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

Erfan Zamanian
erfanz@cs.brown.edu
Brown University

Julian Shun
jshun@mit.edu
MIT CSAIL

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

Tim Kraska
kraska@csail.mit.edu
MIT CSAIL

Abstract

Distributed transactions on high-overhead TCP/IP-based networks were conventionally considered to be prohibitively expensive and thus were avoided at all costs. To that end, the primary goal of almost any existing partitioning scheme 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 However, with the new generation of fast RDMA-enabled networks, this assumption is no longer valid.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 fact, recent work has shown that distributed databases can scale even when the majority of transactions are cross-partition.

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. Finally, we evaluate Chiller using various workloads, and show that our partitioning and execution strategy outperforms traditional partitioning techniques which try to avoid distributed transactions, by up to a factor of 2.

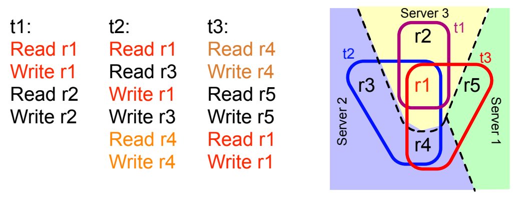
ACM Reference Format:
Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389724

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 [8, 28, 29, 34, 37, 44]. Yet, a recent result [43] 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. In this paper, we argue that the new optimization goal should be to minimize contention rather than distributed transactions.

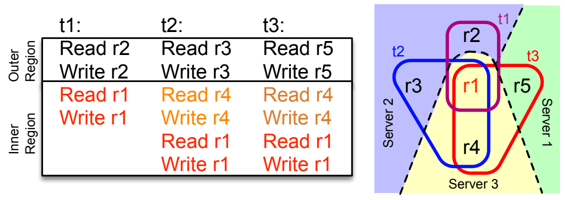
In this paper, we present Chiller, a new partitioning scheme and execution model based on 2-phase-locking which aims to minimize contention. Chiller is based on two complementary ideas: **(1) a novel commit protocol** based on re-ordering transaction operations with the goal of minimizing the lock duration 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. 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



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

Figure 1: Traditional Execution and Partitioning.



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

Figure 2: Chiller Execution and Partitioning.

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 increase the total transaction throughput. The idea is that re-ordering the transaction operations minimizes the lock duration for the "hot" items and subsequently the chance of conflicting with concurrent transactions. More importantly, after the re-ordering, the success of a transaction relies entirely on the success of acquiring the lock for the most contended records. That is, if a distributed transaction has already acquired the necessary locks for all non-contended records (referred to as the *outer region*), the commit outcome depends solely 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 [8]), 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 [40]). By taking advantage of the co-location of hot records, Chiller's execution scheme reorders operations such that it can release locks on hot records early and thus reduce the overall contention span on those records. As we will show, these two complementary ideas together provide significant performance benefits over existing state-of-the-art approaches on various workloads.

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

The throughput of distributed transactions is limited by three factors: (1) message overhead, (2) network bandwidth, and (3) increased contention [3]. The first two limitations are significantly alleviated with the new generation of high-speed RDMA-enabled networks. However, what remains is the increased contention likelihood, as message delays are still significantly longer than local memory accesses.

2.1 Transaction Processing with 2PL & 2PC

To understand the impact of contention in distributed transactions, let us consider a traditional 2PL with 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.

While our example used 2PL, other concurrency control (CC) methods suffer from this issue to various extents [16]. For example in OCC, transactions must pass a validation phase before committing. If another transaction has modified the data accessed by a validating transaction, it has to abort and all its work will be wasted [9, 16].

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.

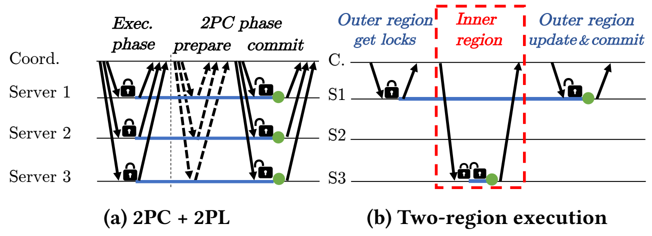


Figure 3: The lifetime of a distributed transaction. The green dots denote when each server releases its locks. The blue lines represent the contention span for each server.

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 and simplifications hidden in the execution scheme presented above.

First, after sending the request to Server 3, neither the coordinator nor the rest of the partitions is allowed to abort the transaction and this decision is only up to Server 3. For this reason, our system currently does not support triggers, which may cause the transaction to abort at any arbitrary point. In that matter, its requirement is very similar to that of H-Store [21], VoltDB [33], Calvin [35] and MongoDB [2]. Also, the required determinism to disallow transactions to abort after a certain point in their life cycles is realized through the combination of Chiller's novel execution, replication and recovery protocols, which will be discussed in Section 5.

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 [40]). Due to the low overhead of our re-ordering algorithm, ad-hoc transactions can also be supported, as long as all operations of a transaction are issued in one shot. 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 yet demand high throughput, because the database cannot reason about the boundaries of transactions upfront, and therefore all locks and latches have to be held for the entire scope of the client interaction which may last multiple roundtrips [36].

3 Two-region Execution

3.1 General Overview

The goal of the two-region execution scheme is to minimize the duration of locks on contended records by postponing their lock acquisition until right before the end of the expanding phase of 2PL, and performing their lock release right after they are read/modified, without involving them in the 2PC protocol. More specifically, the execution engine re-orders operations into cold operations (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 important point is that the inner region *commits* upon completion without coordinating with the other participants. Because of the way that the inner region is not involved in 2PC, fault tolerance requires a complete re-visit, otherwise many failure scenarios may sacrifice the system's correctness or liveness. Until we discuss our fault tolerance algorithm in Section 5, we present the execution and partitioning schemes under a no-failure assumption.

To help 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 (lines 14 and 16) and the ticket fee plus state-tax is deducted from their account (line 15). Otherwise, the transaction aborts (line 19).

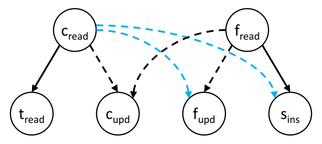
The remainder of this section is structured as follows. Section 3.2 describes how we extract the constraints in re-ordering operations from the transaction logic and model it as a dependency graph. Using this graph, a five-step protocol, described in Section 3.3, is used to execute a two-stage transaction. Finally, in Section 3.4, we present optimizations, and look at challenges for the protocol to be correct and fault tolerant. Our solution to these challenges is then presented throughout the subsequent sections.

3.2 Constructing a Dependency Graph

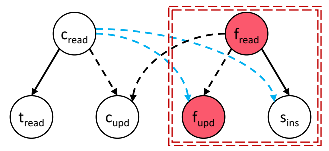
There may be constraints on data values that must hold true (e.g., availability of a flight seat). Furthermore, operations in a transaction may have dependencies among each other. The goal is to reflect such constraints in the *dependency graph*. Here, we distinguish between two types of dependencies. 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

```
 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 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).

read operation for the customer record in line 7). In a *value dependency (v-dep)*, the new values for the update columns of a record r_2 are known only after accessing r_1 (e.g. the update operation in line 15). We are only concerned about the pk-deps, and not the v-deps, since v-deps do not restrict the order of lock acquisition, while pk-deps do put restrictions on which re-orderings are possible.

Each operation of the transaction 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 in line 16 (s_{ins} in the graph) has a pk-dep on the read operation in line 6 (f_{read}), and has a v-dep on the read operation in line 7 (c_{read}). This means that getting the lock for the insert query can only happen after the flight record is read (pk-dep), but it can happen before the customer is read (v-dep). Please refer to the figure's caption for the explanation of the color codes.

3.3 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, it must find the list of candidate operations for the inner region. An operation can be a candidate if the record(s) accessed by it is marked as contended in the lookup table, and it does not have any pk-dep on operations on 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*, otherwise, one has to be picked. 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 have been acquired for the records in 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 (e.g., transaction ID, input parameters, etc.). 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., there are free seats in the flight). This guarantees that if operation in the outer region should result in an abort, it will be detected by the inner host and the entire transaction will abort.

Once all locks are successfully acquired and the transaction logic is checked to ensure the transaction 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 by another partition.

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 gets updated, and the locks are released from the tax and customer records.

3.4 Challenges

First, it will not be useful if the hot records of a transaction are scattered across different partitions. No matter which partition becomes the inner host, the other contended records will observe long contention spans. Therefore, frequently co-accessed hot records must be co-located. To this end, we present a novel partitioning technique in Section 4. Second, the inner host removes its locks earlier than the other participants (steps 4 and 5). For this reason, fault tolerance requires a revisit, which will be presented in Section 5.

4 Contention-aware Partitioning

To fully unfold the potential of the two-region execution model, the objective of our proposed partitioning algorithm is to find a horizontal partitioning of the data which minimizes the contention. To better explain the idea, we will use 4 transactions shown in Figure 5. The shade of red corresponds to the record hotness (darker is hotter), and the goal is to find two balanced partitions (for now, we define "balanced" as a partitioning that splits the set of records in half). Existing partitioning schemes, such as Schism [8] minimize distributed transactions, as shown in Figure 5b. However, such a split would increase the contention span for records 3 or 4, and 6 in transaction t_2, because t_2 will have to hold locks on either 3 or 4, and 6 as part of an outer region.

Not only the main objective of our proposed partitioning is different than the one in existing techniques, but also their commonly used graph representation of the workload (with records as vertices and co-accesses as edges) does not capture the essential requirements of our problem, that is, the distinction of inner and outer region and differences in their execution. This necessitates a new workload representation.

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 then 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 servers' lookup tables with new partition assignments.

We assume henceforth that the unit of locking and partitioning 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. More specifically, we define the probability of a conflicting access for a given record as:

$$P_c(X_w, X_r) = \overbrace{P(X_w > 1)P(X_r = 0)}^{(i)} + \overbrace{P(X_w > 0)P(X_r > 0)}^{(ii)}$$

Here, X_w and X_r are random variables corresponding to the number of times a given record is read or modified within the lock window, respectively. The equation consists of two terms to account for the two possible conflict scenarios: (i) write-write conflicts, and (ii) read-write conflicts. Since (i) and (ii) are disjoint, we can simply add them together.

Similar to previous work [23, 41], we model X_w (X_r) using a Poisson process with a mean arrival time of λ_w (λ_r), which is the time-normalized access frequency. This allows us to rewrite the above equation as follows:

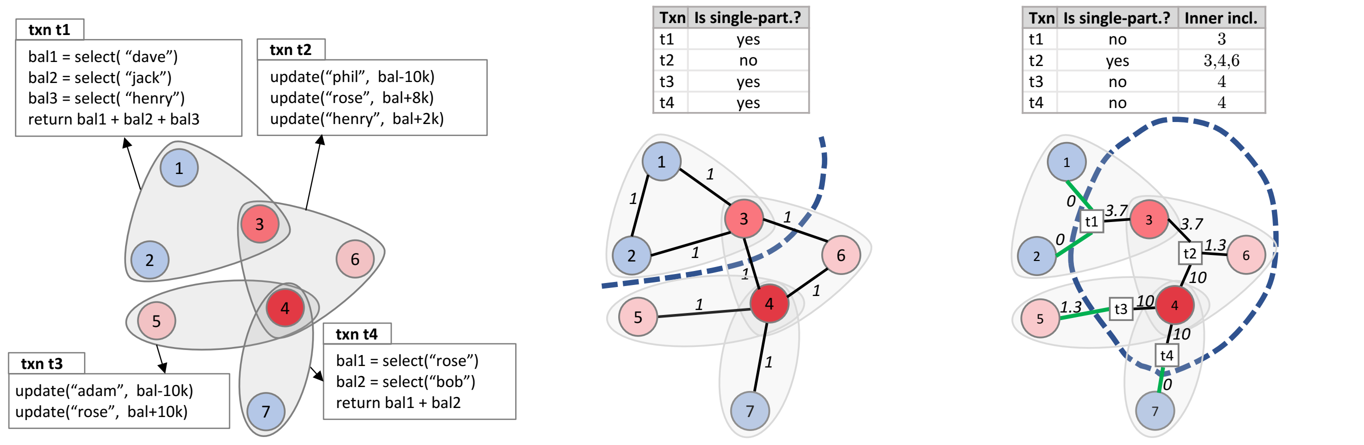
$$\begin{aligned} P_c(X_w, X_r) &= \overbrace{\left(1 - (\frac{{\lambda_w}^0 e^{-\lambda_w}}{0!} + \frac{{\lambda_w}^1 e^{-\lambda_w}}{1!})\right)\left(\frac{{\lambda_r}^0 e^{-\lambda_r}}{0!}\right)}^{(i)} \\ &+ \overbrace{\left(1 - \frac{{\lambda_w}^0 e^{-\lambda_w}}{0!}\right)\left(1 - \frac{{\lambda_r}^0 e^{-\lambda_r}}{0!}\right)}^{(ii)} \\ &= 1 - e^{-\lambda_w} - \lambda_w e^{-\lambda_w} e^{-\lambda_r} \end{aligned}$$

The contention likelihood is defined per lock unit. We use $P_c(\rho)$ to refer to the contention likelihood of record ρ. In the equation above, when λ_w is zero, meaning no write has been made to the record, $P_c(\rho)$ 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

The are three key properties that a graph representation of the workload should have to properly fit in the context of our execution model. First, record contentions must be captured in the graph as this is the main objective. Second, the relationship between records must also be modeled, due to the requirement that there can be only one inner region for a transaction, and hence the frequently co-accessed contended records should be co-located. Third, the final partitioning should also make it possible to determine the inner region for each transaction. Therefore, Chiller models the workload quite differently than existing partitioning algorithms.

As shown in Figure 5c, we model each transaction as a star; at the center is a dummy vertex (referred to as *t-vertex*,



```
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 hyper edges show the transactions' boundaries. Darker red indicates more contended.

(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.

denoted by squares) with edges to all 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. The number of edges will be the sum of the number of records involved per transaction.

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 is not accessed in the inner region of that transaction.

Note that while our graph representation does not directly incorporate dependencies among operations (e.g., *pk-dep*), it should not take long for a running system until the partitioning algorithm would automatically build edges between records frequently accessed as part of these operations.

Applying the contention likelihood formula to our running example and normalizing the weights will produce the graph with the edge weights in Figure 5c. Note that there is no edge between any two records. Co-accessing records is implied by a common t-vertex connecting them. Next, we describe how our partitioning algorithm takes this graph as input and generates a partitioning with low contention.

4.4 Partitioning Algorithm

As we are able to model contention among records using a weighted graph, we can apply standard graph partitioning algorithms. 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, μ is the average load on each partition, and ϵ is a small constant that controls the degree of imbalance. Therefore, $\mu = \frac{\sum_{p \in P} L(p)}{|P|}$. The definition of load will be discussed shortly.

Chiller makes use of METIS [22], 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, and 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 the inner host of t; all r-vertices assigned to the same partition can be executed in the inner region of t. As a result, 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 (which are portrayed as 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 the partitioning in Figure 5b, this split results in a much lower contention.

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 depends 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. The weighting is reversed for the second metric. METIS generates a partitioning such that the sum of vertex weights in each partition is approximately balanced.

4.5 Discussion

4.5.1 Scalability of Partitioning. There are two issues every partitioning scheme has to address: (1) the graph size and the cost of partitioning it, and (2) the size of the lookup table.

It is time- and computation-intensive to partition very large graphs. However, Chiller has a unique advantage over existing partitioning techniques: it produces graphs with significantly fewer edges for most workloads. Schism, for instance, introduces a total of $n(n-1)/2$ edges for a transaction with n records [8]. However, Chiller's star representation introduces only n edges per transaction, resulting in a much smaller graph. For example, we found that on average, constructing the workload graph and applying the METIS partitioning tool take up to 5 times longer on Schism compared to Chiller in our experiments.

Furthermore, our approach provides a unique opportunity to reduce the size of the lookup table. As we are mainly interested in reducing contention, we can primarily focus on the records with a contention likelihood above a given threshold. Hence, the lookup table only needs to store where these hot records are located. The other records can be partitioned using an orthogonal scheme (e.g., hash or range), which takes no lookup-table space. We study this technique in more depth using an experiment (Section 7.5.3).

4.5.2 Re-Partitioning and Data Migration. 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. In our experiments, running this algorithm on 100 thousand sampled transactions for a workload with as many as 30 millions records took less than ten minutes on one machine (Section 7.1). This time includes calculating the contention likelihoods, building the workload graph, and partitioning it using METIS. In addition, we found that the pruning techniques proposed above are quite effective in reducing the partitioning time without significantly impacting the throughput. Therefore, we envision that the offline re-partitioning scheme would be sufficient for many workloads, and re-partitioning can be as simple as running the algorithm on one or multiple partitioning managers. For other workloads with more frequently changing hot spots, however, it is possible that constantly relocating records in an incremental way is more effective. Extending this work to support online re-partitioning is an interesting direction of future work.

Another related topic is when data re-partitioning happens, how the system relocates records while still maintaining ACID guarantees. Our current prototype produces a record relocation list, which can be used to move records transactionally (each tuple migration is performed as one individual transaction). As data migration is a general requirement by every production OLTP partitioning tool, there are many automatic tools which perform this task more efficiently [12, 13, 39]. We are planning to extend our prototype to use Squall [12], which is a live data migration tool.

4.5.3 Minimizing Distributed Transactions. It is also possible to co-optimize for contention and distributed transactions using the same workload representation. 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. Such co-optimization is still relevant even in fast RDMA-enabled networks since a remote access through RDMA is about 10× slower than a local access. However, as we argue in this paper, the optimal partitioning objective should shift in the direction of minimizing contention, and therefore minimizing distributed transactions is just a secondary optimization.

5 Fault Tolerance

The two-region execution model presented in Section 3 modifies the typical 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 despite failures. A participant in the outer region cannot unilaterally abort a transaction once it has granted its locks in the outer region and informed the coordinator, since the transaction may have already been committed by the inner host.

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 before sending its decision to the coordinator may sacrifice the availability, since the coordinator would not know if the inner region is already committed or not, in which case it has to wait for the inner host to recover.

To achieve high availability, Chiller relies on a new replication method based on synchronous log-shipping, explained in Section 5.1. Then, we discuss how this replication protocol achieves high availability while still maintaining consistency.

5.1 Replication Protocol

In conventional synchronous log-shipping replication, the logs are replicated before the transaction commits. Since in Chiller, the transaction commit point (i.e., when the inner region commits) happens *before* the outer region participants commit their changes (see Figure 3b), the inner region replication cannot be postponed to the end of the transaction, 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

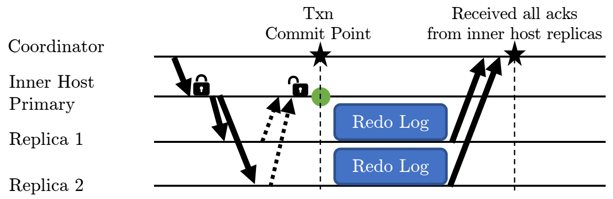


Figure 6: Replication algorithm for the inner region.

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. In case the inner host decides to abort, the replication phase will not be needed and it can directly reply to the coordinator.

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 all updates of the replication message are applied, each replica notifies the *original coordinator* of the transaction, as opposed to responding back to the inner host. This saves one network message delay.

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

In the following, we describe our failure recovery protocol.

5.2 Failure Recovery

For failure detection and membership reconfiguration, Chiller relies on a cluster manager such as Zookeeper [17]. When a machine is suspected of failure, all of the other nodes close their communication channels to the suspected node to prevent it from interfering in the middle of the recovery process.

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, the list of outer region participants are retrieved, and then aggregated at a designated node to make a global list of pending transactions. Below, possible failure scenarios for a pending two-region transaction along with how the fault tolerance is achieved are discussed.

Failure of inner host: If none of the surviving replicas of a failed inner host has 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 it might have not yet replicated on all the replicas. In this case, the coordinator finishes the transaction on the remaining inner host replicas and the outer region participants, 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 is failed too), 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 are 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.

Proof of Correctness — We now provide some intuition on the protocol correctness in terms of safety and liveness.

It is easy to see why the two-region execution model with the described replication protocol maintains safety, since transactions are serialized at the point when their inner host commits. Also, similar to 2PC, if even one participant commits (aborts), no other participant is allowed to abort (commit). This is due to the "no turning back" concept of the commit protocol of the inner region, guaranteeing that all participants will agree on the same decision.

To support liveness, the system first needs to detect failures and repair/replace faulty nodes. For this purpose, Chiller relies on the existence of a fault tolerant coordinator, such as Zookeeper [17] or Chubby [4]. So long as at most f out of $f + 1$ replicas fail, the protocol guarantees liveness by satisfying these two properties:

(1) A transaction will eventually commit at all its participants and their replicas once it has been committed by the inner host: The inner host commits only when its changes are replicated on its replicas. Therefore, there will be at least one replica containing the commit decision which causes the transaction to commit during the recovery process.

(2) A transaction which is not yet committed at its inner host will eventually either abort or commit: If the inner host does not fail, it will eventually process the inner region. However, if it encounters a failure, the transaction will be discovered during the failure recovery, and handled by a new inner host.

6 Implementation

We now briefly present the implementation of the system we used to evaluate Chiller.

In our system, tables are built on top of a distributed hash table, and are split horizontally into multiple partitions, with each *execution server* hosting one partition in its main memory. The unit of locking is a hash bucket, and each hash bucket encapsulates its lock metadata in its header, eliminating the need for a centralized lock manager per partition. Our current implementation performs locking on bucket granularity and does not prevent the phantom problem. However, the core idea of Chiller also works with range locks.

An execution server utilizes all its processing cores through multi-threading, where each execution thread has access to the hosted partition on that machine. To minimize inter-thread communication, each transaction is handled from beginning to end by one execution thread. To extract maximum concurrency, each worker thread employs multiple *co-routine workers*, such that when one transaction is waiting for a network operation, it yields to the next co-routine worker which processes a new transaction. The communication needed for distributed transactions is done either through direct remote memory operations (RDMA Read, Write, and atomic operations), or via RPC messages implemented using RDMA Send and Receive. The access type for each table is specified by the user when creating that table.

Bucket to partition mappings are stored in a lookup table, which is replicated on all servers so that execution threads would know where each record is. It can be either defined by the user in the form of hash or range functions on some table attributes, or produced individually for all or some buckets using the partitioning algorithm. In addition to storing the partition assignments, the lookup table also contains the list of buckets with a contention above a given threshold, which is used to determine the inner region for each transaction.

As discussed in Section 5, to guarantee high availability in the presence of failures, we use log-shipping with 2-safety, where each record is replicated on two other servers. The backup replicas get updated synchronously before the transaction commits on the primary replica. In addition, like other recent high-performance OLTP systems [11, 20, 43], crash recovery is guaranteed by relying on the persistence of execution servers' logs on some form of non-volatile memory (NVM), such as battery-backed DRAM.

7 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?

7.1 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 where the the NIC is directly attached and disable hyper-threading to minimize the variability in measurements caused by same-core threads interference, which is a typical setup also used in other papers [32, 42, 45]. The machines run Ubuntu 14.04 Server Edition as their OS and Mellanox OFED 3.4-1 driver for the network.

7.2 Baselines

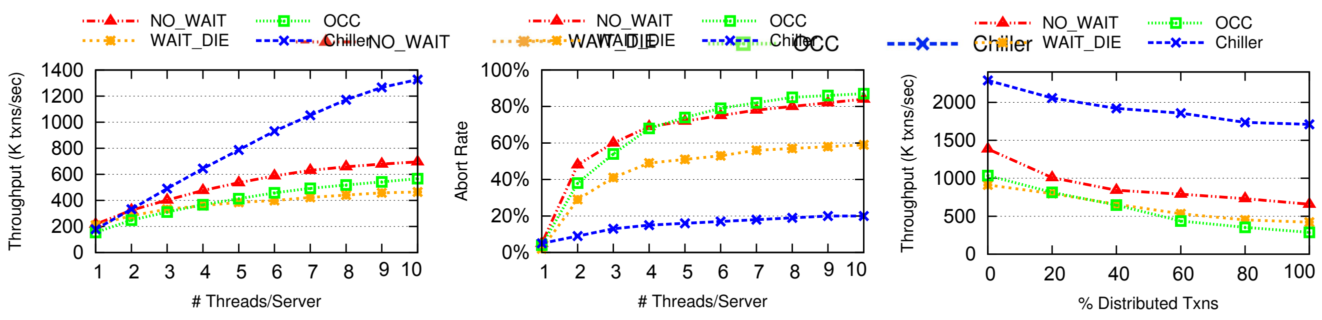
To assess the ability of the two-region execution model in handling contention, we evaluate how it holds up against alternative commonly used concurrency control (CC) models, more specifically these protocols:

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 of a deadlock, i.e., when a record lock request is denied. 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 which is owned by a younger transaction, otherwise it aborts. Timestamp ordering ensures no deadlock is possible. While one could also implement 2PL with deadlock detection, it demands significant network synchronization between servers to detect lock-request cycles, and is therefore very costly in a distributed setting [1, 16]. Therefore, we did not include it in our evaluation.

We based the implementation of Chiller's locking mechanism on `NO_WAIT` due to its lower overhead (no need to manage lock queues), although `WAIT_DIE` could also be used.

Optimistic (OCC): We based our implementation on the MaaT protocol [26], which is an efficient and scalable algorithm for `OCC` in distributed settings [16]. Each transaction is assigned a range for its commit timestamp, initially set to $[0\ \infty)$. Also, the DBMS stores for each record the list of pending reader IDs and writer IDs, and the ID of the last committed transaction which accessed the record. Each time a transaction reads/modifies a record, it modifies its timestamp range to be in compliance with the read/write timestamp of that record, and adds its unique timestamp to the list of the record's read/write IDs. At the end, each participant of the transaction attempts to validate its part by changing the timestamp ranges of the validating transaction and the other conflicting transactions, and votes to commit if the final range of the transaction is valid. The coordinator commits a transaction only if all the participants vote to commit.

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 for OLTP workloads. 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



(a) Throughput for standard TPC-C. **(b) Abort rate for standard TPC-C.** **(c) Impact of distributed transactions.**

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

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.

7.3 Workloads

For our experiments, we use the following workloads and measure throughput as number of committed transactions per second (i.e., excluding aborted transactions).

TPC-C: This is the de facto standard for evaluating OLTP systems. It consists of 9 tables and 5 types of transactions. The majority of transactions access records belonging to a single warehouse. Therefore, the obvious partitioning layout is by warehouse. Despite being highly partitionable, it contains two severe contention points. First, each *new-order* transaction does an increment on one out of 10 records in the district table of a warehouse. Second, every *payment* transaction updates the total balance of a warehouse and one of its 10 districts, creating an even more severe contention point. These two transactions comprise more than 87% of the workload. We used one warehouse per server (i.e., 7 warehouses in total) which translates to a high contention workload. This allows us to focus on the differences in the execution models of Chiller and traditional schemes.

YCSB: It consists of a single table with 1KB records [6]. We generated 5 million records (~ 5 GB) per server. To generate read and write-sets of transactions with a desired level of locality, we used a mapping function from records to partitions. Since the benchmark does not specify transactions, we group multiple read/write-operations into one transaction as discussed next. To explore different aspects of the problem in more depth, we used the following two workloads:

YCSB Local: This workload represents a perfectly partitionable dataset. Each transaction reads and modifies 16 records stored on a single partition using a Zipfian distribution with varying skew factor θ.

YCSB Distributed: Many real OLTP workloads are not as partitionable as *YCSB Local* on the transaction level, but still exhibit some locality on the record level. For example, a purchase that contains one Harry Potter book is likely to contain a few other volumes of the Harry Potter franchise, while still including any other non-related item. To model such cases, we generated a workload where each transaction reads 4 records across different partitions of the entire database uniformly, and reads and modifies 2 other records from a single partition using a Zipfian distribution.

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 [18], which is an online grocery delivery service. The dataset contains over 3 million grocery orders for around 50K items from more than 200K of their customers. On average, each order contains 10 grocery products purchased in one transaction by a customer. 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 the original 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. More importantly, there is a significant skew in the number of purchases of different products. For example, 15% of transactions contain banana.

7.4 TPC-C Results

As common in all TPC-C evaluations, all tables are partitioned by warehouse ID, except for the Items table which is read-only and 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.

7.4.1 Impact of Concurrency Level. We first measure the performance of Chiller, NO_WAIT, WAIT_DIE, and OCC with increasing number of worker threads per server. Although such increase provides more CPU power to process transactions, it also increases the contention. 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 running in each machine (i.e., no concurrent data access), NO_WAIT and WAIT_DIE perform similarly, and has 10% higher throughput than Chiller. This is accounted by the two-region execution overhead. However, as we increase the number of worker threads, the likelihood that transactions conflict with each other increases, 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 then are 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.

7.4.2 Impact of Distributed Transactions. For this experiment, we restricted the transactions to NewOrder and Payment, each making up 50% of the mix (In the standard TPC-C workload, these two transactions are the only ones which can be multi-partition). For Payment, we varied the probability that the paying customer is located at a remote warehouse, and for NewOrder we varied the probability that at least one of the purchased items is located in a remote partition.

Figure 7c shows the total throughput with a varying fraction of distributed transactions. As the percentage of distributed transactions increases, the already existing conflicts become more pronounced due to the prolonged duration of transactions, since a higher ratio of transactions must wait for network roundtrips to access records on remote partitions. This observation clearly shows why having good partitioning layout is a necessity for good performance in traditional CC protocols, and why existing partitioning techniques aim to minimize the percentage of distributed transactions.

Also, compared to the traditional concurrency protocols, Chiller degrades the least when the fraction of distributed transactions increases. More specifically, the performance of Chiller drops only by 26%, while NO_WAIT and WAIT_DIE both observe close to 50% drop in throughput, and the throughput of OCC has the largest decrease, which is about 73%. This is because the execution threads for a partition always have useful work to do; when a transaction is waiting for remote data, the next transaction can be processed. Since in Chiller, conflicts are handled sequentially in the inner region, concurrent transactions have a much smaller likelihood of conflicting with each other. Therefore, an increase in the percentage of distributed transactions only means higher latency per transaction, and not much increased contention, therefore has much less impact on the throughput. This highlights our claim that minimizing the number of multi-partition transactions should not be the primary goal in the next generation of OLTP systems that leverage fast networks, but rather that optimizing for contention should be.

7.5 YCSB Results

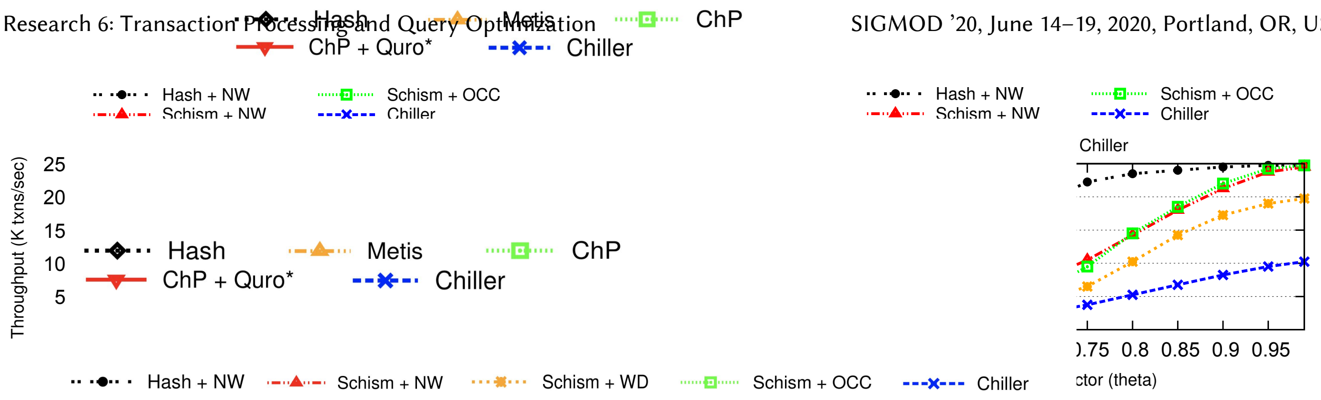
7.5.1 Single-Partition Transactions. We begin by examining the impact of contention on single-partition transactions. We use the *YCSB local* workload and vary the skew level θ from 0.5 (low skew) to 0.99 (high skew, the default in the original YCSB). The aggregated throughput and the average abort rate are shown in Figures 8a and 8b. For this workload, both Chiller and Schism can produce the same split as the ground truth mapping function we used to generate transactions. As explained before, under traditional CC schemes, distributed transactions significantly intensify any contention in the workload, which explains the steep increase in the abort rate of the hash partitioning baseline in Figure 8b.

As the contention increases, all traditional CC schemes face high abort rate, reaching more than 50% with $\theta = 0.85$. Chiller, on the other hand, is able to minimize the contention and hence reduce the abort rate. When the skew is high ($\theta = 0.99$), the throughput of Chiller is more than 85% higher than the second best baseline, NO_WAIT, while its abort rate is about half of WAIT_DIE. The initial increase in the throughput of Chiller and Schism-based schemes can be attributed to cache effects: with higher contention, there is a smaller working set of data for most transactions, making CPU/NIC caching more effective. However, as the contention further increases, the higher abort rate outweighs the caching effect.

This experiment shows that even for a workload with only single-partition transactions which is considered the sweet spot of the traditional partitioning and CC techniques, high contention can result in a major performance degradation, and Chiller's two-region execution model manages to alleviate the problem to a great extent.

7.5.2 Scalability of Distributed Transactions. We next compare the scalability of different schemes in the presence of distributed transactions. For this purpose, we used *YCSB distributed* workload, in which each transaction reads 4 records from the entire database, and modifies 2 skewed records from the same partition. Schism gives up fine-grained record-level partitioning and chooses simple hash partitioning, because in this workload, co-locating the hot records is not advantageous to simple hashing in terms of minimizing distributed transactions. Therefore, we also show the results for the original partitioning produced by Metis, which aims to minimize the number of cross-partition record accesses.

Figure 9 shows the throughput of the different protocols as the number of partitions (and servers) increases. To avoid



bort Rate.

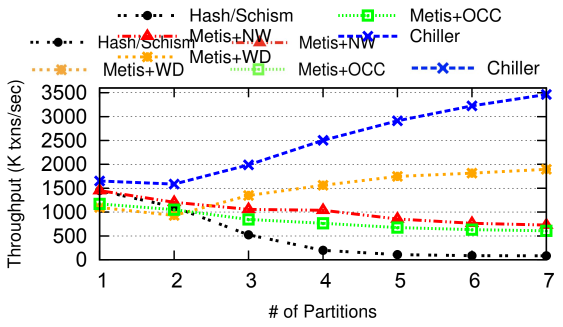


Figure 9: YCSB distributed with increasing cluster sizes.

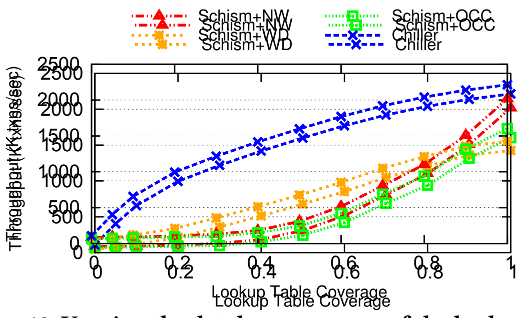


Figure 10: Varying the database coverage of the lookup table.

cluttering the graph, we only show the performance of the best CC scheme for Schism, which is NO_WAIT. To maintain a consistent replication factor of two, for the cluster size of one and two, we dedicate two and one extra backup machines, respectively, which do not process new transactions, and only process replication logs.

At first, all schemes drop in throughput despite the increase in the number of machines from one to two, which is due to the introduced distributed transactions. Surprisingly, as the number of partitions increases, all the CC schemes which use Metis partitioning outperforms the one which uses Schism, even though that almost all transactions are distributed in both cases. This is because in Metis, the contended records are co-located, and this drastically reduces the negative impact of aborting transactions in our system. More specifically, a transaction which fails to get one of the two contended locks would release the other one immediately, whereas in the partitioning produced by Schism, these two records are likely to be placed in different partitions, and releasing the locks for an aborted transaction may take one network roundtrip, further intensifying the contention problem. WAIT_DIE performs better than NO_WAIT since its waiting mechanism is able to overcome the lock thrashing issue in NO_WAIT, though we note that we also observed this phenomenon for WAIT_DIE for workloads where transactions access a higher number of hot records (e.g. see Figure 8b). Compared to all the other schemes, Chiller scales much better since not only its partitioning co-locates the contended records together, but also its two-region execution model is able to access those records in inner regions of transactions and therefore significantly reduces the contention. In fact, our measurements showed that the abort rate of Chiller with 7 machines is only 11%, whereas it is 81% and 86% for NO_WAIT and OCC, respectively. WAIT_DIE again resulted in much lower abort rate of 49%, resulting in better scalability compared to the other baselines. On 7 machines, the performance of Chiller is 4× and 40× of NO_WAIT on Metis and Schism partitions, respectively, while close to 2× of the second best baseline, WAIT_DIE.

7.5.3 Lookup Table Size. In this experiment, we investigate the performance of our proposed scheme in situations where a full-coverage lookup table cannot be either obtained or stored, as discussed in Section 4.5.1. This can mainly happen when the number of database records is too large to store a complete lookup table on each machine.

We used *YCSB local*, and fixed the skew parameter θ to 0.8 to represent a moderately skewed workload. Since all transactions in this workload are single-partition with respect to the ground truth mapping, both Schism and Chiller are able to find the optimal partitioning which makes all transactions single-partition, but this requires to cover the entire database in their resulting lookup tables. To measure the impact of lookup table coverage, we vary the percentage of the records which are partitioned according to the optimization goal of each partitioning algorithm. We used hash partitioning for the remaining records which, as a result, do not take up any lookup table entry, but result in a significant increase in the number of multi-partition transactions.

The results are shown in Figure 10. When all records are hash partitioned (the lookup table is empty), Chiller and all the other schemes achieve similar throughput. As the lookup table increases in size, Chiller starts to diverge from the other schemes. With a coverage of only 20%, Chiller achieves close to half of its peak throughput, whereas NO_WAIT and OCC achieve less than 0.1 of their peak throughput. In contrast, NO_WAIT relies on a 80% coverage to achieve half of its max throughput. The wide gap between Chiller and the other protocols is due to the way that Chiller handles contention. Placing the contended records (which are often a small fraction of the entire database) in the right partitions and handling them in inner regions are enough to remove most of the contention in the workload. The rest of the records can be randomly assigned to partitions without increasing contention.

This experiment supports our claim in Section 4.5 that, compared to partitioning schemes aiming to minimize distributed transactions, Chiller requires a much smaller lookup table to achieve a similar throughput. In addition, while for this particular workload there exists a database split where each transaction accesses one partition, for many real workloads such partitioning does not exist. Therefore, this experiment shows how Chiller compares against the other schemes for workloads with different degrees of partitionability.

7.6 Instacart Results

In our final experiment, we analyze the benefits of combining the Chiller's partitioning scheme with the two-region execution model. We use a real-world Instacart workload (as introduced in Section 7.3), which is harder to partition than TPC-C and YCSB. Furthermore, we use the same replication factor of 2 as for the previous experiments.

In order to understand whether or not the two-region execution model of Chiller is beneficial for the overall performance, 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 [40], 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.

Figure 11 shows the results of this experiment for increasing cluster sizes. 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 technique reduces the number of distributed transactions, but rather because contended records which are accessed together are co-located, which in turn reduces the cost of aborting transactions. More

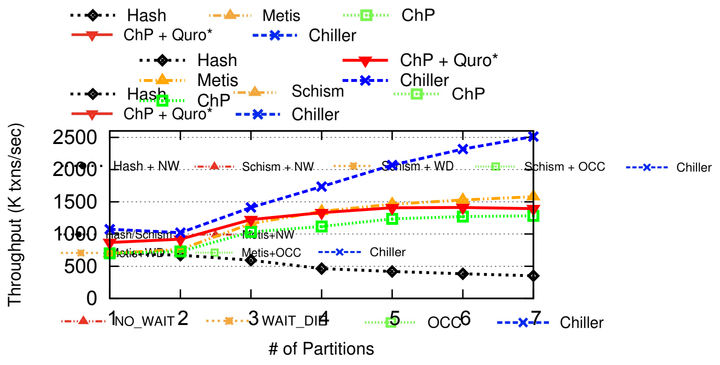


Figure 11: Instacart with different execution models.

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 (cf. Section 7.5.2).

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 with different commit points, and thus can outperform all the other techniques. For example, for the largest cluster size, the throughput of Chiller is by approximately 1 million txns/sec higher than the second best baseline. This clearly shows that the contention-centric partitioning must go hand-in-hand with the two-region execution to be most effective.

8 Related Work

Data Partitioning: A large body of work exists for partitioning OLTP workloads with the ultimate goal of minimizing cross-partition transactions [8, 37]. Most notably, Schism [8] is an automatic partitioning and replication tool that uses a trace of the workload to model the relationship between the database records as a graph, and then applies METIS [22] to find a small cut while approximately balancing the number of records among partitions. Clay [29] builds the same workload graph as Schism, but instead takes an incremental approach to partitioning by building on the previously produced layout as opposed to recomputing it from scratch. E-store [34] balances the load in the presence of skew in tree-structured schemas by spreading the hottest records across different partitions, and then moving large blocks of cold records to the partition where their co-accessed hot record is located. Given the schema of a database, Horticulture [28] heuristically navigates its search space of table schemas to find the ideal set of attributes to partition the database. As stated earlier, all of these methods share their main objective of minimizing inter-partition transactions, which in the past have been known to be prohibitively expensive. However, in the age of new networks and much "cheaper" distributed transactions, such an objective is no longer optimal.

Transaction Decomposition: There has been also work exploring the opportunities in decomposing transactions into smaller units. Gemini [25] introduces a mixed consistency model called BlueRed in which transaction operations are divided into blue operations, which are eventually consistent with lower latency, and red operations, which are strongly consistent which require global serialization. Gemini optimizes for overall latency and requires data to be replicated at all servers, and therefore does not have the notion of distributed transactions. Chiller, on the other hand, optimizes for minimizing contention, and supports distributed transactions. There has also been work on the theory of transaction chopping [30, 31, 46], in which the DBMS splits a transaction into smaller pieces and treats them as a sequence of independent transactions. In contrast to the idea of 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 our technique to abort a transaction later in the inner region. Transaction chopping techniques, however, must adhere to *rollback-safety*, in which all operations with the possibility of rollback must be executed in the first piece, since subsequent pieces must never fail. This restricts the possible ways to chop the transaction.

Determinism and Contention-Reducing Execution: Another line of work aims to reduce contention through enforcing determinism to part or all of the concurrency control (CC) unit [7, 21, 35]. In Granola [7], servers exchange timestamps to serialize conflicting transactions. Calvin [35] takes a similar approach, except that it relies on a global agreement scheme to deterministically sequence the lock requests. Faleiro et al. [14, 15] propose two techniques for deterministic databases, namely lazy execution scheme and early write visibility, which aim to reduce data contention in those systems. All of these techniques and protocols require *a priori* knowledge of read-set and write-set.

There has also been a large body of work on optimizing and extending traditional CC schemes to make them more apt for in-memory databases. MOCC [38] targets thousand-core systems with deep memory hierarchies and proposes a new concurrency control which mixes OCC with selective pessimistic read locks on contended records to reduce clobbered reads in highly contended workloads. Recent work on optimistic CC leverages re-ordering operations inside a batch of transactions to reduce contention both at the storage layer and validation phase [10]. While Chiller also takes advantage of operation re-ordering, it does so at an intra-transaction level without relying on transaction batching. MV3C [9] introduces the notion of repairing transactions in MVCC by re-executing a subset of a failed transaction logic instead of running it from scratch. Most related to Chiller is Quro [40], which also re-orders operations inside transactions in a centralized DBMS with 2PL to reduce 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 scopes.

Transactions over Fast Networks: This paper continues the growing focus on distributed transaction processing on new RDMA-enabled networks [3]. The increasing adoption of these networks by key-value stores [19, 24, 27] and DBMSs [5, 11, 20, 43] is due to their much lower overhead for message processing using RDMA features, low latency, and high bandwidth. These systems are positioned in different points of the spectrum of RDMA. For example, FaSST [20] uses the unreliable datagram connections to build an optimized RPC layer, and FaRM [11] and NAM-DB [43] leverage the RDMA feature to directly read or write data to a remote partition. Though different in their design choices, scalability in the face of cross-partition transactions is a common promise of these systems, provided that the workload itself does not impose contention. Therefore, Chiller's two-region execution and its contention-centric partition are specifically suitable for this class of distributed data stores.

9 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 which are likely to be accessed together are placed on the same partition. Using a novel two-region processing approach, it then executes the *hot* part of a transaction separately from the *cold* part. Our experiments show that Chiller can significantly outperform existing approaches under workloads with varying degrees of contention.

10 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).

References

[1] 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 (2017), 121–134.

[2] Kyle Banker. 2011. *MongoDB in action.* Manning Publications Co.

[3] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *PVLDB* 9, 7 (2016), 528–539.

[4] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06).* USENIX Association, 335–350.

[5] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. 2017. Fast in-memory transaction processing using RDMA and HTM. *ACM Transactions on Computer Systems (TOCS)* 35, 1 (2017), 3.

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

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

[8] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 48–57.

[9] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. 2017. Transaction Repair for Multi-Version Concurrency Control. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 235–250.

[10] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control Through Transaction Batching and Operation Reordering. *PVLDB* 12, 2 (2018), 169–182.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP).* 54–70.

[12] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* 299–313.

[13] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data.* 301–312.

[14] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017), 613–624.

[15] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. 2014. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data.* 15–26.

[16] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment* 10, 5 (2017), 553–564.

[17] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference.*

[18] Instacart. 2017. The Instacart Online Grocery Shopping Dataset 2017.

[19] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2015. Using RDMA efficiently for key-value services. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 295–306.

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 185–201.

[21] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[22] G. Karypis and V. Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.

[23] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency rationing in the cloud: pay only when it matters. *Proceedings of the VLDB Endowment* 2, 1 (2009), 253–264.

[24] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-direct: high-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP).* 137–152.

[25] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 265–278.

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

[27] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store.. In *USENIX Annual Technical Conference.* 103–114.

[28] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* 61–72.

[29] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[30] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst.* 20, 3 (Sept. 1995), 325–363.

[31] Dennis Shasha, Eric Simon, and Patrick Valduriez. 1992. Simple Rational Guidance for Chopping up Transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data.* 298–307.

[32] Utku Sirin, Ahmad Yasin, and Anastasia Ailamaki. 2017. A Methodology for OLTP Micro-architectural Analysis. In *Proceedings of the 13th International Workshop on Data Management on New Hardware (DAMON).* Article 1, 1:1–1:10 pages.

[33] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[34] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

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

[36] Alexander Garvey Thomson. 2013. *Deterministic Transaction Execution in Distributed Database Systems.* Ph.D. Dissertation. Yale University. Advisor(s) Abadi, Daniel J.

[37] Khai Q Tran, Jeffrey F Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. 2014. JECB: A join-extension, code-based approach to OLTP data partitioning. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 39–50.

[38] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *PVLDB* 10, 2 (2016), 49–60.

[39] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. 2017. Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing. In *USENIX Annual Technical Conference*. 335–347.

[40] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment* 9, 5 (2016), 444–455.

[41] Philip S. Yu, Daniel M. Dias, and Stephen S. Lavenberg. 1993. On the Analytical Modeling of Database Concurrency Control. *J. ACM* 40, 4 (1993), 831–872.

[42] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-memory Databases. *Proc. VLDB Endow.* 9, 6 (Jan. 2016), 504–515.

[43] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The end of a myth: Distributed transactions can scale. *Proceedings of the VLDB Endowment* 10, 6 (2017), 685–696.

[44] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. 2015. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 17–30.

[45] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proc. VLDB Endow.* 12, 11 (July 2019), 1637–1650.

[46] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. 276–291.