Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning

Guna Prasaad
University of Washington
guna@cs.washington.edu

Alvin Cheung
University of California, Berkeley
akcheung@cs.berkeley.edu

Dan Suciu
University of Washington
suciu@cs.washington.edu

ABSTRACT

Research on transaction processing has made significant progress towards improving performance of main memory multicore OLTP systems under low contention. However, these systems struggle on workloads with lots of conflicts. Partitioned databases (and variants) perform well on high contention workloads that are *statically partitionable*, but time-varying workloads often make them impractical. Towards addressing this, we propose Strife—a novel transaction processing scheme that clusters transactions together *dynamically* and executes *most* of them without *any* concurrency control. Strife executes transactions in batches, where each batch is partitioned into disjoint clusters without any cross-cluster conflicts and a small set of residuals. The clusters are then executed in parallel with *no* concurrency control, followed by residuals separately executed *with* concurrency control. Strife uses a fast dynamic clustering algorithm that exploits a combination of random sampling and concurrent union-find data structure to partition the batch *online*, before executing it. Strife outperforms lock-based and optimistic protocols by up to 2× on high contention workloads. While Strife incurs about 50% overhead relative to partitioned systems in the statically partitionable case, it performs 2× better when such static partitioning is not possible and adapts to dynamically varying workloads.

ACM Reference Format:
Guna Prasaad, Alvin Cheung, Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389764

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389764

1 INTRODUCTION

As modern OLTP servers today ship with an increasing number of cores, users expect transaction processing performance to scale accordingly. In practice, however, achieving scalability is challenging, with the bottleneck often being the concurrency control needed to ensure a serializable execution of the transactions. Consequently, a new class of high performance concurrency control protocols [8, 18, 19, 37, 39, 44] have been developed and are shown to scale to even thousands of cores [39, 43].

Yet, the performance of these protocols degrades rapidly on workloads with high contention [1, 43]. One promising approach to execute workloads with high contention is to partition the data, and have each core execute serially the transactions that access data in one partition [1, 15, 24, 29]. The intuition is that, if many transactions access a common data item, then they should be executed serially by the core hosting that data item without any need for synchronization. In these systems, the database is partitioned *statically* using either user input, application semantics, or workload-driven techniques [6, 25, 26, 28]. Once the data is partitioned, the transactions are routed to the appropriate core where they are executed serially. For workloads where transactions tend to access data from a single partition [34], static data partitioning systems are known to scale up very well [15].

However, cross-partition transactions (i.e., those that access data from two or more partitions) *do* require synchronization among cores and can significantly degrade the performance of the system. Unfortunately, in some applications, a large fraction of the transactions are cross-partition as no good static partitioning exists, or the hot data items change dynamically over time. As an extreme example, in the 4Chan social network, a post lives for only about 20-30 minutes, and the popularity can vary within seconds [2]. The various static partitioning schemes differ in how they synchronize cross-partition transactions: H-Store [13] acquires partition-level locks, Dora [1, 24] migrates transactions between cores, while Orthrus [29] delegates concurrency control to a dedicated set of threads. If the database is statically partitionable, then these approaches can mitigate a small fraction of cross-partitioned transactions, however, their performance degrades rapidly when this fraction increases, or when the workload is not statically partitionable as discussed in [1].

In this paper, we propose Strife, a new batch-based OLTP system that partitions transactions *dynamically*. Strife groups transactions into batches, partitions them based on the data they access, then executes transactions in each partition serially on an exclusive core. Our key insight is to recognize that most OLTP workloads, even those that are not amenable to static partitioning, can be "almost" perfectly partitioned at a batch granularity. More precisely, Strife divides the batch into two parts. The first consists of transactions that can be grouped into disjoint *clusters*, such that no conflicts exist between transactions in different clusters, and the second consists of *residual* transactions, which include the remaining ones. Strife executes the batch in two phases: in the first phase it executes each cluster on a dedicated core, without any synchronization, and in the second phase it executes the residual transactions using some fine-grained serializable concurrency protocol.

As a result, Strife does not require a static partition of the database apriori; it *discovers* the partitions dynamically at runtime. If the workload is statically partitionable, Strife will rediscover these partitions, setting aside the cross-partition transactions as residuals. When the workload is not statically partitionable, Strife will discover a *good* way to partition the workload at runtime, based on hot data items. It easily adapts to time-varying workloads, since it computes the optimal partition for every batch.

The major challenge in Strife is the clustering algorithm, and this forms the main technical contribution of this paper. Optimal graph clustering is NP-hard in general, which rules out an exact solution. Several polynomial time approximation algorithms exist [9, 16, 21, 23], but their performance is still too low for our needs. To appreciate the challenge, consider a statically partitioned system that happens to execute a perfectly partitionable balanced workload. Its runtime is linear in the total number of data accesses and scales with the number of cores. To be competitive, the clustering algorithm must run in linear time, with a small constant, and must be perfectly parallelizable across cores. Our novel clustering algorithm satisfies both criteria – it performs clustering using only three passes over data items and exploits intra-batch parallelism (Sec. 3). To achieve this, we also extend the union-find data structure to support concurrent execution (Sec. 4).

We implemented a prototype of Strife and conducted an extensive empirical evaluation of the entire algorithm (Sec. 5). We start by evaluating Strife's clustering algorithm and show that, in addition to having high performance, it also finds high quality clusters on three different benchmarks. Next, we compare Strife with general purpose concurrency control techniques, and find that its throughput is at least 2× that of the nearest competitor. A more interesting comparison is with protocols designed specifically for high contention workloads. We find that, in the ideal case when the workload is statically partitionable and there are very few cross-partition transactions, Strife achieves about half of the throughput of a statically partitioned system (due to clustering overhead). However, when the workload is not statically partitioned, or there are many cross-partition transactions, then it can improve throughput by 2× as compared to other systems. We conclude by reporting a number of micro-benchmarks to evaluate the quality of the clustering algorithm, and the impact of its parameters.

While many automated static partitioning techniques exist in the literature (surveyed in Sec. 6), to the best of our knowledge the only system that considers dynamic partitioning is LADS [42]. Like Strife, LADS uses dynamic graph-based partitioning for a batch of transactions. Unlike Strife, the partitioning algorithm in LADS depends on range-based partitioning. In addition, it does not eliminate the need for concurrency control, instead it breaks a transaction into record actions, and migrates different pieces to different cores, to reduce the number of locks required. This creates new challenges, such as cascading aborts, and is unclear how much overhead it introduces. In contrast, Strife is not limited to range partitions, is free from cascading aborts, and completely eliminates concurrency control when possible.

In summary, we make the following contributions:

- We describe Strife, an OLTP system that relies on dynamic clustering of transactions, aiming to eliminate concurrency control for *most* transactions in a batch. (Sec. 2).
- We describe a novel clustering algorithm that produces high quality clusters using only three scans over the set of transactions in the batch. (Sec. 3).
- We then present a concurrent implementation of the union-find data structure that enables parallelizing the clustering of transactions. (Sec. 4).
- We conduct an extensive experimental evaluation, comparing our system with both general-purpose concurrency control systems, and with systems designed for high contention, based on a static partition of the database. (Sec. 5).

2 OVERVIEW

Strife executes transactions in batches, where it collects transactions for a fixed duration and executes them together. For each batch, Strife analyzes the conflicts among transactions based on their read-write sets[1] and partitions them with the goal of minimizing concurrency control. Strife executes batches in three phases: analysis, conflict-free, and residual. These phases are executed synchronously on all cores as shown in Fig. 1. During analysis phase, the batch is partitioned into a number of *conflict-free* clusters and some

[1]Read-write sets can be obtained either through static analysis of the transaction code or via a reconnaissance query and conditional execution as in deterministic databases [36].

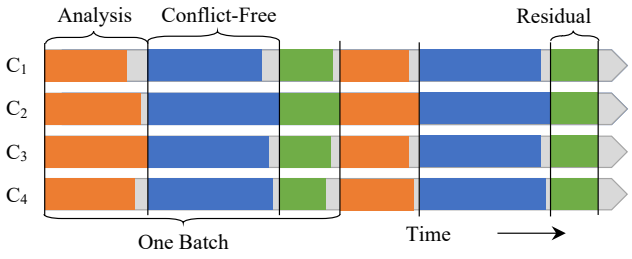


Fig. 1. STRIFE execution scheme

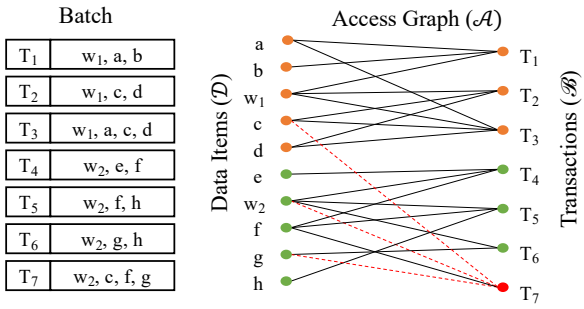


Fig. 2. Access Graph of a batch of transactions

residuals. Transactions in a conflict-free cluster are executed serially with *no* concurrency control, with several clusters executed in parallel on different cores in CONFLICT-FREE phase. After all conflict-free transactions are executed, residuals are then executed in a separate RESIDUAL phase across all cores using a conventional concurrency control protocol. Once the residual transactions are executed, STRIFE repeats the same process for the next batch.

The goal of ANALYSIS phase is to partition the batch into clusters such that transactions from two different clusters are *conflict-free*. To do so, STRIFE first represents a batch of transactions using its *data access graph*. A data access graph is an undirected bipartite graph $\mathcal{A} = (\mathcal{B} \cup \mathcal{D}, \mathcal{E})$, where $\mathcal{B}$ is the set of transactions, $\mathcal{D}$ is the set of data items (e.g., tuples or tables) accessed by transactions in $\mathcal{B}$, and the edges $\mathcal{E}$ contain all pairs (T, d) where transaction T accesses data item d. Transactions T, T' are *in conflict* if they access a common data item with at least one of them updating it.

Fig. 2 depicts the access graph of an example batch (colors explained later) derived from the TPC-C [5] benchmark. Each transaction updates a warehouse (w_1 or w_2) and a set of stocks ($a - h$). Transaction T_1, for example, accesses warehouse w_1 and stocks a and b. Transactions T_1, T_2 are in conflict because they both access w_1; whereas T_1, T_5 are not.

A batch can be partitioned into many conflict-free clusters when transactions access disjoint sets of data items. Transactions $T_1 - T_3$ and $T_4 - T_6$ (Fig. 2) access different data items ($\{w_1, a - d\}$ and $\{w_2, e - h\}$ respectively). Hence, the batch $T_1 - T_6$ can be partitioned into 2 clusters $\{T_1, T_2, T_3\}$ and $\{T_4, T_5, T_6\}$ with *no* cross-cluster conflicts.

However, real workloads often contain *outliers* that access data items from multiple clusters (such as red transaction T_7 in Fig. 2). A conflict-free cluster for the batch $T_1 - T_7$, thus will contain all transactions. To avoid this, we instead remove T_7 as a residual and create conflict-free clusters *only* with respect to the remaining batch. This results in 2 conflict-free clusters $\{T_1, T_2, T_3\}$, $\{T_4, T_5, T_6\}$ and a residual T_7.

Formally, a *clustering* in STRIFE is an $(n + 1)$-partition of a batch of transactions $\mathcal{B}$ into $\{C_1, C_2, \ldots, C_n, R\}$ such that, for any $i \neq j$ and transactions $T \in C_i, T' \in C_j$, T and T' do not conflict. Notice that there is no such requirement on R.

Ideally, we want to maximize n, the number of conflict-free clusters, to maximize parallelism, while minimizing the size of R. To elucidate the trade-offs, we describe two naive clusterings. The first is *fallback clustering*, where all transactions are in R and set $n = 0$; this corresponds to running the entire batch using conventional concurrency control. The second is *sequential clustering*, where we place all transactions in C_1, and set $n = 1$ and $R = \emptyset$; this corresponds to executing all transactions sequentially, on a single core. Obviously, neither of these are good clusterings. Hence, we want n to be at least as large as number of cores, and constrain R to be no larger than a small fraction, α, of the batch.

In practice, a good clustering exists for most workloads, except in the extreme cases. In one extreme, when all transactions access a *single* highly contended data item, then no good clustering exists besides fallback and sequential, and STRIFE simply resorts to fallback clustering. Once data contention decreases, i.e., the number of contentious data items increases, a good clustering would try to place each hot item in a different conflict-free cluster. When contention further decreases such that all transactions access different data items, then any n-clustering of roughly equal sizes is adequate. Thus, we expect good clustering to exist in all but most extreme workloads. The challenge, however, is to find a good clustering *efficiently*.

After partitioning, conflict-free clusters are placed in a shared worklist. Multiple cores then obtain a cluster from the worklist, and execute all transactions in it serially one after another in the CONFLICT-FREE phase. Once done with a cluster, a core obtains another from the worklist until done. The degree of parallelism in this phase is determined by the number and size of conflict-free clusters. More clusters result in parallel execution, thus reducing total time to execute them. Once the worklist has exhausted, a core waits for other cores to finish processing their clusters before moving to the RESIDUAL phase. This is because the residuals may conflict with the conflict-free transactions that are being executed concurrently, and hence requires data synchronization. A skew in cluster sizes may reduce concurrency since threads that complete early cannot advance to next phase.

Algorithm 1: STRIFE Clustering Algorithm

```
Input: List<Txn, ReadWriteSet> batch
Output: Queue<Queue<Txn>> worklist
        Queue<Txn> residuals
set<Cluster> special;
int count[k][k] = {0};
// Spot Step (initially each data node is a cluster)
i = 0;
repeat
    Pick a random transaction T from batch;
    set<Cluster> C = { Find(r) | r ∈ T.nbrs };
    set<Cluster> S = { c | c ∈ C and c is special };
    if |S| == 0 then
        c = C.first;
        foreach other ∈ C do
            Union(c, other);
        c.id = i;
        c.count++;
        c.is_special = true;
        special.insert(c);
        i++;
until k times;
// Fuse Step
foreach T ∈ batch do
    set<Cluster> C = { Find(r) | r ∈ T.nbrs };
    set<Cluster> S = { c | c ∈ C and c is special };
    if |S| ≤ 1 then
        c = (|S| == 0) ? C.first : S.first;
        foreach other ∈ C do
            Union(c, other);
        c.count++;
    else
        foreach (c1, c2) ∈ S × S do
            count[c1.id][c2.id]++;
// Merge Step
foreach (c1, c2) ∈ special × special do
    n1 = count[c1.id][c2.id];
    n2 = c1.count + c2.count + n1;
    if n1 ≥ α × n2 then
        Union(c1, c2);
// Allocate Step
foreach T ∈ batch do
    C = { Find(r) | r ∈ T.nbrs };
    if |C| = 1 then
        c = C.first;
        if c.queue = ∅ then
            c.queue = new Queue<Txn>();
            worklist.Push(c.queue);
        c.queue.Push(T);
    else
        residuals.Push(T);
return (worklist, residuals);
```

As such, optimal allocation of conflict-free clusters to cores can be modeled as a 1-D bin-packing problem. However, solving this requires estimating the cost of executing a transaction, and the quality of solution critically depends on how good the estimate is. We instead rely on work-sharing parallelism powered by a low-overhead concurrent queue to balance load across cores dynamically.

After the conflict-free clusters are processed, transactions in the residual cluster are executed in RESIDUAL phase. As discussed earlier, concurrency control is needed to execute them concurrently across cores. STRIFE uses two-phase locking with no-wait deadlock avoidance policy as it is shown to be highly scalable [43]. Under this policy, a transaction aborts immediately when it fails to acquire a lock. STRIFE retries transactions aborted this way until successful commit or logical abort. Finally, after the residuals are executed, STRIFE processes the next batch. We next describe the clustering algorithm in STRIFE.

3 STRIFE CLUSTERING ALGORITHM

A batch of transactions in STRIFE, $\mathcal{B}$, is partitioned into a conflict-free clusters $C_1, C_2, \ldots, C_n$, and a set of residuals, R, of size at most $\alpha|\mathcal{B}|$. The inputs to the algorithm are the batch of transactions (denoted $\mathcal{B}$) along with their read-write sets, a number k, typically a small factor times core count, and the residual parameter α, a fraction between 0 and 1.

Our clustering problem can be viewed as partitioning the data access graph $\mathcal{A} = (\mathcal{B} \cup \mathcal{D}, \mathcal{E})$ (refer Fig. 2) with dual objectives: maximize number of conflict-free clusters, and reduce the size of residuals. Graph partitioning in general is NP-hard, and several PTIME approximations have been proposed in the literature [9, 10, 21, 23, 27]. These solutions, however, do not meet the latency budget of our use case. For instance, our preliminary study revealed that partitioning a batch from the TPC-C benchmark using METIS [21], a state-of-the-art graph partitioning library, is an order of magnitude slower than executing directly with concurrency protocol. Hence, our key challenge is to devise a clustering algorithm by touching each node in the graph as few times as possible.

In other words, we seek an approximation algorithm that runs in linear time, with a very small constant and can be parallelized to incur the least runtime overhead. We next describe our algorithm (Sec. 3.1), illustrate it using an example (Sec. 3.2), and finally analyze its runtime (Sec. 3.3); the pseudo-code is listed in Alg. 1.

3.1 Description

Partitioning a batch of transactions in STRIFE is akin to partitioning the data nodes in $\mathcal{A}$ into *data clusters*. Once that is done, we can then allocate each transaction to a conflict-free cluster or residuals depending on how its connected data nodes are clustered. Hence, our algorithm will cluster the

data nodes first, then allocate transactions to an appropriate cluster or the residuals. The algorithm runs in 5 steps: (1) PREPARE (2) SPOT (3) FUSE (4) MERGE and (5) ALLOCATE.

Initially each data node in the graph is a singleton cluster. The algorithm repeatedly merges clusters connected to a transaction, so that data nodes accessed by that transaction belong to the same data cluster. It uses different heuristics to prevent merging all data nodes into a single large data cluster. We describe each step in detail below.

3.1.1 Prepare Step. First, STRIFE creates the data access graph from read-write set of transactions in the batch. STRIFE does not explicitly create a graph data structure in memory, instead uses the read-write sets along with some metadata to avoid overheads. We consider only data nodes that are *updated* by at least one transaction in the batch, since there is no need to coordinate concurrent access to read-only data items. For instance, the `Items` table in the TPC-C benchmark is only read and never updated; as a result `Items` is not part of our access graph. We do this by scanning and labeling records that are updated as *active* and ignoring all inactive records in subsequent steps (not shown in Alg. 1 for brevity).

3.1.2 Spot Step. The SPOT step (Lines 3-17) identifies hot data items in the batch so that we can avoid placing several of them in the same data cluster. Initially, all data nodes in the access graph are singleton data clusters. We say a transaction is *connected* to a data cluster c if it accesses any data node in c. In this step, we create a small number of clusters that are all labeled "special" using a sample-reject scheme. We begin by picking a transaction T from $\mathcal{B}$ uniformly at random. We merge data clusters connected to T and label the resulting cluster "special."

We continue by repeatedly picking another transaction T' at random; if T' is connected to a special cluster, then we reject it, otherwise we merge all its data nodes into a new "special" cluster. We continue this for k trials, creating k or fewer special clusters. Recall that k is an input parameter to the algorithm, and is typically chosen to be a small factor times the number of cores. For instance, in our experiments we use $k = 100$ for 30 cores.

With a high probability, our scheme will find the "hot" data items (i.e., items accessed by many transactions) and create a special cluster for each of them. To understand the intuition, suppose d is a hot data item accessed by half of the transactions. Then, with high probability, one of the randomly chosen transactions will access d, and SPOT will create a special cluster for d; if another sampled transaction also accesses d, it will be rejected. Suppose now that there are n "hot" data items, $d_1, d_2, \ldots, d_n$, and each is accessed by approximately $|\mathcal{B}|/n$ transactions. In this case, with high probability, SPOT will create a special cluster for each d_i, because the probability that none of the k sampled transactions accesses a data item d_i is $(1-1/n)^k \approx 1/e^{k/n}$, which is very low when $k \gg n$; for example, if $k \gtrsim 3n$, then $1/e^{k/n} \leq 1/e^3 = 0.04$. This argument extends to the case of pairs of data items d_1, d_2 that have *direct affinity*, meaning that a large number of transactions access both; then, with high probability, some special cluster will include both.

3.1.3 Fuse Step. The SPOT step processed only a small number of transactions (up to k); the FUSE step (Lines 18-28) processes all remaining transactions in the batch. For each transaction, we examine the data clusters connected to it. If at most one of them is labeled "special," we merge all of them together. To prevent creating a single cluster with all data items, FUSE never merges two special clusters: if a transaction accesses two or more special clusters, then FUSE skips it, and instead records the count of such transactions for each special cluster pair (Line 28). The number of special clusters remains the same throughout FUSE step, but each special cluster may have grown from it's initial size in the SPOT step, and new non-special clusters may be created as well. The merge of two clusters is "special" if and only if at least one of them was special.

3.1.4 Merge Step. So far the special clusters created in SPOT have never been merged. In this step, we revisit this invariant and merge *some* of them to reduce the number of residuals. If two hot data items d_1, d_2 are accessed by many transactions, then the special clusters they belong to are good candidates for merging; we say that d_1, d_2 have *direct affinity*. Special clusters may also have to be merged due to *indirect affinity* as illustrated by the following example.

Suppose there are two hot items, d_1, d_2, each in a separate special cluster, and a third item d_3, such that a large number of transactions access both d_1, d_3 and another large number access d_2, d_3; then we say that d_1, d_2 have high *indirect affinity*. Depending on the order of processing, FUSE will include d_3 either in d_1's special cluster or in d_2's special cluster. In the first case, all transactions that access d_2, d_3 are skipped as residuals, in the second case the d_1, d_3 transactions are skipped. MERGE step examines the benefit of merging the two special clusters of d_1 and d_2, thus allowing all these transactions to be executed in the CONFLICT-FREE phase.

In such scenarios, we relax the criterion from FUSE step and merge special clusters. We use a *new* heuristic to prevent merging all data items into a single huge cluster in the MERGE step (Lines 29-33). If two special clusters c_i and c_j are merged, transactions connected to c_i and c_j can now be classified into a conflict-free cluster. We merge them only when residuals are more than the bound specified by parameter α, i.e., $|R| \geq \alpha|\mathcal{B}|$. Recall that α is one of the input parameters of the algorithm: it chooses between executing transactions on more cores with concurrency control (if α is small) versus

on fewer cores without concurrency control. Empirically, we found $\alpha = 0.2$ to give the best results in our experiments.

Let $\texttt{count}[c_i][c_j]$ denote the number of transactions in $\mathcal{B}$ that access data items in c_i and c_j. Note that the transactions that are accounted for in $\texttt{count}[c_i][c_j]$ can access data items from clusters other than c_i and c_j as well. If the two clusters c_i and c_j are not merged, then all of $\texttt{count}[c_i][c_j]$ will be residuals. So, we merge cluster pairs c_i, c_j when:

$$\texttt{count}[c_i][c_j] \geq \alpha \times (|c_i| + |c_j| + \texttt{count}[c_i][c_j])$$

Since, $|R| \leq \sum_{i \neq j} count[c_i][c_j]$, this merge scheme results in residuals fewer than $\alpha|\mathcal{B}|$ when most transactions access at most two special clusters.

3.1.5 Allocate Step. In the ALLOCATE step (Lines 34-43), we examine data clusters connected to each transaction and allot it to a conflict-free cluster or the residuals. If the number of clusters connected to a transaction T, denoted $|C|$, is more than one, then T is added to the residuals since it has conflicting data accesses with more than one data cluster. If $|C| = 1$, all data items accessed by T belong to one data cluster and hence does not conflict with transactions not connected to the data cluster. So, T can be assigned to a conflict-free cluster. We first check if the data cluster already has a designated queue (Line 38). If so, we add T to that queue. Else, we create a new conflict-free queue and add T to it. This queue is added to the shared worklist. This way all transactions connected to a data cluster are added to the same conflict-free queue, thus ensuring a serializable execution.

3.2 Example

We illustrate in Fig. 3 how STRIFE clusters an example batch of new-order transactions from the TPC-C benchmark. We consider only stock and warehouse accesses for brevity. As shown in Fig. 3a, each transaction (mid column) updates a warehouse (right column) and few stock tuples (left column). Each stock item is listed in the catalog of one warehouse, as labeled on the left of Fig. 3a.

At the start, each warehouse and stock item is a singleton data cluster. SPOT randomly samples one transaction, say the first from the top, and merges its data items into one special cluster: they are shown in orange. If the next transaction sampled accesses any of the orange items, it is ignored, otherwise its data items form a new special cluster. After several trials, STRIFE creates four special clusters as shown in Fig. 3b: orange, green, red, blue. All black data items continue to be singleton clusters.

Next, FUSE processes all transactions, top to bottom. FUSE merges each transaction's data items into one single cluster, unless if doing so will merge two special clusters, in which case the transaction is skipped. For instance, the transaction accessing both a green and a red data item is skipped. In our example, after the FUSE step, all data items will be included in one of the four special clusters as shown in Fig. 3c; in general, this is not the case, instead some new clusters may be formed that are not special, or some data items may be left in their singleton cluster (also non-special). Notice that most of the catalog stock items of w_2 and w_3 are also accessed by transactions of the other warehouse and, since we process transactions from top to bottom, most of them will be allotted to the green cluster (of w_2).

At this stage, most of w_1, w_2 and w_4 transactions are connected to a single data cluster, while several w_3 transactions access data from both green and red clusters (notice that w_3 itself is red). We say that the w_2 and w_3 warehouse tuples have an *indirect affinity* via their stock items. These two special clusters are candidates for merging, because many transactions access both red and green items. More precisely, MERGE checks that the number of transactions accessing both clusters is more than α fraction of those accessing either of them, then merges them (refer Fig. 3d). This reduces the number of transactions in the residual cluster. Note that the clusters of w_1 and w_2 (orange and green) are not merged, because too few transactions access both (just two).

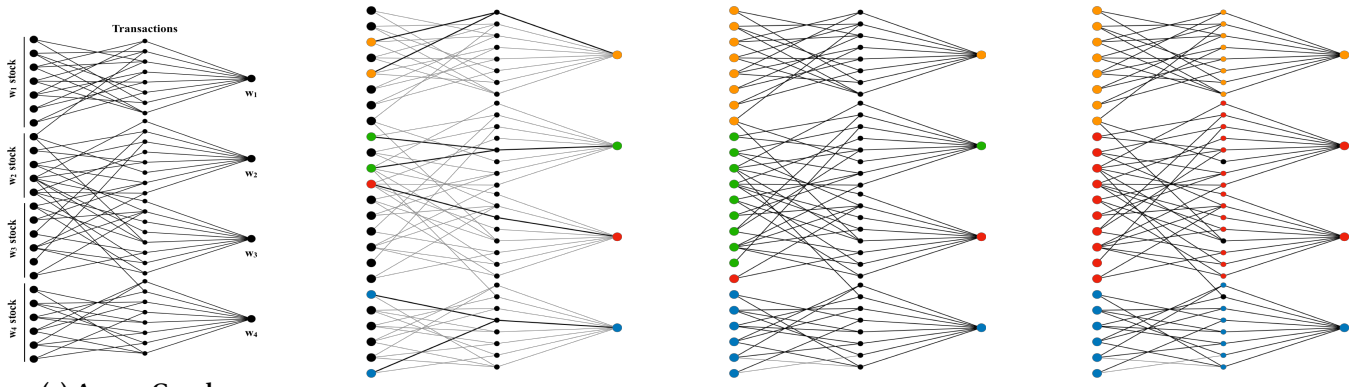
Finally, ALLOCATE iterates over all transactions, and for each, checks if it can be included in a conflict-free cluster or residuals. As shown in Fig. 3d, only three transactions are residual, two accessing orange+red, the other accessing red+blue; other transactions are included in one of the three conflict-free clusters: orange, red, or blue.

3.3 Runtime Analysis

We use the union-find [33] data structure to maintain data clusters in STRIFE. The amortized cost of a union-find operation is $O(\alpha(N)) \approx O(1)$ where N is the number of data items and α the inverse Ackermann function, thus we assume that every union/find operation takes time $O(1)$. PREPARE, FUSE and ALLOCATE steps iterate over the entire batch of transactions $\mathcal{B}$, but do this in parallel using n cores, thus, their total runtime is $O(|\mathcal{B}|/n)$, where the constant under $O(\cdots)$ is small (about 3). SPOT and MERGE are executed sequentially, and their runtime is as follows: SPOT takes time $O(k)$, since it samples using k trials, while MERGE takes time $O(k^2)$, since it examines up to k^2 pairs of special clusters. The total runtime of the clustering algorithm is $O(|\mathcal{B}|/n + k^2)$. For comparison, if the database can be perfectly partitioned statically, and there are no cross-partition transactions, then a system using static partitioning can execute entire batch in time $O(|\mathcal{B}|/n)$.

4 PARALLEL UNION-FIND

STRIFE heavily exploits the union-find data structure to partition the batch efficiently. The analysis is done *in parallel* on multiple cores to reduce end-to-end transaction latency. Specifically, the SPOT and MERGE steps are executed in a



(a) Access Graph (b) After SPOT (c) After FUSE (d) Final Solution

Fig. 3. STRIFE clustering on an example batch of TPC-C transactions (middle column, small, black nodes).

```
1  bool Union(Record* r1, Record* r2) {
2    while (true) {
3      Record* parent = Find(r1);
4      Record* child  = Find(r2);
5      if (parent == child) {
6        // do nothing
7        return true;
8      } else if (parent > M and child > M) {
9        // both are special
10       return false;
11     }
12     // choose parent, child
13     if (parent < child) {
14       Record* temp = parent;
15       parent = child;
16       child = temp;
17     }
18     // invariant: parent > child
19     // swap parent of child
20     if (cas(&child->parent, child, parent)) { // LP1
21       return true;
22     }
23   }
24 }
25
26 Record* Find(Record* r) {
27   // first pass: record path and find root
28   Record* root = r;
29   while (root != root->parent) { // LP2 (when false)
30     root = root->parent;
31   }
32   // second pass: path compression
33   CompressPath(r, root);
34   return root;
35 }
36
37 void CompressPath(Record* rec, Record* root) {
38   while (rec < root) {
39     Record* child = rec;
40     Record* parent = rec->parent;
41     if (parent < root)
42       cas(&child->parent, parent, root);
43     rec = rec->parent;
44   }
45 }
```

Fig. 4. Pseudo-code for Concurrent Union/Find

single thread, while PREPARE, FUSE and ALLOCATE steps are executed in multiple threads. In these steps, the batch of transactions is divided into equal-sized chunks and each chunk is processed by one thread. The main challenge here is to redesign `Union` and `Find` operations used in the STRIFE clustering algorithm to support parallel execution. We describe concurrent versions of these two operations below; the pseudo-code is shown in Fig. 4.

During `Union`, the clusters must be merged *atomically*. An additional invariant that STRIFE maintains during FUSE is "to not merge two special clusters" i.e., `Union` can be performed only between a special and non-special cluster or between two non-special clusters. In `Union` (as shown in Fig. 4), we first obtain the roots of the two clusters that correspond to r_1 and r_2 by invoking `Find`. If they have same roots, we return `true`. In STRIFE, special clusters are identified by having a root address that is greater than a constant M. If both clusters are special, i.e., root address $> M$, then we return `false` without any merge. Otherwise, we atomically merge the two clusters in a global order determined by memory addresses: we choose the cluster with larger root address and merge the other cluster with it. When merging a special cluster with a non-special cluster, special cluster is always chosen as parent, since their root address $> M$ and that of a non-special cluster is $< M$. We update `parent` pointers using the atomic *compare-and-swap* (`cas`) instruction. Since `child` always corresponds to a non-special cluster, and `cas` succeeds only when `child` is still the root of its cluster, the FUSE step invariant is guaranteed.

We implement a concurrent version of the `Find` operation with path compression [33]. Path compression amortizes the cost of union-find operations. However, when performed concurrently, this can lead to cycles and/or inconsistencies in the data structure. In our implementation, we exploit the invariant that the root address for a cluster increases monotonically. We find the `root` by tracing parent pointers. Then, we retrace the path from initial record to perform path compression. But now, we trace only until record address is less than `root`. We update the `parent` of a record only if the current parent is smaller than root using a `cas` operation.

5 EVALUATION

In this section, we evaluate STRIFE, by comparing it to other systems and on micro benchmarks. Specifically:

- §5.1 First, we analyze how well STRIFE's clustering algorithm works on OLTP workloads.
- §5.2 Then, we compare STRIFE with several concurrency control (CC) protocols using the shared-everything architecture.
- §5.3 We next evaluate STRIFE against state-of-the-art CCs for handling high contention workloads: MOCC [39], Dora [24], Orthrus [29] and PartCC [13].
- §5.4 We then evaluate STRIFE's adaptability when the hot data items dynamically vary over time.
- §5.5 Finally, we study the impact of the residual parameter α, batch size, and contention using microbenchmarks.

5.1 Setup, Implementation and Workloads

Experimental Setup. We run all our experiments on a multi-socket Intel Xeon CPU E7-4890 v2 @ 2.80GHz machine with 2TB Memory and two sockets - each having 15 physical cores. We implemented STRIFE and our baselines in C++. We use a default batch size of 10K transactions.

System Implementation. We have implemented a prototype of STRIFE that clusters transactions using the algorithm along with optimizations described in Sec. 3. STRIFE, at its core, is a multi-threaded transaction manager that interacts with storage layer of the database using the standard `get-put` interface. A table is implemented as a collection of in-memory pages that contain sequentially organized records. Some tables are indexed using a hashmap that maps primary keys to records. We implement the index using libcuckoo [22] library, a thread-safe fast concurrent hashmap. Records contain meta-data for clustering and concurrency control. We chose this design to avoid overheads due to multiple hash look-ups and focus primarily on concurrency control.

The ANALYSIS phase produces several conflict-free (CF) queues and a residual queue. All CF queues are stored in a multi-producer multi-consumer (MPMC) concurrent queue, called the *worklist*. The worklist is shared among all threads. In CONFLICT-FREE phase, threads obtain a queue from the worklist and execute the transactions in it serially one after another without any concurrency control. Once a thread is done with its queue, it obtains another from the worklist. When worklist is empty, threads spin wait for other threads to complete.

Once all threads are done with conflict-free queues, they enter RESIDUAL phase. The residuals are stored in a shared MPMC queue. Threads dequeue and execute them using 2-phase locking concurrency protocol with no wait deadlock avoidance policy (i.e., immediately aborts the transaction if it fails to grab a lock). This phase is similar to traditional concurrency control based OLTP systems.

STRIFE moves to ANALYSIS phase of next batch once RESIDUAL phase is complete. Threads could start analyzing the next batch while RESIDUAL phase of previous batch is in-progress. However, we did not implement this optimization to simplify the analysis and interpretability of results.

Workloads. We use the following benchmarks:

TPC-C [5]: We use a subset of full TPC-C consisting of a 50/50 mixture of new-order and payment transactions, with all payments accessing customer through primary key lookups. These two transactions make up the vast majority of the benchmark and has been used previously [29] in evaluating high contention workloads to stress concurrency control. Refer Sec. 5.6.4 for payment transactions that access customer records through last name. In TPC-C benchmark, 15% payment transactions are remote and 1% of items are ordered from remote warehouses.

YCSB [4]: For YCSB, each transaction reads/updates 20 records from a single table with 20M keys and a payload size of 128 bytes. We use a workload that is statically partitionable and hence is the best case scenario for such protocols. Transactions are such that intra-partition key access is determined by a Zipfian distribution controlled by the θ parameter. Unless specified otherwise, we use $\theta = 0.99$ to generate a highly contended workload.

HOT: HOT is a microbenchmark on a single table database with 50M records, each having an 8-byte key and twenty 8-byte integer fields. Each transaction updates 10 records from the table. To add contention, we classify some records (default: 100) as hot and the remaining as cold. Every transaction updates one hot record and 9 cold records. We control partitionability of the workload by limiting remote partition accesses to at most 3. By default, hot records are evenly distributed across all partitions.

5.2 Clustering Algorithm

Fig. 5 summarizes the clustering solutions produced by STRIFE on a batch of 10K transactions, $\alpha = 0.2$ and different benchmark parameters. *Spot Clusters* refers to the number of clusters identified in the SPOT step, *CF Clusters* denotes the number of conflict-free clusters finally produced, and *Residuals* denotes the size of residual cluster.

We present TPC-C results for different number of warehouses (denoted w), specifically 4, 15 and 30 in Fig. 5. SPOT step correctly identifies 4, 15 and 30 special clusters, one corresponding to each warehouse. TPC-C has around 25% cross-warehouse transactions and hence some of them are classified as residuals. We note that residuals are much less

		TPC-C			YCSB (30 Partitions)					HOT			
	Description/Workload Parameter	**4**	**15**	**30**	**0.1**	**0.5**	**0.8**	**0.99**	**1.2**	**10**	**50**	**100**	**200**
(a)	Spot Clusters	4	15	30	100	98	78	48	30	10	43	63	84
(b)	CF Clusters	4	15	30	100	98	78	30	30	10	43	63	84
(d)	Residuals	636	191	94	0	0	298	0	0	350	371	330	250

Fig. 5. Clustering: TPC-C (Varying #Warehouses), YCSB (Varying θ), HOT (Varying #Hot Records)

than 25% because a cross-warehouse transaction is not necessarily a cross-cluster transaction in the batch. For example, a remote order for item i from warehouse w_2 by a w_1 transaction is cross-cluster only when some w_2 transaction in the batch orders item i from w_2, which is not always the case.

We vary the Zipfian parameter θ in YCSB with 30 partitions. SPOT identifies approximately 100 special clusters when $\theta = 0.1$ or 0.5 since there are very few conflicting accesses, resulting in 100 CF clusters of almost equal size with no residuals. When $\theta = 0.8$, SPOT identifies 78 special clusters and produces same number of CF clusters. However, 30 of them contain majority of transactions, each corresponding to a partition. The remaining 48 CF clusters are small: they contain transactions that access partition-local data from one of these 30 partitions. Since some of these transactions result in cross-edges with the main 30 CF clusters, we have a small fraction (3%) of residuals; MERGE does not combine them since it does not meet the merging threshold derived from α. When $\theta = 0.99$, SPOT identifies 48 special clusters. But after FUSE and MERGE, the final number of CF clusters is 30 since STRIFE merges some of them. When the contention is very high ($\theta = 1.2$), SPOT easily identifies the 30 partitions and produces a conflict-free cluster for each.

We now vary number of hot records in the HOT benchmark. In each case, SPOT identifies most but not all of the hot records in the batch since we sample the batch only 100 times. We found that more sampling attempts identify all hot records. The same number of CF clusters are produced since none of them are merged. As we will see in Sec. 5.3, different sizes for CF clusters does not impact system throughput heavily since STRIFE uses a shared worklist and dynamically balances load across cores. In summary, the experiments show that STRIFE identifies hot records and clusters conflicting transactions together while distributing hot records to as many clusters as possible.

5.3 CC protocols

We now compare STRIFE against 4 classes of CC protocols:
LockSorted sorts the read-write set and acquires locks in an ordered fashion before executing the transaction.
DlDetect is a 2-Phase-Locking (2PL) variant which maintains a *waits-for* graph among transactions and uses it to detect and eliminate deadlocks by abortion. To avoid scalability bottlenecks, each thread maintains a local partition of the graph and updates it only when waiting [43].
NoWait is 2PL variant that aborts a transaction immediately when a lock cannot be acquired during transaction execution. We do not use a centralized lock table in any of our 2PL implementations since it is a known scalability bottleneck and instead use metadata co-located with records in the database.
PreNoWait acquires all locks at the start of the transaction and releases only after commit; it aborts the transaction immediately if any of the requested locks cannot be acquired.
Silo [37] is a representative OCC protocol and is considered state-of-the-art [1, 43, 44].
We implemented these protocols in the STRIFE code-base and corroborated it with [43] to make the comparisons fair.

Fig. 6 depicts the throughput as we vary the number of cores on TPC-C (4 and 30 warehouses), YCSB, and HOT benchmarks. We present the execution profile for HOT benchmark on 30 cores in Fig. 7; these protocols exhibit similar behavior on others. *Abort* is time spent in aborted execution, *Lock* is time spent in acquiring, releasing and waiting for locks, *Execute* is transaction code execution, *Index* is time spent querying the index, *Validation* is time spent in validating for Silo, and *Analysis* is time spent in analysis (STRIFE).

STRIFE produces as many CF clusters as warehouses for TPC-C (Sec. 5.2). With 2 cores, STRIFE is significantly better than other protocols on TPC-C (4 warehouses) due to its very high contention, and at the same time caching benefits for clustering due to its small size. With more cores, throughput doubles from 2 to 4 cores since CONFLICT-FREE phase is 2× faster. Beyond 4 cores, improvement in throughput is only due to increased parallelism in ANALYSIS and RESIDUAL phases and CONFLICT-FREE phase remains the same. With 30 warehouses, STRIFE increases parallelism in all three phases to scale almost linearly. STRIFE throughput on YCSB (30 partitions) also follows a similar trend. Since YCSB is partitionable without any cross-partition transactions, no residuals are created. Throughput increase from 15 to 25 cores is gradual in both TPC-C (30 warehouses) and YCSB since the core count is insufficient to execute all 30 CF clusters in parallel. However, with 30 cores STRIFE exploits all parallelism yielding almost a 2× improvement over 15 cores.

A batch in HOT benchmark is clustered into approximately 60 clusters, each containing at least one hot record. CONFLICT-FREE phase exploits maximum parallelism, thus scaling linearly as we increase number of cores since STRIFE

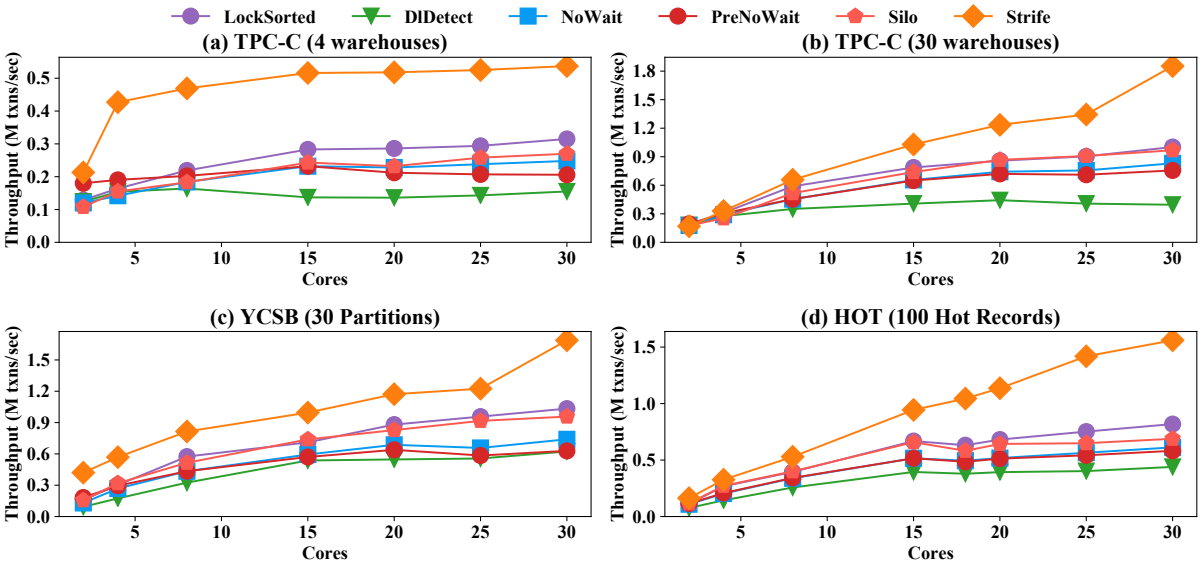


Fig. 6. Throughput comparison among protocols

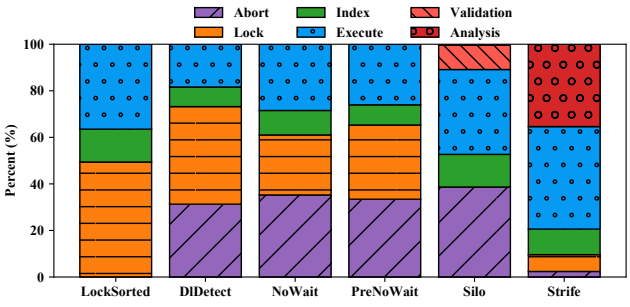


Fig. 7. HOT Benchmark: Execution Profile

dynamically balances load using a shared worklist. The runtime breakdown reveals that 40% time is spent on ANALYSIS, 50% in CONFLICT-FREE and the remaining in RESIDUAL.

In comparison, on TPC-C and HOT, DlDetect scales up to 15 cores and flattens after due to higher deadlock detection overhead and lock thrashing, especially across NUMA sockets. In YCSB, DlDetect also suffers due to inflated abort rates since data items are accessed in a random order.

NoWait, PreNoWait and Silo are lightweight compared to DlDetect but they suffer from aborts. Even though Silo executes to completion before aborting, it performs better than NoWait since it eliminates latching overheads on reads. NoWait spends about 30% time on locks while Silo spends almost 10% on validation. Both incur about 40% overhead due to aborts. Performance of PreNoWait is quite similar to NoWait. PreNoWait performs better than NoWait on fewer cores since abort rates are reduced, but with more cores PreNoWait suffers due to longer lock-hold times leading to more aborts (Fig. 7 depicts this for 30 cores).

LockSorted performs the best among CC protocols with almost 50% of STRIFE throughput. Unlike other protocols, it neither performs expensive deadlock detection nor does it incur abort-restart overheads. Instead, LockSorted waits on the lock in an ordered fashion. On HOT, LockSorted spends about 45% time acquiring, waiting and releasing locks.

The results show the effectiveness of STRIFE on high-contention workloads: analyzing the batch before execution yields significant benefits compared to traditional shared-everything protocols.

5.4 Proposals for High Contention

We next evaluate STRIFE against OLTP design proposals for handling high contention workloads.

MOCC [39] is an OCC scheme that combines validation-based optimistic execution with pessimistic locking for hot records identified dynamically using temperature statistics.

PartCC [13] adopts a partitioned architecture exemplified by H-Store [15] and Hyper [17] where threads acquire partition-level locks before executing a transaction.

Dora [1, 24] is based on a thread-to-data assignment that logically partitions the database similar to PartCC, but transactions flow from one thread to another as it accesses different data. As Dora was originally designed for disk-based systems, we use a recent main-memory variant [1].

Orthrus [29] uses message passing between threads instead of locks to improve scalability; threads are dedicated to either CC or execution. In our experiments, we use a 1:4 ratio of CC and execution threads as in prior work [39].

We use source code from Dora and Orthrus, and implemented MOCC and PartCC to use STRIFE as the storage manager. A head-to-head comparison with LADS [42] was

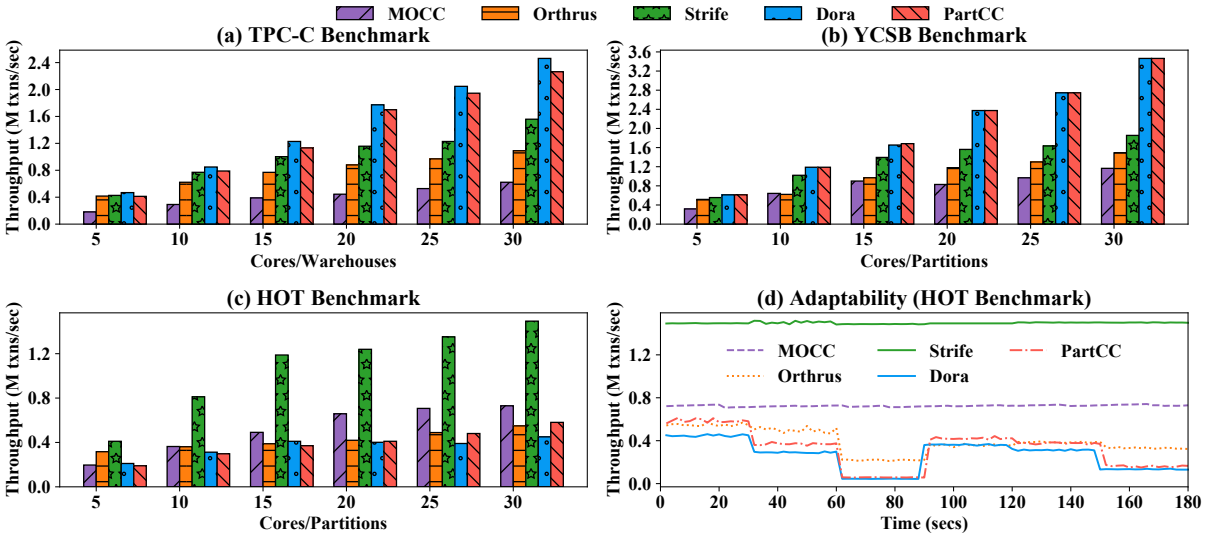


Fig. 8. Comparison against high contention proposals

not viable since we could not obtain its source code from authors; we instead compare with LADS in Sec. 6. Some of these systems assume a statically partitioned database, so we use *same* number of partitions as cores in our experiments. The results are shown in Fig. 8.

YCSB is partitionable without any cross-partition transactions. TPC-C, on the other hand, is partitionable using warehouses but with some cross-partition transactions due to remote payments and orders. In both cases, the workload is known a priori and can be leveraged by partitioned systems. Unsurprisingly, PartCC and Dora exploit this and perform better than Strife, which incurs a 40 − 50% overhead due to analysis. Dora outperforms PartCC on TPC-C marginally, since it handles cross-partition transactions better using fine-grained record-level locks. PartCC replaces multiple record-levels locks with one partition lock, but Dora uses partition-local record-level locks owned by the thread. Hence, there is no significant difference between them on YCSB. Compared to Dora and PartCC, Orthrus performs poorly due to message passing across threads even when workload is partitionable Although MOCC's analysis enables it to switch from optimistic to pessimistic locking for hot records, as we show in Sec. 5.3 even locking (LockSorted) has higher overheads compared to Strife.

Unlike TPC-C and YCSB, HOT is not amenable to static partitioning since the cold items may belong to any random partition. We use hash-based partitioning and evenly distribute hot records. This makes HOT transactions multi-partition, and PartCC, Dora and Orthrus suffer significantly from it. PartCC acquires locks on multiple-partitions, effectively reducing concurrency. Though Dora and Orthrus use fine-grained locking logically, a multi-partition transaction flows across multiple cores (sometimes NUMA sockets) during its execution. This combined with message passing significantly increase the lock hold time for hot records leading to degraded performance. MOCC is not influenced by partitionability and hence its performance is similar to that on TPC-C and YCSB. Interestingly, on a non-statically-partitionable workload, a design oblivious to partitioning (MOCC) performs better than those that exploit it (PartCC, Dora and Orthrus). Strife, on the other hand, performs a fine-grained clustering of transactions dynamically to execute them without any CC. So, Strife outperforms PartCC, Dora and Orthrus by up to 2.5× and MOCC by 2×.

In sum, while prior work for handling high contention perform well on statically partitionable workloads, they suffer on those that are not statically partitionable. Strife incurs about 40 − 50% overhead on the former, but it can robustly handle those that are not statically partitionable.

5.5 Adapting to Varying Workloads

Next, we analyze Strife's adaptability using a time-varying workload derived from the HOT benchmark. Initially, all hot records are evenly distributed across the 30 partitions and vary hot records approximately every 30secs such that they are distributed across a random number n (< 30) of partitions. The results are presented in Fig. 8(d).

PartCC performance varies significantly depending on n: when n is small, obtaining a partition-level lock reduces concurrency even further as compared to $n = 30$. Dora suffers with a smaller n as well. A Dora transaction flows from one thread to another during execution and when it is skewed, transactions are held up at skewed partitions, thus increasing lock hold time further. Orthrus, however, has a partitioned functionality. CC threads are responsible only for acquiring

and releasing locks, so its performance is relatively stable compared to Dora even with partition-induced skews. When the distribution is very skewed, even CC threads in Orthrus become a bottleneck and performance drops. MOCC is able to detect changing hot items dynamically and is not affected by partition-induced skews since it is oblivious to partitions. Strife spots hot records in a given batch by random sampling and distributes them dynamically across as many CF clusters as possible. This eliminates any partition-induced skews present in other architectures, and enables Strife to outperform all other protocols with stable throughput as it eliminates partition-induced skews.

5.6 Microbenchmarks

5.6.1 Clustering Performance. Fig. 9(a) depicts Strife analysis throughput (transactions analyzed per second) as we vary number of cores. Strife algorithm scales almost linearly until 15 cores (within single NUMA socket). Beyond 15 cores, there is a slight drop in scalability due to cross-socket communication but throughput still increases. analysis cost depends on total number of data accesses in the batch. The depicted performance (YCSB) is for 20 read-write accesses and analysis throughput is much better than most main-memory OLTP engines today. This shows that in Strife execution pipeline, analysis phase is unlikely to be the bottleneck.

Fig. 9(b) breaks down the time taken by each step during analysis for TPC-C, YCSB-Low ($\theta = 0.1$), YCSB-High ($\theta = 0.99$) and HOT. prepare step scans through read-write sets, and this is the most expensive since records are looked up for the first time and brought into cache. In subsequent steps, cached data is reused, and hence they are much faster. analysis time also depends on number of *distinct* data items accessed and cache size. In YCSB-Low, data items are accessed almost uniformly, while YCSB-High access fewer distinct data items. So, analysis of a YCSB-Low batch is more expensive than that of YCSB-High. We do not show spot and merge as they are negligible compared to these 3 steps. Our implementation caches the current set of clusters connected to a transaction in thread-local cache, which further improves fuse and allocate performance.

5.6.2 Residual Parameter. The residual parameter α plays a key role in Strife. It determines (1) the proportion of transactions executed with or without CC and (2) the number of CF clusters. A larger α allows Strife to classify more cross-cluster transactions as residuals improving the number of CF clusters; but more transactions are executed with CC, reducing the benefits of CC-free execution. A smaller α, on the other hand, may reduce concurrency in the conflict-free phase but fewer transactions are executed with CC.

Fig. 10(a) depicts the number of conflict-free clusters produced for different number of warehouses in TPC-C as we vary α. When α is zero, Strife clusters all transactions into a single cluster. However, as we increase α number of CF clusters increase and settles at the number of warehouses. Fig. 10(b) depicts the CF clusters produced for various values of θ (controls contention in the workload) in YCSB with 30 partitions. A smaller θ implies lower contention and hence Strife results in 100 special clusters, which is the maximum value set in our experiments. Note that there is minimal impact of increasing α for lower θ cases. For larger θ, each CF cluster corresponds to a YCSB partition and hence no impact.

5.6.3 Latency and Batch Size. Strife employs a *batched* execution strategy. Similar to streaming systems [3, 46], batch size poses an important trade-off between throughput and latency. With a larger batch size, the latency is higher, but has more potential for amortization and hence throughput. When the batch is small, other overheads in the system become more prominent but with a smaller latency.

We present throughput, end-to-end latency and analysis latency for TPC-C, YCSB and HOT as we vary batch size in Fig. 11 (in log scale). For small batch sizes (1K), synchronization overheads at phase boundaries affect the throughput. For larger batches, throughput is marginally higher due to better amortization and cache locality. Beyond a point, throughput does not increase since the data structures no longer fit in cache. End-to-end latency increases linearly with batch size. Although, we can cluster a 1K batch in sub millisecond latencies, the throughput is sub-optimal. In our experiments we chose a batch size of 10K since it provides a good trade-off between latency (< 5 ms) and throughput.

5.6.4 Cost of Reconnaissance Queries. When read write sets are not known a priori, Strife uses the OLLP protocol [36]: issues a read-only, low isolation level query called the reconnaissance query to acquire the entire read/write set as part of the prepare step. Additionally index lookups are cached in transaction context for later execution. A particularly common pattern of such transactions are those that perform a secondary index lookup [36]. In full TPC-C, some payment transactions obtain customer record through last name which involves a secondary index lookup. Strife executes these transactions using the OLLP protocol. Fig. 12 depicts the throughput of Strife and other CC protocols on 30 warehouses as we increase the percentage of payment transactions that require a reconnaissance query in a 50/50 mixture of payment and new-order transactions. While there is a marginal decrease in performance for other CC protocols due to secondary index lookups, the decrease is more pronounced in Strife since it performs it twice. At the same time, additional cost incurred is a single lock-free secondary index lookup, so the impact of reconnaissance query is minimal (at most 9%) and Strife still outperforms other CC protocols on high contention workloads.

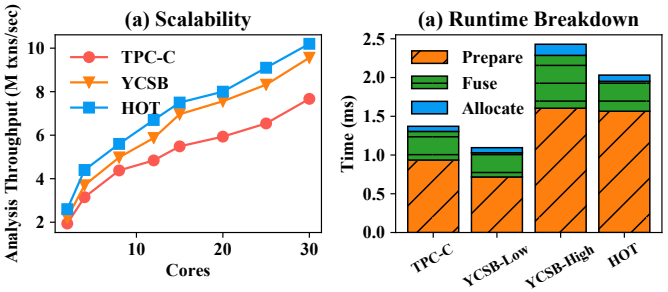


Fig. 9. Clustering Performance Experiments

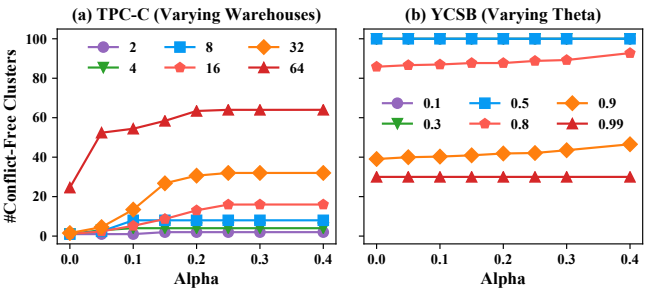


Fig. 10. Residual Parameter Experiments

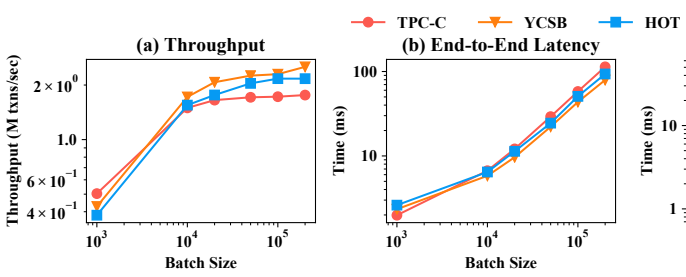


Fig. 11. Batch Size Experiments

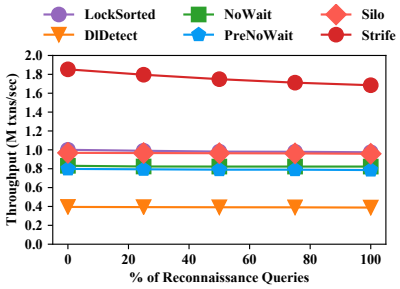


Fig. 12. Reconnaissance Queries

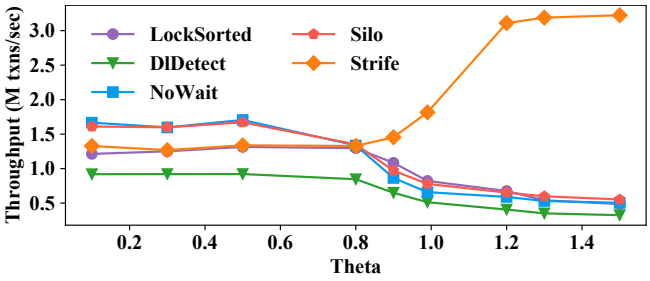


Fig. 13. Performance as Contention Varies

5.6.5 Varying Amount of Contention. We compare STRIFE against CC protocols as we vary the amount of contention. Fig. 13 shows YCSB throughput with 30 partitions on 30 cores as we vary the Zipfian constant(θ).

Though STRIFE is not designed for low contention workloads, this helps us understand the behavior of our clustering algorithm empirically. STRIFE is not universally better than other concurrency control protocols. For low contention, Silo and NoWait perform better than STRIFE. STRIFE incurs clustering overhead (3 passes over batch), while cost of acquiring a lock in these CC protocols, especially in counting-based semaphore locks [30] is quite low under low contention. As we increase θ, conflicts in the workload increase since more transactions access a few hot items, leading to increased deadlock detection or abort-restart overheads. When $\theta = 1.5$, STRIFE outperforms other protocols by 5×. ANALYSIS is inexpensive and CONFLICT-FREE exploits maximum parallelism. Compared to the overheads of executing a high contention workload with CC, ANALYSIS quickly reveals the underlying structure of the batch which is then used to schedule conflict-free clusters in parallel without any CC. In summary, STRIFE, while not universally better, yields significant benefits for highly contended workloads.

6 RELATED WORK

We review three classes of related work here: (1) OLTP systems designed for high contention; (2) Automatic database partitioning; (3) Semantics-aware optimizations that are complementary to STRIFE and could be used to improve performance under high contention.

CC Protocols. On update-intensive high contention workloads, waiting-based pessimistic protocols perform poorly due to lock thrashing, while pessimistic deadlock avoidance (e.g. NO-WAIT) and optimistic protocols [19, 20, 37] suffer from an inflated abort rate [43]. Pessimistic protocols are worse than optimistic even on conflict-free workloads with high physical contention (read-intensive) [1].

Some proposals for handling high contention include queue-based locks (e.g. MCS locks [38]), latch-free data structures [11], scalable lock manager designs [14], speculative lock inheritance [12], multi-versioning [32] and counting semaphores [30]. STRIFE executes *most* transactions without CC, thus completely eliminating any latching/locking overheads.

Optimistic protocols suffer from a high abort rate. Data-driven timestamp management [44], dynamic dependency

tracking [45] or reusing partial executions of aborted transactions [7] have addressed this partially. MOCC [39], state-of-the-art for high contention, combines pessimistic locking with optimistic validation-based execution. There are no aborts due to conflicts in Strife.

Partitioned and Deterministic Databases. H-Store [13, 15], Hyper [17] yield state-of-the-art performance on partitionable high contention workloads but are plagued by cross-partition transactions [43]. Appuswamy et al. [1] advocate for fine-grained partitioning to avoid skew induced by static partitions, which Strife eliminates and dynamically adapts to changing hot set. Calvin [31, 36] acquires locks in a deterministic order on a single thread while Strife is free from such concurrency bottlenecks and mostly avoids locks.

Alternate Architectures. Orthrus [29] delegates CC to dedicated threads and uses message passing to synchronize access. Dora [1, 24] adopts thread-to-data assignment on a partitioned database; transactions flow from one thread to another acquiring/releasing locks. Both suffer on update-intensive workload due to longer lock-hold times and worsened by cross-partition transactions and skews [1]. Strife does not use locks, adopts fine-grained partitioning, and avoids skews. Tang et al. [35] propose adaptive concurrency control (ACC) that clusters data and chooses optimal CC for each cluster using a machine learning model. Both these combine existing CC protocols to yield better performance on a mixed workload. This is unlike Strife which optimizes for and performs better than each of these CC protocols under high contention.

Automatic Database Partitioning. Schism [6] partitions the co-access graph among data items and uses the partition to infer partition predicates statically; Horticulture [25] automatically partitions a database using local neighborhood search on the design space with an analytical cost model. While it seeks to reduce partition-induced skews and cross-partition transactions similar to Strife, its solution space is limited to hash/range partitioning. Sword [28] is an extension of Schism that uses hyper-graph compression to partition a database also with incremental repartitioning. All these approaches are designed for offline computation and are not efficient enough for dynamic clustering in Strife.

Transaction Decomposition. Callas [41] partitions a transaction into groups that are executed independently under different CC protocols. IC3 [40] tracks run time dependencies allowing transactions to be chopped into finer pieces than possible statically. LADS [42] breaks a transaction into a set of record actions using their semantics. Overall, decomposing a transaction into smaller pieces can help alleviate reduce contention by reducing effective lock hold time on hot data items. LADS adopts a similar graph-based partitioning strategy on batches of transactions as Strife, but there are several key differences: LADS does not entirely eliminate CC during execution since it synchronizes on logical dependencies between record actions of a transaction. A user initiated abort will lead to cascading aborts in LADS due to speculative execution of record actions. Further, graph partitioning algorithm in LADS initially assumes a range-based partition and iteratively removes cross-edges by relocating records. This does not perform well on workloads not suitable for range-based partitioning, such as the HOT benchmark.

7 CONCLUSIONS

We presented Strife, a transaction processing protocol for high-contention workloads. Strife identifies hot records in the database during runtime, clusters transactions that access them together, and executes them without concurrency control. Strife achieves this with an algorithm that exploits random sampling and efficient union-find data structures to cluster thousands of transactions in milliseconds. Our experiments have shown that Strife can achieve substantial performance improvement over conventional CC protocols; while it incurs about 50% overhead over partitioned systems on a statically partitionable workload, Strife can improve performance by up to 2× when such static partitioning is not viable and when workload changes over time.

ACKNOWLEDGEMENTS

This work is supported in part by the NSF grants IIS-1907997, AITF-1535565, IIS-1546083, IIS-1651489; the Intel-NSF CAPA Center; DOE award DE-SC0016260; and gifts from Adobe and Google. We also thank Xiangyao Yu and the anonymous reviewers for their valuable comments and feedback.

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. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 121–134. https://doi.org/10.14778/3149193.3149194

[2] Michael S Bernstein, Andrés Monroy-Hernández, Drew Harry, Paul André, Katrina Panovich, and Gregory G Vargas. 2011. 4chan and/b: An Analysis of Anonymity and Ephemerality in a Large Online Community.. In *ICWSM*.

[3] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 401–412. https://doi.org/10.14778/2735496.2735503

[4] 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 '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[5] T. P. P. Council. 2018. TPCC-C Standards Specification. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf. [Online; accessed 19-Dec-2017].

[6] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 48–57. https://doi.org/10.14778/1920841.1920853

[7] 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 (SIGMOD '17)*. ACM, New York, NY, USA, 235–250. https://doi.org/10.1145/3035918.3035919

[8] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[9] Michael T. Heath and Padma Raghavan. 1995. A Cartesian Parallel Nested Dissection Algorithm. *SIAM J. Matrix Anal. Appl.* 16, 1 (Jan. 1995), 235–253. https://doi.org/10.1137/S0895479892238270

[10] Bruce Hendrickson and Robert Leland. 1995. A Multilevel Algorithm for Partitioning Graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (Supercomputing '95)*. ACM, New York, NY, USA, Article 28. https://doi.org/10.1145/224170.224228

[11] Takashi Horikawa. 2013. Latch-free Data Structures for DBMS: Design, Implementation, and Evaluation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 409–420. https://doi.org/10.1145/2463676.2463720

[12] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. 2009. Improving OLTP Scalability Using Speculative Lock Inheritance. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 479–489. https://doi.org/10.14778/1687627.1687682

[13] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 603–614. https://doi.org/10.1145/1807167.1807233

[14] Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Y. Yeom. 2013. A Scalable Lock Manager for Multicores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/2463676.2465271

[15] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[16] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (Dec. 1998), 359–392. https://doi.org/10.1137/S1064827595287997

[17] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, Washington, DC, USA, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[18] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1675–1687. https://doi.org/10.1145/2882903.2882905

[19] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 691–706. https://doi.org/10.1145/2723372.2746480

[20] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226. https://doi.org/10.1145/319566.319567

[21] Dominique LaSalle, Md Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Pradeep Dubey, and George Karypis. 2015. Improving Graph Partitioning for Modern Graphs and Architectures. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (IA3 '15)*. ACM, New York, NY, USA, Article 14, 4 pages. https://doi.org/10.1145/2833179.2833188

[22] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 27, 14 pages. https://doi.org/10.1145/2592798.2592820

[23] Gary L. Miller, Shang-Hua Teng, William Thurston, and Stephen A. Vavasis. 1993. Automatic Mesh Partitioning. In *Graph Theory and Sparse Matrix Computation*, Alan George, John R. Gilbert, and Joseph W. H. Liu (Eds.). Springer New York, New York, NY, 57–84.

[24] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 928–939. https://doi.org/10.14778/1920841.1920959

[25] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD '12: Proceedings of the 2012 international conference on Management of Data*. 61–72. http://hstore.cs.brown.edu/papers/hstore-partitioning.pdf

[26] Andrew Pavlo, Evan P. C. Jones, and Stanley Zdonik. 2011. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *Proc. VLDB Endow.* 5, 2 (Oct. 2011), 85–96. https://doi.org/10.14778/2078324.2078325

[27] Alex Pothen, Horst D. Simon, and Kan-Pu Liou. 1990. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM J. Matrix Anal. Appl.* 11, 3 (May 1990), 430–452. https://doi.org/10.1137/0611030

[28] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: Scalable Workload-aware Data Placement for Transactional

Workloads. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. ACM, New York, NY, USA, 430–441. https://doi.org/10.1145/2452376.2452427

[29] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1583–1598. https://doi.org/10.1145/2882903.2882958

[30] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2013. Lightweight locking for main memory database systems. In *Proceedings of the 39th international conference on Very Large Data Bases (PVLDB'13)*. VLDB Endowment, 145–156. http://dl.acm.org/citation.cfm?id=2448936.2448947

[31] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2014. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *Proc. VLDB Endow.* 7, 10 (June 2014), 821–832. https://doi.org/10.14778/2732951.2732955

[32] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. 2014. Reducing Database Locking Contention Through Multi-version Concurrency. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1331–1342. https://doi.org/10.14778/2733004.2733006

[33] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms* (4th ed.). Addison-Wesley Professional.

[34] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, Washington, DC, USA, 2–11. https://doi.org/10.1109/ICDE.2005.1

[35] Dixin Tang, Hao Jiang, and Aaron J. Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. http://cidrdb.org/cidr2017/papers/p63-tang-cidr17.pdf

[36] 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 (SIGMOD '12)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[37] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[38] Tianzheng Wang, Milind Chabbi, and Hideaki Kimura. 2016. Be My Guest: MCS Lock Now Welcomes Guests. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 21, 12 pages. https://doi.org/10.1145/2851141.2851160

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

[40] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1643–1658. https://doi.org/10.1145/2882903.2882934

[41] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 279–294. https://doi.org/10.1145/2815400.2815430

[42] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting Single-Threaded Model in Multi-Core In-Memory Systems. *IEEE Trans. on Knowl. and Data Eng.* 28, 10 (Oct. 2016), 2635–2650. https://doi.org/10.1109/TKDE.2016.2578319

[43] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[44] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1629–1642. https://doi.org/10.1145/2882903.2882935

[45] 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. https://doi.org/10.14778/2904121.2904126

[46] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737