.seats ← f.seats - 1)
15     update(c, c.balance ← c.balance - cost)
16     insert("seats", key:[flight_id, seat_id],
17                     value: [cust_id, c.name])
18   }
19   else abort
20 End transaction
```

(a) Original transaction



(b) Static analysis: Construct dependency graph

(c) Step 1&2: Select the inner host, if exists

```
// input: cust_id
Begin Outer Region – Phase 1
  c = read_with_wl("customer", key:cust_id)
  t = read_with_rl("tax", key:c.state)
End Outer Region – Phase 1
```

(d) Step 3: Read records in the outer region

```
// input: flight_id, tax t, customer c
Begin Inner Region
  f = read("flight", key:flight_id)
  cost = calculate_cost(f.price, t)
  if (c.balance >= cost AND f.seats > 0)
    seat_id = f.seats
    update(f, f.seats ← f.seats - 1)
    insert("seats", key:[flight_id, seat_id])
  else abort
End Inner Region
```

(e) Step 4: Execute and "**commit**" the inner region

```
// input: customer c, cost
Begin Outer Region – Phase 2
  update(c, c.balance ← c.balance - cost)
End Outer Region – Phase 2
```

(f) Step 5: Commit the outer region

Figure 4: Two-region execution of a ticket purchasing transaction. In the dependency graph, primary key and value dependencies are shown in solid and dashed lines, respectively (blue for conditional constraints, e.g., an "if" statement). Assuming that the flight record is contended (red circles), the red box in (c) shows the operations in the inner region (Step 4). The rest of the operations will be performed in the outer region (Steps 3 and 5).

There may be constraints on data values that must hold true (e.g., seat availability). Furthermore, operations in a transaction may have dependencies among each other. The goal is to reflect such constraints in the *dependency graph*. Here, we distinguish between two types of dependencies. A *primary key dependency (pk-dep)* is when accessing a record r_2 can happen only after accessing record r_1, as the primary key of r_2 is only known after r_1 is read (e.g., the read operation for the tax record in line 8 must happen after the read operation for the customer record on line 7). In a *value dependency (v-dep)*, the new values for a record r_2 are known only after accessing r_1 (e.g., the update operation on line 15). pk-deps determine the constraints in re-ordering operations, while v-deps do not.

Each operation corresponds to a node in the dependency graph. There is an edge from node n_1 to n_2 if the corresponding operation of n_2 depends on that of n_1. The dependency graph for our running example is shown in Figure 4b. For example, the insert operation on line 16 (s_{ins} in the graph) has a pk-dep on the read operation on line 6 (f_{read}), and has a v-dep on the read operation on line 7 (c_{read}). This means that obtaining the lock for the insert query can only happen after the flight record is read (pk-dep), but can happen before the customer is read (v-dep).

3.2 Run-Time Decision

Given the dependency graph, we describe step by step how the protocol executes a two-region transaction.

1) Decide on the execution model: First, the algorithm finds the candidate operations for the inner region. An operation can be a candidate if the records accessed by it are marked as contended in the lookup table, and it does not have any pk-dep to other partitions, since if it does, it would make early commit of the inner region impossible. In Figure 4b, if the insert operation s_{ins} belongs to a different partition than f_{read}, the latter cannot be considered for the inner region because there is a pk-dep between them.

Finding the hosting partition of an operation which accesses records by their primary keys is quite straightforward. However, finding this information for operations which access records by non-primary-key attributes may require secondary indexes. In case no such information is available, such operations will not be considered for the inner region.

2) Select the host for inner region: If all candidate operations for the inner region belong to the same host, then it is chosen as the *inner host*, and otherwise, a single host has to be chosen. Currently, we choose the host with the highest number of candidate operations as the inner host.

3) Read records in outer region: The transaction attempts to lock and read the records in its outer region. In our example, an exclusive lock for the customer record and a shared lock for the tax record are acquired. If either of these lock requests fails, the transaction aborts.

4) Execute and commit inner region: Once all locks are acquired for the outer region, the coordinator delegates processing the inner region to the inner host by sending a message with all information needed to execute its part. Having the values for all of the records in the read-set allows the inner host to check if all of the constraints in the transaction are met (e.g., that there are free seats in the flight). This guarantees that if an operation in the outer region results in an abort, it will be detected by the inner host and the entire transaction will abort.

Once all locks are acquired and the transaction logic is checked to ensure that it can commit, the inner host updates the records, replicates its changes to its replicas (Section 5.1), and *commits*. In case any of the lock requests or transaction constraints fails, the inner host aborts the transaction and directly informs the coordinator about its decision. In our example, the update to the flight record is applied, a new record gets inserted into the seats table, the partial transaction commits, and the value for the `cost` variable is returned, as it will be needed to update the customer's balance.

5) Commit outer region: If the inner region succeeds, the transaction is already considered committed and the coordinator must commit all changes in the outer region. In our example, the customer's balance is updated, and the locks are released from the tax and customer records.

3.3 The Need for a New Partitioning

The two-region execution will not be useful if the transaction's hot records reside in different partitions. No matter which partition becomes the inner host, the contended records on the other partitions will observe long contention spans. Therefore, frequently co-accessed hot records must be co-located per transaction. To this end, we present a novel partitioning technique in Section 4.

4. CONTENTION-AWARE PARTITIONING

To fully unfold the potential of our execution model, Chiller must find contention-minimizing horizontal partitioning of data. We will use the transactions in Figure 5 to explain the partitioning idea. The shade of red corresponds to the record hotness (darker is hotter), and the goal is to find two balanced partitions. Existing partitioning tools (e.g. Schism [2]) minimize distributed transactions (Figure 5b). However, such a split would increase the contention span for the records in transaction t_2, because t_2 would have to hold locks on either 3 or 4, and 6 as part of an outer region.

4.1 Overview of Partitioning

To measure the hotness of records, servers randomly sample the transactions' read- and write-sets during execution. These samples are aggregated over a pre-defined time interval by the *partitioning manager* server (PM). PM uses this information to estimate the contention of individual records (Section 4.2). It then creates the graph representation of the workload, which accommodates the requirements for the two-region execution model (Section 4.3). Based on this representation, it uses a graph partitioning tool to partition the records with the objective of minimizing the overall contention of the workload (Section 4.4). Finally, it updates the servers' lookup tables with new partition assignments.

We assume henceforth that the unit of locking is records. However, the same concepts apply to more coarse-grained lock units, such as pages or hash buckets.

4.2 Contention Likelihood

Using the aggregated samples, PM calculates the conflict likelihood for each record. Due to space constraints, we omit the details and only show the final equation that we derived for calculating record contention. We refer the curious readers to our extended published paper [17]. In the following equation, we use $P_c(\rho)$ to refer to the contention likelihood of record ρ.

$$P_c(X_w, X_r) = 1 - e^{-\lambda_w} - \lambda_w e^{-\lambda_w} e^{-\lambda_r}$$

λ_w and λ_r are time-normalized access frequency for reading and writing to the record, respectively. When λ_w is zero, meaning no writes have been made to the record, the contention will be zero, since shared locks are compatible so no conflict is expected. With a non-zero λ_w, higher values of λ_r will increase the contention likelihood due to the conflict of read and write locks.

4.3 Graph Representation

Chiller models workloads quite differently from existing partitioning algorithms, since record contention must be captured in the graph as minimizing the overall contention is the main objective.

As shown in Figure 5c, we model each transaction as a star; at the center is a dummy vertex (referred to as a *t-vertex*, denoted by a square) with edges to all of the records that are accessed by that transaction. Thus, the number of vertices in the graph is $|T| + |R|$, where $|T|$ is the number of transactions and $|R|$ is the number of records.

All edges connecting a given record-vertex (*r-vertex*) to all of its t-vertex neighbors have the same weight. This weight is proportional to the record's contention likelihood. The weight of the edge between an r-vertex and a connected t-vertex reflects how bad it would be if the record were not accessed in the inner region of that transaction.

Applying the contention likelihood formula to our running example and normalizing the weights produces the graph with the edge weights in Figure 5c. Next, we describe how our partitioning algorithm takes this graph as input and generates a partitioning with low contention.

4.4 Partitioning Algorithm

More formally, our goal is to find a partitioning, which minimizes the contention:

$$\min_{S} \sum_{\rho \in R} P_c^{(S)}(\rho)$$

$$s.t.\ \forall p \in S : L(p) \leqslant (1 + \epsilon) \cdot \mu$$

Here, S is a partitioning of the set of records R into k partitions, $P_c^{(S)}(\rho)$ is the contention likelihood of record ρ under partitioning S, $L(p)$ is the load on partition p, $\mu = \frac{\sum_{p \in P} L(p)}{|P|}$ is the average load on each partition, and ϵ is a small constant that controls the degree of imbalance.

Chiller makes use of METIS [7], a graph partitioning tool which aims to find a high-quality partitioning of the input graph with a small cut, while at the same time respecting the constraint of approximately balanced load across partitions.

The interpretation of the partitioning is as follows: A cut edge e connecting a r-vertex v in one partition to a t-vertex t in another partition implies that t will access v in its outer region, thus observing a conflicting access with a probability proportional to e's weight. To put it differently, the partition to which t is assigned determines t's inner host, and all r-vertices assigned to the same partition can be executed in its inner region. Therefore, a split that minimize the total weight of all cut edges also minimizes the contention.

In our example, the sum of the weights of all cut edges (green lines) is 1.3. Transaction t_1 will access record 3 in its inner region as its t-vertex is in the same partition as record 3, while it will access records 1 and 2 in its outer region. Even though the number of multi-partition transactions is increased compared to Figure 5b, this split results in a much lower contention (1.3 for Chiller as opposed to 3.7 for the partitioning Figure 5b).

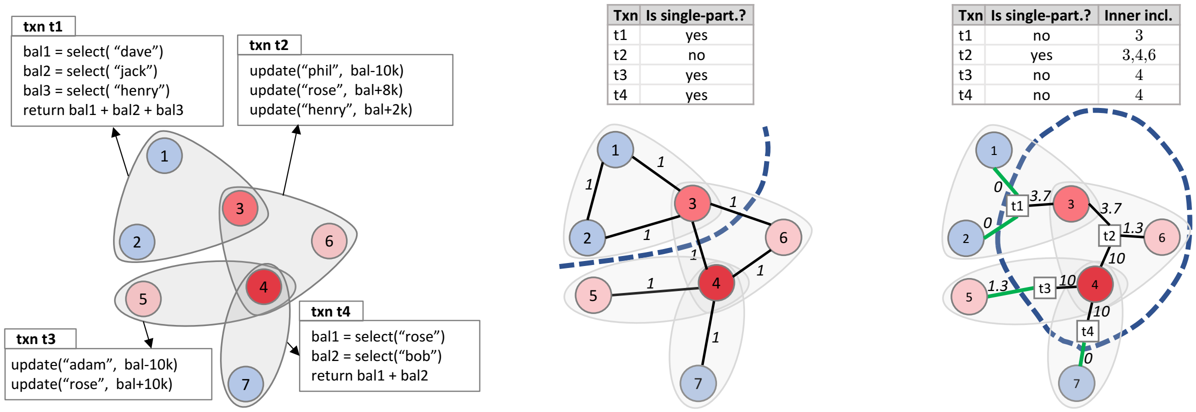
The load L for a partition can be defined in different ways, such as the number of executed transactions, hosting records, or record accesses. The vertex weights depend on the chosen load metric. For the metric of number of executed transactions, t-vertices have a weight of 1 while r-vertices will have a weight of 0.

4.5 Discussion

4.5.1 Scalability of Partitioning.

Chiller has a unique advantage over existing partitioning techniques when it comes to large data sets: it produces graphs with typically significantly fewer edges. Schism, for instance, introduces a total of $n(n-1)/2$ edges for a transaction with n records [2]. However, Chiller's star representation introduces only n edges per transaction, resulting in a much smaller graph. This results in a huge partitioning time improvement.

Furthermore, our approach provides a unique opportunity



txn t1

```
bal1 = select( "dave")
bal2 = select( "jack")
bal3 = select( "henry")
return bal1 + bal2 + bal3
```

txn t2

```
update("phil",  bal-10k)
update("rose",  bal+8k)
update("henry",  bal+2k)
```

txn t3

```
update("adam",  bal-10k)
update("rose",  bal+10k)
```

txn t4

```
bal1 = select("rose")
bal2 = select("bob")
return bal1 + bal2
```

Txn	Is single-part.?
t1	yes
t2	no
t3	yes
t4	yes

Txn	Is single-part.?	Inner incl.
t1	no	3
t2	yes	3,4,6
t3	no	4
t4	no	4

(a) The original workload. The hyperedges show the transactions' boundaries. Darker red indicates higher contention.

(b) Distributed transaction minimization schemes. Edge weights are co-access frequencies.

(c) Contention-centric partitioning. Squares denote transactions. Green edges show outer regions.

Figure 5: An example workload and how partitioning techniques with different objectives will partition it into two parts.

to reduce the size of the lookup table. As we are mainly interested in reducing contention, only records whose contention is above a given threshold can be put in the lookup table. Hence, the lookup table needs to store the information for only those hot records. The other records can be partitioned using hash or range functions, which takes no lookup table space. Please refer to our published work to see the benefits of using this technique in action [17].

4.5.2 *Re-Partitioning*

While the process described in Section 4.1 can be done periodically for the purpose of re-partitioning, our current prototype is based on an offline implementation of the Chiller partitioner. We envision that the offline re-partitioning scheme would be sufficient for many workloads. For other workloads with more frequently changing hot spots, however, it is possible that constantly relocating records in an incremental fashion is more effective.

4.5.3 *Minimizing Distributed Transactions.*

In order to co-optimize for contention and distributed transactions, one only needs to assign a minimum positive weight to all edges in the graph. The bigger the minimum weight, the stronger the objective to co-locate records from the same transaction. While co-optimization is still relevant even in fast RDMA-enabled networks due to higher latency of remote access, we argue that minimizing distributed transactions is just a secondary optimization, as the optimal partitioning objective should shift in the direction of minimizing contention.

5. FAULT TOLERANCE

The two-region execution model modifies 2PC for transactions accessing contended records. A transaction is considered committed once its processing is finished by the inner host, after which, it *must* be able to commit on the other participants even under failures. Chiller employs write-ahead logging to non-volatile memory. However, similar to 2PC, while logging enables crash recovery, it does not provide high availability. The failure of the inner host may sacrifice availability, since the coordinator would not know if the inner region has already committed or not. To achieve high availability, Chiller relies on a new replication method based on synchronous log-shipping, described below.

5.1 Replication Protocol

Since in Chiller, the transaction commit point (i.e., when the inner region commits) happens *before* the outer region participants commit their changes, the inner region replication cannot be postponed until the end of the transaction, as otherwise its changes may be lost if the inner host fails.

To solve this problem, Chiller employs two different algorithms for the replication of the inner and outer regions. The changes in the outer region are replicated as normal—once the coordinator finishes performing the operations in the transaction, it replicates the changes to the replicas of the outer region before making the changes visible. The inner region replication, however, must be done *before the transaction commit point*, so that the commit decision will survive failures. Below, we describe the inner region replication in terms of the different roles of the participants:

Inner host: As shown in Figure 6, when the inner host finishes executing its part, it sends an RPC message to its replicas containing the new values of its records, the transaction read-set, and the sequence ID of the replication message. It then waits for the acks from its NIC hardware, which guarantee that the messages have been successfully sent to the replicas. Finally, it safely commits its changes.

Inner host replicas: Each replica applies the updates in the message in the sequence ID order. This guarantees that the data in the replicas synchronously reflect the changes in the primary inner host partition. When updates are applied, each replica notifies the *original coordinator* of the transaction, as opposed to responding back to the inner host.

Coordinator: The coordinator is allowed to resume the transaction only after it has received the notifications from of all the replicas of the inner host.

5.2 Failure Recovery

The recovery procedure is as follows: First, each partition p probes its local log, and compiles a list of pending transactions on p. For each transaction, its coordinator, inner host, and the list of outer region participants are retrieved, and are then aggregated at a designated node to create a global list of pending transactions. Below, we discuss possible failure scenarios for a pending two-region transaction

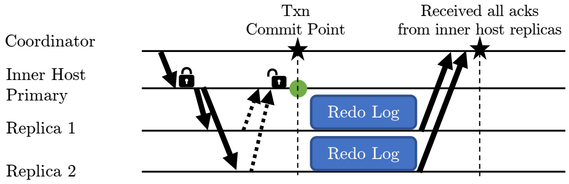


Figure 6: Replication algorithm for the inner region.

along with how the fault tolerance is achieved.

Failure of inner host: If none of the surviving replicas of a failed inner host have received the replication message, the transaction can be safely aborted, because it indicates that the inner host has not committed either. However, if at least one of its replicas has received such a message, that transaction *can* commit, even though that the transaction's updates might have not yet been replicated on all of the replicas. In this case, the coordinator finishes the transaction on the remaining inner host replicas and commits.

Failure of coordinator: If a node is found to be the inner host (or one of its replicas, in case the inner host has also failed), it will be elected as the new coordinator, since it already has the values for the transaction read-set. Otherwise, the transaction can be safely aborted because its changes have not yet received/committed by its inner host.

Failure of an outer region participant: If the failure of participant i happens before the coordinator initiates the inner region, then the transaction is safely aborted. Otherwise, one of i's replicas which has been elected as the new primary will be used to take over i's role in the transaction.

For a sketch of a proof of correctness, we refer the reader to our published work [17].

6. EVALUATION

We evaluated our system to answer two main questions:

(**1**) How does Chiller and its two-region execution model perform under various levels of contention compared to existing techniques?

(**2**) Is the contention-aware data partitioning effective in producing results that can efficiently benefit from the two-region execution model?

6.1 Setup

The test bed we used for our experiments consists of 7 machines connected to a single InfiniBand EDR 4X switch using a Mellanox ConnectX-4 card. Each machine has 256GB RAM and two Intel Xeon E5-2660 v2 processors with 2 sockets and 10 cores per socket. In all experiments, we use only one socket per machine to which the NIC is directly attached. The machines run Ubuntu 14.04 Server Edition as their OS and Mellanox OFED 3.4-1 driver for the network.

6.2 Baselines

We compare the two-region execution scheme against the following commonly used concurrency control (CC) models:

Two-Phase Locking (2PL): we implemented two widely used variants of distributed 2PL with deadlock prevention. In `NO_WAIT`, the system aborts a transaction once it suspects that there is a possibility of deadlock. Therefore, waiting for locks is not allowed. In `WAIT_DIE`, transactions are assigned unique timestamps before execution. An older transaction is allowed to wait for a lock that is owned by a younger transaction, and otherwise it aborts. Timestamp ordering ensures that no deadlock is possible.

Optimistic (OCC): In `OCC`, each participant verifies that its read-set has not been modified by some other transaction. The coordinator commits a transaction only if all the participants pass the validation phase. We based our implementation on the MaaT protocol [8], which is an efficient and scalable algorithm for `OCC` in distributed settings.

In addition, we evaluate two common partitioning schemes: **Hash-partitioning** is the method of assigning records to partitions based on the hash value of their primary key(s). **Schism** is the most notable automatic partitioning technique. It first uses METIS to find a small cut of the workload graph, then compares this record-level partitioning to both a decision tree-learned range partitioning and a simple hash partitioning, and picks the one which results in the minimum number of distributed transactions, or if equal, requires a smaller lookup table. We include the results for different CC schemes for Schism partitioning, and report only `NO_WAIT` for hash partitioning as a simple baseline.

6.3 Workloads

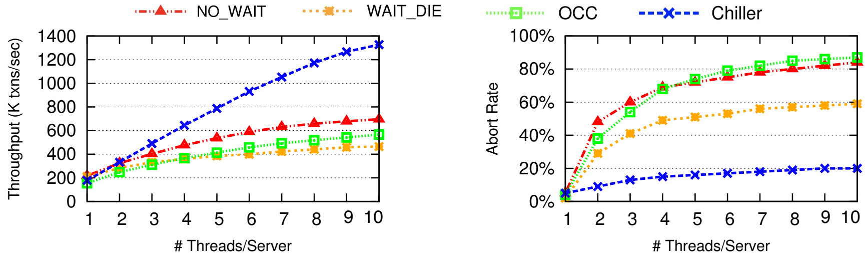
We extensively evaluated Chiller using different workloads. In this paper, we show the results for a subset of our experiments on TPC-C and InstaCart. We refer the reader to our published paper [17] for a more comprehensive experimental evaluation, including more experiments on these two workloads and also various YCSB experiments.

TPC-C: It is the de facto standard for evaluating OLTP systems. Despite being highly partitionable, it contains two severe contention points: the *warehouse* table, and the *district* table. We used one warehouse per server (i.e., 7 warehouses in total) which translates to a high contention workload. As common in all TPC-C evaluations, all tables are partitioned by warehouse ID, except for the Items table which is read-only and is therefore replicated on all servers. Both Chiller and Schism produce this partitioning given the workload trace. Therefore in the following experiments, we mainly focus on the two-region execution feature of Chiller, and evaluate it against the other CC schemes.

InstaCart: To assess the effectiveness of our approach to deal with difficult to partition workloads, we used a real-world data set released by Instacart [4], which is an online grocery delivery service. The dataset contains over 3 million grocery orders for around 50K items from more than 200K customers. On average, each order contains 10 grocery products purchased in one transaction. To model a transactional workload based on the Instacart data, we used the TPC-C's `NewOrder` where each transaction reads the stock values of a number of items, subtracts each one by 1, and inserts a new record in the order table. However, instead of randomly selecting items according to the TPC-C specification, we used the actual Instacart data set. Unlike TPC-C, this data set is actually difficult to partition due to the nature of grocery shopping, where items from different categories (e.g., dairy, produce, and meat) may be purchased together. There is also a significant skew in the number of purchases of different products (e.g. 15% of transactions contain banana).

6.4 TPC-C Results

We first measure the performance of Chiller, `NO_WAIT`, `WAIT_DIE`, and `OCC` with increasing number of worker threads per server. For this experiment, we use TPC-C workload. Although increasing the number threads provides more CPU power to process transactions, it also increases the con-



(a) Throughput.

(b) Abort rate.

Figure 7: Comparison of different concurrency control methods and Chiller for TPC-C.

tention. Studying this factor is therefore of great importance since many modern in-memory databases are designed for systems with multi-core CPUs.

As Figure 7a shows, with only one worker thread per machine, NO_WAIT and WAIT_DIE perform similarly, and have 10% higher throughput than Chiller. This is accounted for by the two-region execution overhead. However, increasing the number of worker threads also raises the chance of conflicts, negatively impacting the scalability of 2PL and OCC. Chiller, on the other hand, minimizes the lock duration for the two contention points in TPC-C (warehouse and district records) and thus, scales much better. With 10 threads, the throughput of Chiller is 2× and 3× higher than that of NO_WAIT and WAIT_DIE, respectively.

Figure 7b shows the corresponding abort rates (averaged over all threads). With more than 4 threads, OCC's abort rate is even higher than NO_WAIT, which is attributed to the fact that many transactions are executed to the validation phase and are then forced to abort. Compared to the other techniques, the abort rate of Chiller increases much more slowly as the level of concurrency per server increases. This experiment shows the inherent scalability issue with traditional CC schemes when deployed on multi-core systems, and how Chiller manages to significantly alleviate it.

6.5 InstaCart Results

We analyzed the benefits of combining the Chiller's partitioning scheme with the two-region execution model by using a real-world Instacart workload (as introduced in Section 6.3), which is much harder to partition than TPC-C.

In order to understand the effectiveness of the two-region execution, we compare full Chiller (Chiller) to Chiller partitioning without the two-region execution model (ChP) and Chiller partitioning using Quro* (ChP+Quro*). In contrast to ChP which does not re-order operations, ChP+Quro* re-orders operations using Quro [15], which is a recent contention-reduction technique for centralized database systems. Moreover, we compare full Chiller to two other non-Chiller baselines (Hash-partitioning and Schism-partitioning). For both ChP and ChP+Quro* as well as the non-Chiller baselines (Hash and Schism), we only show the results for a WAIT_DIE scheme as it yielded the best throughput compared to NO_WAIT and OCC for this experiment.

As Figure 8 shows, compared to the Hash-partitioning baseline (black line), both ChP and ChP+Quro* (green and red lines) have significantly higher throughput. We found that this is not because the Chiller partitioning reduces the number of distributed transactions, but rather because contended records which are accessed together are co-located, reducing the cost of aborts. More specifically, if a transaction on contented records needs to be aborted, it only takes one round-trip, leading to an overall higher throughput since the failed transaction can be restarted faster.

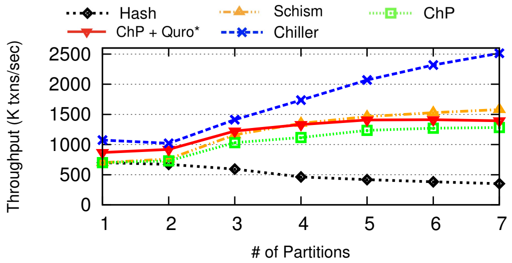


Figure 8: Instacart with different execution models.

Furthermore, we see that ChP+Quro*, which re-orders operations to access the low contended records first, initially increases the throughput by 20% compared to ChP but then its advantage decreases as the number of partitions increases. The reason for this is that the longer latency of multi-partition transactions offsets most of the benefits of operation re-ordering if the commit order of operations remains unchanged. In fact, with 5 partitions, Schism (yellow line) starts to outperform ChP+Quro*, even though Schism does not leverage operation re-ordering.

In contrast to these baselines, Chiller (blue line) not only re-orders operations but also splits them into an inner and outer region, thus outperforms all the other techniques. For the largest cluster size, the throughput of Chiller is approximately 1 million txns/sec higher than the second best baseline. This experiment clearly shows that the contention-centric partitioning must go hand-in-hand with the two-region execution to be most effective.

7. RELATED WORK

Data Partitioning: A large body of work exists for partitioning OLTP workloads with the ultimate goal of minimizing cross-partition transactions. Most notably, Schism [2] is an automatic partitioning tool that uses the workload trace to model the relationship between the database records as a graph, and then applies METIS [7] to find a small cut while approximately balancing the number of records among partitions. Clay [10] builds the same workload graph as Schism, but instead takes an incremental approach to partitioning by building on the previously produced layout. All of these methods share their main objective of minimizing inter-partition transactions. In the age of new networks and much "cheaper" distributed transactions, such an objective is no longer optimal.

Transaction Decomposition: There has also been work on the theory of transaction chopping [11, 18], in which a transaction gets split into smaller pieces, with each piece being an independent transaction. In contrast to transaction chopping, our two-region execution not only splits a transaction into cold and hot operations, but re-orders operations based on which region they belong to. Also, we do not treat the outer region as an independent transaction and will hold the locks on its records until the end of the transaction. This allows us to abort a transaction later in the inner region.

Determinism and Contention-Reducing Execution: Another line of work aims to reduce contention through enforcing determinism to the concurrency control (CC) unit. Most notably, Calvin [13] uses a global agreement scheme to deterministically sequence the lock requests. Deterministic execution requires *a priori* knowledge of read-set and write-set.

Most related to Chiller is Quro [15], which also re-orders operations inside transactions in a centralized DBMS with 2PL to reduce the lock duration of contended data. However, unlike Chiller, the granularity of contention for Quro is tables, and not records. Furthermore, almost all these works deal with single-node DBMSs and do not have the notion of distributed transactions, 2PC, or asynchronous replication on remote machines, and hence finding a good partitioning scheme is not within their scope.

Transactions over Fast Networks: This paper continues the growing focus on distributed transaction processing on new RDMA-enabled networks [1]. The increasing adoption of these networks by key-value stores [9] and DBMSs [3, 16, 5] is due to their much lower overhead for message processing using RDMA features, low latency, and high bandwidth. The common promise of these systems is better scalability by imposing far less overhead for cross-partition transactions. Therefore, Chiller's two-region execution and its contention-centric partition are specifically suitable for this class of distributed data stores.

8. CONCLUSIONS

This paper presents Chiller, a distributed transaction processing and data partitioning scheme that aims to minimize contention. Chiller is designed for fast RDMA-enabled networks, where the cost of distributed transactions is already low, and the system's scalability depends on the absence of contention in the workload. Chiller partitions the data such that the hot records that are likely to be accessed together are placed in the same partition. Using a novel two-region processing approach, it then executes the *contended* part of a transaction separately from the *un-contended* part. Chiller can significantly outperform existing approaches under workloads with varying degrees of contention.

9. ACKNOWLEDGEMENT

This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, gifts from Mellanox and Huawei, as well as NSF IIS Career Award 1453171, NSF CAREER Award #CCF-1845763, DOE Early Career Award #DE-SC0018947, and grant BI2011/1 of the German Research Foundation (DFG).

10. REFERENCES

[1] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[2] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *VLDB Endowment*, 3(1-2):48–57, 2010.

[3] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM SOSP*, pages 54–70, 2015.

[4] Instacart. The instacart online grocery shopping dataset 2017, 2017.

[5] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX OSDI*, pages 185–201, 2016.

[6] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *VLDB Endowment*, 1(2):1496–1499, 2008.

[7] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[8] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *VLDB Endowment*, 7(5):329–340, 2014.

[9] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[10] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: fine-grained adaptive partitioning for general database schemas. *VLDB Endowment*, 10(4):445–456, 2016.

[11] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3):325–363, Sept. 1995.

[12] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[13] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *ACM SIGMOD*, pages 1–12, 2012.

[14] A. G. Thomson. *Deterministic Transaction Execution in Distributed Database Systems*. PhD thesis, Yale University, 2013.

[15] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. *VLDB Endowment*, 9(5):444–455, 2016.

[16] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. *VLDB Endowment*, 10(6):685–696, 2017.

[17] E. Zamanian, J. Shun, C. Binnig, and T. Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *ACM SIGMOD*, page 511–526, 2020.

[18] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *ACM SOSP*, page 276–291, 2013.