A Study of Database Performance Sensitivity to Experiment Settings

Yang Wang
The Ohio State University
wang.7564@osu.edu

Miao Yu
The Ohio State University
yu.3053@buckeyemail.osu.edu

Yujie Hui
The Ohio State University
hui.82@buckeyemail.osu.edu

Fang Zhou
The Ohio State University
zhou.1250@buckeyemail.osu.edu

Yuyang Huang
The Ohio State University
huang.2928@buckeyemail.osu.edu

Rui Zhu
The Ohio State University
zhu.2455@buckeyemail.osu.edu

Xueyuan Ren
The Ohio State University
ren.450@buckeyemail.osu.edu

Tianxi Li
The Ohio State University
li.9443@buckeyemail.osu.edu

Xiaoyi Lu
University of California, Merced
xiaoyi.lu@ucmerced.edu

ABSTRACT

To allow performance comparison across different systems, our community has developed multiple benchmarks, such as TPC-C and YCSB, which are widely used. However, despite such effort, interpreting and comparing performance numbers is still a challenging task, because one can tune benchmark parameters, system features, and hardware settings, which can lead to very different system behaviors. Such tuning creates a long-standing question of whether the conclusion of a work can hold under different settings.

This work tries to shed light on this question by reproducing 11 works evaluated under TPC-C and YCSB, measuring their performance under a wider range of settings, and investigating the reasons for the change of performance numbers. By doing so, this paper tries to motivate the discussion about whether and how we should address this problem. While this paper does not give a complete solution—this is beyond the scope of a single paper, it proposes concrete suggestions we can take to improve the state of the art.

PVLDB Reference Format:
Yang Wang, Miao Yu, Yujie Hui, Fang Zhou, Yuyang Huang, Rui Zhu, Xueyuan Ren, Tianxi Li, and Xiaoyi Lu. A Study of Database Performance Sensitivity to Experiment Settings. PVLDB, 15(7): 1439 - 1452, 2022.
doi:10.14778/3523210.3523221

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sam1016yu/DB-Exp-Sensitivity.

1 INTRODUCTION

This paper targets a long-standing problem in evaluating research prototypes: how sensitive are their evaluation results to experiment settings and will their conclusions hold under a different setting? While we believe this problem is known in our community, this paper, through quantitative measurement, tries to illustrate the extent of this problem and open the discussion whether and how we should address this problem.

To allow a fair performance comparison across different systems, our community has developed multiple benchmarks, such as TPC-C [84] and YCSB [20], which are widely used to evaluate numerous works [14, 43, 49, 50, 53, 54, 60, 79, 86, 89, 94].

However, despite such effort, we observe interpreting and comparing performance numbers under the same benchmark is still a challenging task: a benchmark often has multiple tunable parameters, whose values could have a drastic impact on the behavior of the benchmark; the target system often has its own tunable parameters or features, which could interact with benchmark parameters in complicated ways; with many tunable parameters, measuring all possible settings is time-consuming, and presenting all results could take a lot of space in an article. As a result, many articles compare to prior works under certain settings, and we sometimes wonder whether their conclusions can still hold under other settings.

This work tries to shed light on this question with the following methodology: we choose TPC-C and YCSB, two of the most widely used benchmarks; we have reproduced 11 works (Table 1) evaluated under these two benchmarks; we then tune benchmark parameters and system features to see how such tuning can change the performance of these works; finally we analyze the reasons when we see a significant change in performance numbers.

We highlight our key findings and contributions:

- We find the conclusions of many works are indeed sensitive to experiment settings. For example, we observe that the contention level of TPC-C has a critical impact, and it often happens that a work can significantly outperform others under one contention level but has less or no improvement under another level.
- We analyze how experiment settings can affect evaluation results. For example, we observe, in TPC-C, introducing wait time makes the experiments I/O intensive; removing wait time makes the experiments CPU/memory intensive; further reducing the number of warehouses makes the experiments contention intensive. In summary, we find TPC-C and YCSB can be tuned to stress test almost every key component of a computer system: while such versatility is helpful to measure a variety of systems, it brings challenges to how to fairly compare different systems.

Proceedings of the VLDB Endowment, Vol. 15, No. 7 ISSN 2150-8097.
doi:10.14778/3523210.3523221

Table 1: Overview of the systems we investigate in this paper: * means the feature is discussed in the original article but not used in our reproduction, either because the feature is not implemented in the prototype or we failed to reproduce it. 2PC=Two-phase commit [37]; 2PL=Two-phase locking [11]; OCC=Optimistic Concurrency Control [67].

Name	Network	Sharding	Replication	Persistence	API	Competitors
Calvin [72, 78, 79]	TCP	Yes	Primacy-backup*	Yes*	Customized	2PC
Silo [74, 86]	No	No	No	Yes*	Txn-KV	PartitionedStore [77]
HERD [41, 43]	RDMA	Yes	No	No	KV	FaRM [27], Pilaf [59]
MICA [49, 58]	DPDK	No	No	No	KV	Memcached [57], MemC3 [31], Masstree [55], RAMCloud [65]
DrTM [28, 89]	RDMA	Yes	No	No	Txn-KV	Calvin
TAPIR [93, 94]	UDP	Yes	Paxos-like	No	Txn-KV	2PL, OCC
Janus [42, 60]	TCP	Yes	Paxos-like	No	Txn-KV	2PL, OCC, TAPIR
Cicada [17, 50]	No	No	No	No	Txn-KV	Silo, TicToc [92], FOEDUS [45], MOCC [88], ERMIA [44], Hekaton [24, 48], 2PL
GAM [14, 35]	RDMA	Yes	No	No	Txn-KV	FaRM [27], L-Store [51], Tell [52]
Star [54, 76]	TCP	Yes	Primacy-backup	Yes*	Txn-KV	2PL, OCC, Calvin
Aria [9, 53]	TCP*	Yes*	Yes*	No	Txn-KV	BOHM [30], PWV [29], Calvin

- Based on these findings, we discuss possible solutions and raise open questions in the following two directions: first, we could encourage researchers to perform experiments extensively, but whether this will be too time-consuming and how to present results remains open questions; second, we could encourage researchers to use some "realistic" settings, probably from the study of industrial system, but this approach may discourage research in new directions that do not target existing settings. While we don't have complete answers to these questions, we propose concrete suggestions to improve the state of the art.
- We discuss several other questions motivated by our study, like how to mitigate the weakness of TPC-C when testing concurrency control mechanisms and potential new research directions.

2 METHODOLOGY

We surveyed research papers published in top system and database venues (i.e., VLDB, SIGMOD, OSDI, SOSP, etc) in the last decade, and found TPC-C and YCSB are the most popular benchmarks. Then among the works that use these two benchmarks, we selected those that are 1) open-source and 2) do not require special hardware that is not available to us. Then we tried to reproduce the results of these works and we succeeded in reproducing 11 of them, as listed in Table 1. Since our hardware is not exactly the same as those used by these works, we consider "success" as 1) our reproduced numbers are reasonably close to those in the corresponding article or have an explainable deviation, and 2) the conclusion of the article still holds. Note that Table 1 is by no means a complete list of re-producible works: we chose these works since they cover a variety of design choices, and given more time, we probably can reproduce more.

We then tune benchmark parameters and system features to see how they affect the performance of these works; finally we analyze the reasons when we see a significant change of performance.

In this paper, we mainly focus on the throughputs of these works. And we mainly focus on significant factors that can cause at least several times of difference in throughput.

Overview of reproduced works. Silo [86] and Cicada [50] mainly focus on building efficient in-memory transaction processing engines. In particular, Cicada tries to outperform or match prior engines under a variety of workloads.

Calvin [72, 79], DrTM [89], Star [54], and Aria [53] focus on distributed transactions. To avoid the expensive two-phase commit (2PC) protocol, Calvin builds a deterministic scheduler to schedule the operations of different transactions. Star runs multi-shard transactions on a replica that holds all data, and runs single-shard transactions on other partial replicas. To address the limitation that deterministic schedulers like Calvin need to know the read/write set of transactions before execution, Aria builds a deterministic and serializable transaction processing engine. DrTM accelerates distributed transactions by utilizing transactional memory and RDMA.

TAPIR [94] and Janus [60] focus on geo-distributed databases which replicate data with Paxos-like protocols [46, 47]. TAPIR observes that 2PC and Paxos are redundant to some extent and thus proposes a protocol to merge 2PC and Paxos. Janus reduces the number of round trips with a fast-path protocol and reduces the abort rate by building a global dependency graph and re-ordering transactions that violate serializability.

GAM [14] builds a share-memory engine based on RDMA. Based on this engine, it can treat distributed transactions in the same way as a local transaction.

MICA [49] and HERD [43] focus on building efficient key-value stores by utilizing DPDK and RDMA respectively.

In our experiments, we use the source codes from the original authors and have slightly modified these source codes to either fix a bug or change a fixed parameter. We have published our modified source codes on our artifact website [21].

Testbed and results. We run our experiments on CloudLab [18] and Ohio Supercomputer Center [15]. We try to use hardware settings that are close to the ones used by the original articles. Due to space limitations, we document our hardware settings and the ones used by original works in our artifact website [22].

For Aria, Cicada, Silo YCSB, Calvin, Star, and HERD, our reproduced numbers are within the 75% to 125% range of the original numbers. For Silo TPC-C, Janus, TAPIR, and DrTM, our reproduced numbers are about 1.5x to 2.5x better than the original numbers, mainly because our CPUs or NICs are better. In particular, original experiments of TAPIR and Janus were carried out on Google Compute Engine and Amazon Web Services (AWS) respectively, and our experiments were carried out on CloudLab bare-metal machines. For GAM, our reproduced numbers are about 30% to 50% of the original numbers, because our CPUs are less powerful. For MICA, our reproduced numbers are 15% of the original numbers, since the original experiments utilize eight connections while ours only has one. In all cases, we observe the deviation is explainable and the shapes of our reproduced figures are similar to the original ones.

Due to space limitations, this paper will only show part of the results. When presenting results in figures, we try to use a format that is close to the one used in the original article: this makes the figures' formats inconsistent across this paper, but we believe this is helpful to show how the conclusions of the original articles can change. **Due to this choice and the nature of this work, some of our figures look similar to the figures in the original articles, but all numbers are from our reproduced experiments.**

3 ANALYSIS OF TPC-C RESULTS

To help the understanding of TPC-C numbers, we first present the design of TPC-C. In particular, we highlight the parameters that are often tuned by different works.

TPC-C simulates the database of a wholesale company. It includes a number of warehouses, each maintaining stocks for 100,000 items and covering 10 districts with 3,000 customers in each. It consists of nine tables (WAREHOUSE, DISTRICT, CUSTOMER, HISTORY, ORDER, NEW-ORDER, ORDER-LINE, STOCK, and ITEM) and five types of transactions:

- *New-Order*: it simulates the procedure of entering an order. It randomly selects a district from a warehouse, randomly selects 5 to 15 items, and randomly selects a quantity of one to ten for each item. For each item, it has 1% chance to order from another warehouse. This means, with an average of 10 items per order, **about 9.5% of the orders need to access remote warehouses.** This transaction retrieves and increments a D_NEXT_O_ID value (i.e, the next available order number) from the DISTRICT table, uses the value to create an order, and inserts a row in both the NEW-ORDER and ORDER table. Then for each item in the order, it retrieves the price from the ITEM table and updates the count in the STOCK table. Finally it inserts a new row for each item in the ORDER-LINE table. There is a 1% chance that the item is not found or its count is not sufficient, which will cause this transaction to rollback. **The D_NEXT_O_ID value is a major contention point, since all *New-Order* transactions of the same district needs to update the same D_NEXT_O_ID.**
- *Payment*: it updates the customer's balance and reflects the payment on the district and warehouse sales statistics. It randomly selects a district from a warehouse, randomly selects a customer from the district, and updates the balance and payment values in the corresponding tables. **There is a 15% chance that the customer's resident warehouse is a remote warehouse.**
- *Order-Status*: it queries a customer's last order. It randomly selects a district from a warehouse, randomly selects a customer from the district, and selects the customer's order with the largest order ID in the ORDER table. Then it selects matching rows from the ORDER-LINE table.
- *Delivery*: given a district, it selects the oldest order from the NEW-ORDER table, deletes this row, retrieves detailed information from the ORDER table, and then updates the ORDER-LINE table. It finally updates the balance in the CUSTOMER table.
- *Stock-Level*: it finds the recently sold items that have a stock level below a specified threshold. Given a district, it first retrieves the D_NEXT_O_ID from the DISTRICT table and then selects order lines whose order IDs are greater than or equal to D_NEXT_O_ID - 20 from the ORDER-LINE table. For each item in these order lines, this transaction checks whether its quantity in the STOCK table is less than a threshold.

The TPC-C specification further specifies **the distribution of these five types of transactions**: *New-Order* 45%, *Payment* 43%, *Order-Status* 4%, *Delivery* 4%, and *Stock-Level* 4%. Because of randomness in the workload, a small deviation from such distribution is allowed.

TPC-C adds **a wait time**, including a keying time and a think time, before each transaction to simulate the user's behavior of typing keyboard and thinking before making a decision. **Each warehouse has ten terminals**, one for each district, which means each warehouse can have at most ten concurrent transactions.

Note that vanilla TPC-C does not allow tuning of these parameters, except the number of warehouses. However, research prototypes often tune some of them and we analyze the reasons and effects of such tuning in this section.

3.1 Questions Raised by Reported Numbers

Table 2 shows the TPC-C setting and reported throughput of the works we have reproduced. It further shows the related information of OceanBase and Oracle, two commercial database systems which reported the top 2 highest throughput numbers by the end of 2021 [83]. We further measure H2 [38], an open-source in-memory database, and MySQL [61], with the TPC-C implementation in OLTPBench [25, 63].

As one can see in this table, the settings and the reported throughput of different systems vary drastically. This may raise many questions. For example,

- Considering commercial systems are tested with "wait time" and research prototypes are not, how much difference does it make?
- Comparing to commercial systems, most research prototypes are tested with a small number of warehouses. What is the impact of the number of warehouses?
- Different prototypes are evaluated under different degrees of concurrency. Will their conclusions still hold under a different degree of concurrency?
- What is the difference between running transactions as stored procedures and running them as interactive SQL transactions?
- Considering some research prototypes only run two types of transactions (i.e. *New-Order* and *Payment*), what is the impact of the remaining three types?

The rest of this section will try to answer these questions.

Table 2: TPC-C results reported by different works: SP=Stored Procedure; WH=warehouses; concurrency is defined as the maximal number of concurrent transactions per warehouse: if a worker thread blocks when contention happens, concurrency is computed as $\frac{\#workers}{\#warehouses}$; if a worker thread switches to another client when contention happens, concurrency is computed as $\frac{\#clients}{\#warehouses}$.

Name	SP	Wait	#Txn-Types	#WH	Concurrency	Cross-WH (%)	#Servers	Throughput (trans/sec)
Research								
Calvin	Yes	No	2	20*#Servers	1:20	0-100%	8	70K-120K
Silo	Yes	No	5	1-32	1:1	0-100%	1	0.1M-0.8M
DrTM	Yes	No	5	8*#Servers	1:1	0-100%	6 - 24	1.3M - 5.3M
Janus	Yes	No	5	6	1:6-10K:6	Default	9	100-10K
Cicada	Yes	No	5	1-28	1:4-28:1	Default	1	0.1M-6.5M
GAM	Yes	No	2	4*#Servers	1:1	0-100%	8	50K - 420K
Star	Yes	No	2	12*#Servers	1:1	0-100%	4	700K-1.6M
Aria	Yes	No	2	1-180	1:1	0-100%	1-8	20K-1.8M
Commercial								
OceanBase [7]	Yes	Yes	5	56M	10:1	Default	1557	26.2M
Oracle [64]	Yes	Yes	5	2.43M	10:1	Default	27	1.1M
Open-source								
H2-Mem [38]	No	No	5	50	2.5:1	Default	1	5180
MySQL-Disk [61]	No	No	5	8	2.5:1	Default	1	363

3.2 Impact of Wait Time

As discussed previously, vanilla TPC-C adds a wait time (i.e., keying and think time), which is at the level of several to tens of seconds, before each transaction. The existence of wait time, combined with the restriction that each warehouse can have only ten concurrent users, means that the maximal throughput we can achieve on each warehouse is limited.

To be precise, we compute the maximal throughput per warehouse as follows: given the distribution of the five types of transactions and the average keying and think time for each type of transaction (i.e., Section 5.2.5.7 in the TPC-C specification version 5.11.0 [82]), we can compute the average wait time per transaction as $(18+12)\times 45\% + (3+12)\times 43\% + (2+10)\times 4\% + (2+5)\times 4\% + (2+5)\times 4\% \approx 21$ seconds. This means the average throughput per user is about $\frac{1}{21} \approx 0.048$ transactions/second. Considering we can have 10 concurrent users per warehouse, we can achieve a throughput of 0.48 transactions/second per warehouse. One can see this number is consistent with the numbers reported by OceanBase ($\frac{26.2M}{56M} \approx 0.47$) and Oracle ($\frac{1.1M}{2.43M} \approx 0.45$).

This very low limitation on the throughput per warehouse means that to get a higher throughput, an experimenter must use many warehouses: that's why OceanBase and Oracle use millions of warehouses. For such a workload, which stores a large amount of data but has low throughput requirement per GB of data, storing data in SSDs can meet the throughput requirement and is much cheaper than storing all data in DRAM, and this is consistent with the the hardware configuration of OceanBase and Oracle.

As a result, vanilla TPC-C is essentially an I/O intensive benchmark with a low contention level: because of the long wait time and the restriction that each warehouse can have only 10 concurrent users, a warehouse is idle in most of the time, which results in a low probability of multiple transactions accessing the same warehouse at the same time.

On the other hand, most research works in Table 2 focus on addressing contentions. To evaluate their effectiveness, they have to remove the wait time so that they can gain a high throughput per warehouse and, as a result, a high contention level. For the same reason, they cannot use many warehouses, since more warehouses will reduce the chance that two transactions contend.

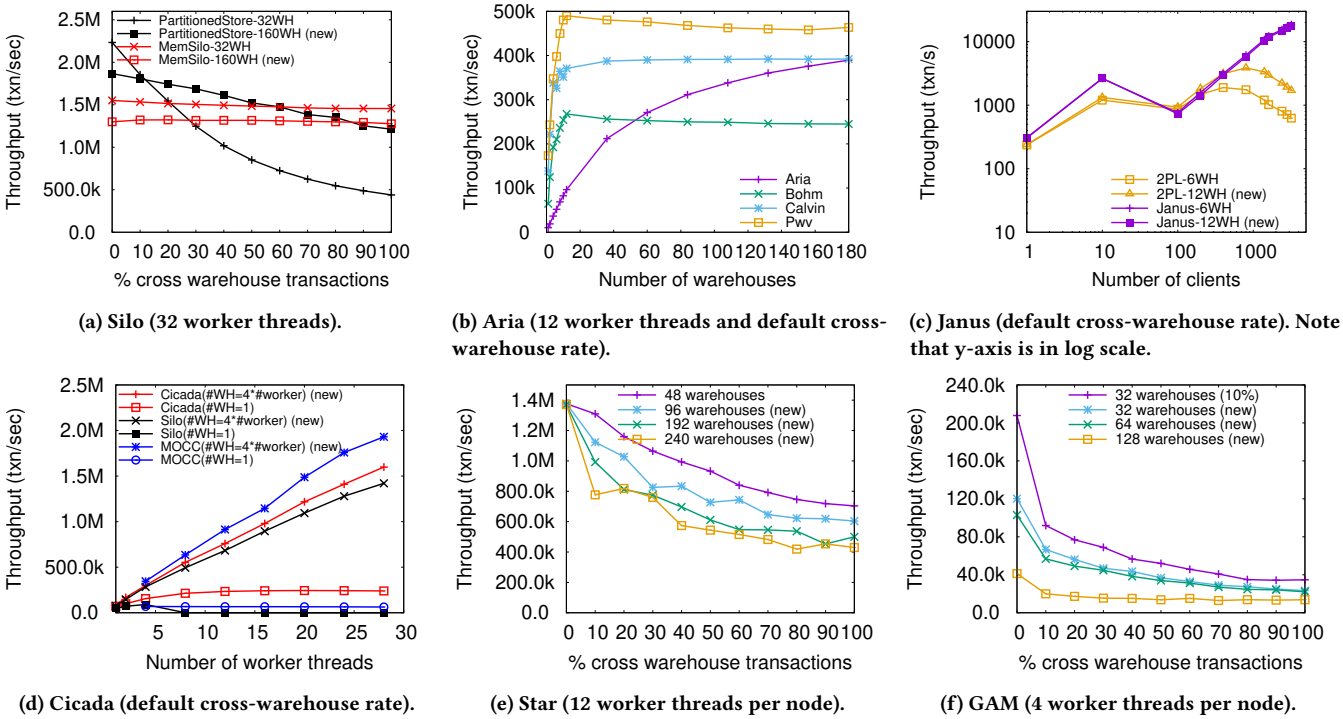
It is not the purpose of this paper to determine which setting is more important or realistic. At the very least, as shown in the above analysis, **TPC-C numbers with and without wait time are not comparable**, since they may be stress testing different components in the system.

3.3 Impact of Contention Level

Since many works focus on handling contentions, this section further discusses how different TPC-C settings can affect the contention level and the performance.

In TPC-C, the contention level can be determined by two factors. First, the number of concurrent users per warehouse has an obvious impact on the contention level and we call this number "concurrency degree". We observe some systems let a worker thread block if it cannot acquire a lock: in this case, we use $\frac{\#workers}{\#warehouses}$ to represent its concurrency degree; some systems let a worker thread switch to another transaction if it cannot acquire a lock: in this case, we use $\frac{\#clients}{\#warehouses}$ to represent its concurrency degree. Table 2 shows the concurrency degree used by different works.

Second, the *New-Order* transaction and the *Payment* transaction have a chance to access remote warehouses (called "cross-warehouse transactions" in many articles). Compared to single-warehouse transactions, cross-warehouse transactions access more warehouses and thus have a higher chance to contend with another transaction. Therefore, increasing the ratio of cross-warehouse transactions will increase the chance of contention as well.



(a) Silo (32 worker threads).

(b) Aria (12 worker threads and default cross-warehouse rate).

(c) Janus (default cross-warehouse rate). Note that y-axis is in log scale.

(d) Cicada (default cross-warehouse rate).

(e) Star (12 worker threads per node).

(f) GAM (4 worker threads per node).

Figure 1: Changing the number of warehouses, the percentage of cross warehouse transactions, and the number of workers/-clients for different systems. Lines with (new) tags are those not existed in the original articles.

We tune the number of warehouses, the number of worker threads or concurrent users, and the percentage of cross-warehouse transactions for different systems and show part of results in Figure 1. Since many systems assume data can fit into DRAM, we make sure all warehouses can fit into DRAM when increasing the number of warehouses. We make two observations:

On the one hand, for experiments in which contention is the bottleneck—this is usually represented by underutilized CPUs during the experiments, increasing the number of warehouses will increase their throughputs, due to the fact that more warehouses will reduce contention level and thus improve CPU utilization. For example, for PartitionedStore (Figure 1a), which is a system to simulate H-Store/VoltDB [77] by assigning a lock to each warehouse, using 160 warehouses can improve its throughput by up to three times compared to using 32 warehouses. For Aria (Figure 1b), using 180 warehouses can improve its throughput by two times compared to using 32 warehouses. For 2PL (Figure 1c), using 12 warehouses can almost double the throughput compared to using 6 warehouses. For Cicada and its competitors (Figure 1d), using four warehouses per thread can significantly improve the throughputs of all systems compared to using a total of one warehouse. Note that we do not use an exceptionally large number of warehouses, considering all data can fit into the DRAM of an inexpensive commodity server.

On the other hand, for experiments in which contention is not the bottleneck, increasing the number of warehouses will not help and may even decrease their throughputs due to various reasons. For example, as shown in Figure 1a, Silo's throughput slightly decreases when increasing the number of warehouses from 32 to 160 and we observe a similar slight decrease for DrTM's throughput when increasing the number of warehouses per node from 8 to 32. And we observe a moderate decrease of Star's throughput (Figure 1e) and a quite significant decrease of GAM's throughput (Figure 1f). We use *perf* [68] to profile GAM and find, with more warehouses, GAM spends more time fetching data from remote machines, because its caching mechanism becomes less efficient.

As a result, **many works' conclusions are highly sensitive to these parameters**. For example, as shown in Figure 1a, Silo has an advantage over PartitionedStore when concurrency degree is close to 1 but such advantage diminishes when concurrency degree decreases. Similarly, as shown in Figure 1d, Cicada is 3.8 times as fast as Mostly-Optimistic Concurrency Control (MOCC) [88] when concurrency degree is 28 (i.e., one warehouse and 28 threads) but is 17% slower than MOCC when concurrency degree is 0.25 (i.e., 112 warehouses and 28 threads). On the contrary, as shown in Figure 1b, Aria has a clear advantage over Calvin when concurrency degree is lower than 0.2, since their throughput numbers are similar but Aria does not require to know the read/write set of transactions as Calvin does; with a higher concurrency degree (e.g., 1), however, Calvin outperforms Aria significantly, which makes the comparison unclear. On the other extreme, Janus shines over its competitors when the concurrency degree is above 1,000.

Again, this observation is not completely new: some of the works mentioned above, such as Cicada [50] and Aria [53], have measured and reported their sensitivity to these contention related parameters.

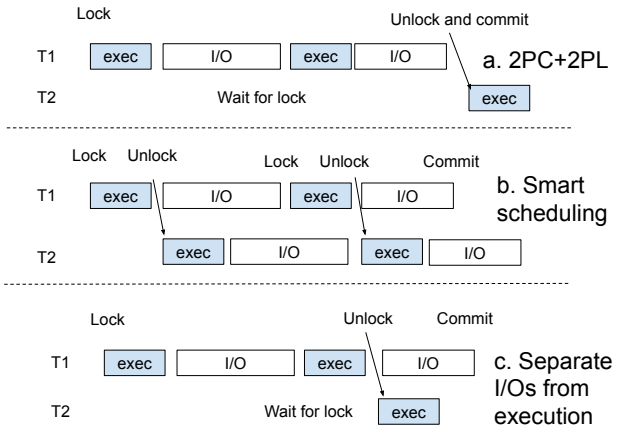


Figure 2: Reducing I/Os during a critical section.

However, by showing the prevalence of such sensitivity across a variety of systems, this paper hopes that it can motivate the discussion of whether and how we should address this problem, which is discussed in detail in Section 5.

3.4 Impact of Network and Disk I/Os

A fully-featured database system may need to incorporate network I/Os to perform protocols like 2PC and data replication (e.g. primary-backup [12, 13] or Paxos [46, 47]), and incorporate disk I/Os to persist data. Since a number of works do not incorporate these I/Os, it is a natural question to ask how much impact it would have to add these I/Os. Among the works we studied, Silo quotes the Masstree work, which reports that networked clients reduced throughput by 23% [55]. However, our investigation shows the impact is much more complicated, depending on design details.

First, we measure the network throughput on our testbeds: when using TCP over 10 Gbps Ethernet, the maximal throughput is about 1.5M-5M packets/sec with 8B packets and the round trip latency is about 50us; when using RDMA over 56 Gbps InfiniBand, the maximal throughput can reach 26M packets/sec and the round trip latency can drop to 3us. Of course these numbers may change depending on hardware and implementation so they should be viewed as a rough estimate. As one can see, if the system only needs to send one packet to start one TPC-C transaction, then TCP may be enough to support millions of transactions per second, though it will still slow down the system since processing packets consumes CPUs. However, if we further include 2PC, replication, and interactive transactions (see Section 3.5) into consideration, then the number of packets per transaction will significantly grow. For example, if we apply standard Paxos to replicate data to three replicas, the system needs to send at least four packets for each transaction. In this case, the TCP stack may not be able to support the high transaction rate of those in-memory engines and thus the TCP stack will become the bottleneck.

Second, network and/or disk latency may have a significant impact on the throughput of a contended workload: in this case, since the execution of a transaction will block contended transactions, the length of executing a transaction will be inversely proportional to the throughput of the system. Therefore, **if the length of a critical section includes network or disk latencies, the max throughput will be limited** and of course longer I/O latency will have a larger impact. Figure 2.a depicts this phenomenon: if a transaction performs I/Os while holding a lock, the system will be idle during these I/Os since it cannot execute a contented transaction. Note that if we allow many warehouses, this phenomenon will not limit throughput since we can add more warehouses to utilize idle CPUs; if we limit the number of warehouses, however, we cannot do that. Also note that OCC does not address this problem: in Figure 2.a, OCC will allow T2 to start executing earlier, but eventually it may abort T2 due to T2 conflicting with T1.

In summary, performing I/Os while holding a lock is particularly problematic for contended workload. Multiple works have tried to address this problem:

- Faster I/Os. DrTM uses RDMA for network communication, which can reduce the round trip latency by an order of magnitude. TAPIR and Janus merge 2PC and Paxos to reduce the total number of round trips to execute a transaction. Star executes multi-shard transactions on a replica with all data, completely avoiding 2PC. By reducing or eliminating I/Os, these works can naturally improve throughput.
- Smart scheduling. To achieve serializability, a classic implementation is to hold a write lock to the end of a transaction, which sometimes is unnecessary. Calvin uses a scheduler to determine when an operation can release the lock. Janus uses a dependency graph to determine whether transactions can commit. As shown in Figure 2.b, these methods can decouple I/Os from critical sections, and thus reduce the impact of network or disk latencies.
- Separate I/Os from transaction execution. Calvin and Aria replicate and persist transactions before executing them, so that the length of critical sections will not include I/O latency. This approach requires a deterministic concurrency control mechanism. Silo and Star replicate and/or persist the effects of transactions after executing them: in this case, they unlock the data items before performing replication and persistence, but hold the reply until the replication or persistence operations are complete (Figure 2.c). Both methods can decouple I/Os from critical sections.

However, we observe that none of these solutions are perfect. For example, RDMA requires more expensive hardware, more sophisticated software code, and does not help much in a geo-distributed setting; smart scheduling needs to know the read/write set of transactions before executing them; and while we can separate replication or persistence related I/Os from transaction execution, we don't know any work that can separate 2PC related I/Os.

We quantitatively measure the impact of I/O latency by injecting network latency into existing systems with *netem* [40]. Figure 3 shows the results of Janus and 2PL with 10ms and 100ms extra network latency, which has confirmed the above analysis: Janus' maximal throughput does not drop when adding extra latency because its smart scheduling technique decouples I/O latency from the critical section, though with extra latency, Janus needs more clients to fill the pipeline; 2PL, however, has significantly lower max throughput when adding extra latency, because its critical section includes I/Os.

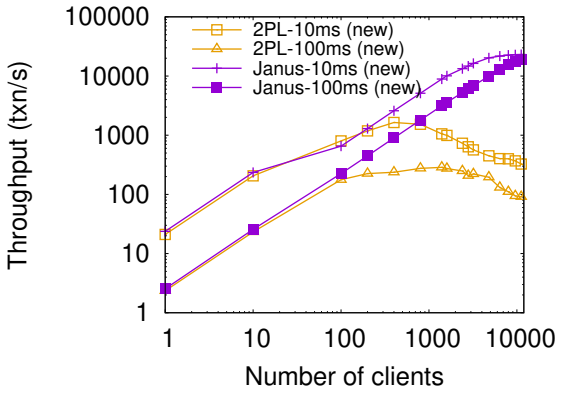


Figure 3: Throughputs of Janus and 2PL with extra network latencies (6 warehouses and default cross-warehouse rate). Note that y-axis is in log scale. Lines with (new) tags are those not existed in the original articles.

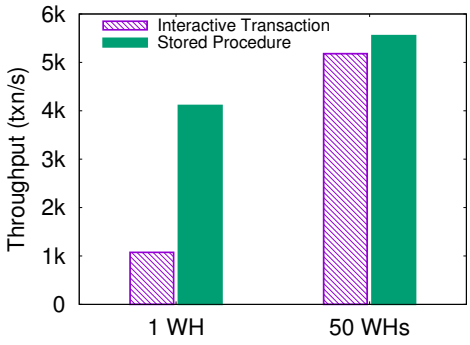


Figure 4: TPC-C throughputs of H2 with interactive transactions and stored procedures (WH=Warehouses).

3.5 Impact of Not Using Stored Procedure

All research prototypes we investigated define a transaction as a piece of C code fully executed at the server side, so that there will be no interaction between the client and the server during the execution of a transaction (i.e., stored procedure mode). In practice, however, many applications embed SQL statements into application code, and the application needs to send each statement individually to the database server (i.e., interactive mode) [66].

Using interactive SQL transactions introduces at least three challenges to existing works. First, interactive transactions make it harder, if not impossible, to infer the read/write set of a transaction before executing it, which will break the assumptions of several works. Second, interactive transactions will add a round trip of network I/Os to each statement. For example, a sample *New-Order* transaction in the TPC-C specification v5.11.0 (page 108 - page 109) includes 5+4*#items statements (i.e., 45 statements on average), excluding error handling and commit statements. As discussed in Section 3.4, it will not only incur an overhead to the network stack, but also exacerbate the "I/O latency within critical sections" problem: smart scheduling does not work with interactive transactions since it requires to know the read/write set; we don't know a way to move such I/Os out of the critical section. Finally, parsing SQL statements also incurs additional overhead.

To quantitatively measure the overhead, we compare the throughput of TPC-C with interactive transactions and that with stored procedures on the H2 database. Since the TPC-C implementation provided by OLTPBench does not provide stored procedure mode, we port its implementation to H2 as stored procedures.

As shown in Figure 4, with one warehouse, the stored procedure mode is about four times faster than the interactive mode, mainly because interactive transactions suffer from the "I/O latency within critical sections" problem since they all contend on the same warehouse. With 50 warehouses, however, the gap becomes much smaller, because multiple interactive transactions can execute in parallel on different warehouses, hiding their long I/O latencies. In this case, our profiling shows that Java Garbage Collection (GC) becomes the main bottleneck.

This set of experiments is an example of how benchmark and system parameters can affect conclusions together: under a high contention level (i.e., one warehouse), the difference between interactive transactions and stored procedures is significant; under a low contention level (i.e., 50 warehouses), their difference becomes much smaller, since the bottleneck has moved to somewhere else (e.g., GC in H2); with a highly efficient transaction engine like Silo, the gap between stored procedure mode and interactive mode may become large again since the overhead of processing packets will become relatively large compared to transaction processing.

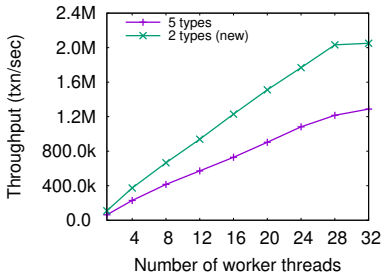
While accelerating SQL connectors with RDMA is possible [32], it only addresses part of the challenges: it is still hard to predict the read/write set of interactive transactions; while RDMA can improve the throughput of SQL connectors, it can improve the throughput of the database engines as well, so the relative slowdown caused by interactive transactions still exists. To fully address these challenges, it might be an interesting direction to investigate whether it is possible to analyze application source code to extract SQL statements and convert them into stored procedures.

3.6 Impact of Transaction Types

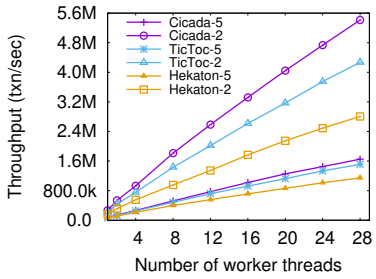
Since many works are only evaluated with two types of transactions (i.e., *New-Order* and *Payment*), we investigate whether missing three types of transactions will have a significant impact on performance.

Our method is as follows: we choose works that are evaluated with all five types and then change the percentage of *Order-Status*, *Delivery*, and *Stock-Level* to zero; we adjust the percentage of *New-Order* and *Payment* accordingly.

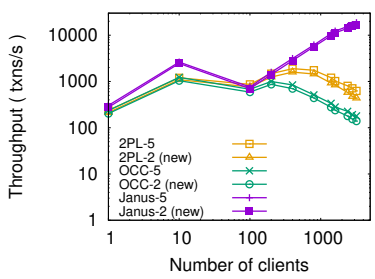
As shown in Figure 5, **removing three types has different levels of impacts on different systems**: on Silo, we observe a significant increase in throughput if using only two types of transactions (Figure 5a); Cicada reports a similar trend for other in-memory database engines as well (Figure 5b); however, we observe no significant change for Janus' throughput and its competitors (Figure 5c). The reason is as follows: comparing to *New-Order* and *Payment*, the other three types of transaction are relatively heavier (TPC-C specification describes *Payment* as "light-weight", *New-Order* as "mid-weight", but *Stock-Level* as "heavy-weight"). Therefore, if the



(a) Silo (#warehouse=#worker and no cross-warehouse transactions).



(b) Cicada (#warehouse=#worker and default cross-warehouse rate).



(c) Janus (6 warehouses and default cross-warehouse rate).

Figure 5: TPC-C throughput numbers with two and five types of transactions. Lines with (new) tags are those not existed in the original articles.

bottleneck of the database is the CPU, which is the case for those highly-optimized in-memory engines, then adding those relatively heavier transactions will have a significant impact since they consume more CPUs on average; if the bottleneck is not the CPU, which is the case for those distributed protocols in Janus, adding those heavier transactions has less impact since the system still has free CPU cycles. Furthermore, since both *New-Order* and *Payment* may be cross-warehouse, and the other three types don't, *New-Order* and *Payment* become relatively heavier in a distributed setting, which reduces the relative impacts of the remaining three types.

We don't observe a change of conclusion in existing works when removing three types because these works mainly compare to other works of the same category (i.e., in-memory vs in-memory; distributed vs distributed), which is reasonable. However, the risk of inaccuracy may increase when we perform a broader comparison.

Furthermore, note that some systems cannot support all five types: for example, Aria states "the other three transactions require range scans, which are currently not supported in our system." Adding such support may need a re-design of these systems.

3.7 Summary of Tuning TPC-C

This section summarizes how the settings of TPC-C, the target system, and hardware affects experiment results.

At a high level, more warehouses, fewer cross-warehouse transactions, fewer workers/users per warehouse, adding wait time, and short or no I/Os within a critical section will reduce the contention level. Under a low-contention setting, the throughput of the target system is usually limited by the slowest component in the system: when data are larger than DRAM size, disk I/O is usually the bottleneck; when data can be kept in DRAM, network I/Os could become the bottleneck if the system uses traditional TCP stack; for in-memory engines without a network stack or for systems with RDMA network stack, CPU or DRAM speed could become the bottleneck. Furthermore, systems that use centralized sequencers or global dependency graphs may introduce a scalability bottleneck.

On the contrary, fewer warehouses, more cross-warehouse transactions, more workers/users per warehouse, no wait time, and long I/Os with a critical section will increase the contention level. Under a high-contention setting, the throughput of the target system will be determined by the concurrency control mechanism of the target system: systems which can release locks earlier or reduce the number of aborts will have advantages in such a setting.

Using interactive transactions or running protocols like 2PC or Paxos will bring significant overhead to both settings: for the low-contention setting, they will incur many network messages, which will slow down the system if the network stack is the bottleneck; for the high-contention setting, they may add multiple round trips of network latency into a critical section, exacerbating the "I/O latency within critical sections" problem. For the latter case, systems that can move those I/Os out of the critical sections will have advantages.

As a result, the throughput numbers of different systems under different settings can vary drastically: in Table 2, on one extreme, MySQL stores data on hard drives and can only reach 363 transactions per second; on the other extreme, a single-node in-memory engine can reach millions of transactions per second; other systems can lie somewhere in the middle depending on their settings.

In summary, we can tune TPC-C to test pretty much every key component: we could use it to test disk throughput if we ensure data does not fit into DRAM; we could use it to test network stack if we run interactive transactions, 2PC, Paxos, etc and choose the number of warehouses so that they can fit into DRAM but do not cause a high contention; we could use it to test CPU speed or DRAM bandwidth if we don't test with the network stack; finally we could use it to test concurrency control if we incorporate a high contention level as discussed above. While such versatility is helpful to evaluate a variety of systems, it creates challenges to interpret performance numbers and compare different systems.

4 THE ANALYSIS OF YCSB RESULTS

Yahoo! Cloud Serving Benchmark (YCSB) is a benchmark primarily targeting systems with key-value like interface [20]. It first populates the target system with key-value pairs and then measures the system with *Insert*, *Update*, *Read*, and *Scan* operations.

YCSB defines five classes of workloads: YCSB-A uses 50% *Read* and 50% *Update* and uses Zipfian distribution to generate the keys to access; YCSB-B uses 95% *Read* and 5% *Update* with a Zipfian distribution; YCSB-C uses 100% *Read* with a Zipfian distribution; YCSB-D uses 95% *Read* and 5% *Insert*, in which *Read* always gets the latest record; YCSB-E uses 95% *Scan* and 5% *Insert*, in which *Scan*

Table 3: YCSB results reported by different works. For throughput marked as A×B, it means the system uses YCSB+T to encapsulate B operations in one transaction and achieve A transactions per second.

Name	Batching	#KV pairs	KV size	R/W ratio	Distribution	#Servers	Throughput
Star	No	2.4M	8B/100B	90:10	Uniform	4	(1.2M-3.6M) × 10
Silo	No	160M	8B/100B	80:20	Uniform	1	1M-16M
Cicada	No	10M	8B/100B	95:5/50:50	Zipfian	1	(0.2M-5.5M) × 16; (3M-55M) × 1
HERD	No	480M	16B/(4B - 1KB)	50:50	Uniform/Zipfian	13	5M - 25M
MICA	Yes	192M	8B-128B/8B-1KB	95:5/50:50	Uniform/Zipfian	1	8.6M-76.9M
TAPIR	No	1M	8B/1KB	50:50	Zipfian	6	8.5K × 1

uses the Zipfian distribution to choose the first key and uses the uniform distribution to choose the number of keys to scan. Zipfian distribution can let a large percentage of requests focus on a small number of keys and it has a parameter to tune such skewness.

Since YCSB only touches a single KV in each operation except *Scan*, it is not suitable to measure systems that support transactions. To address this problem, later works build YCSB+T [23], by encapsulating a number of YCSB operations into a single transaction.

Table 3 records the YCSB numbers reported by different systems. Again, it raises several questions:

- Since some systems have network stacks and some do not, how does network stack affect the result?
- How does the skewness of keys in the workload affect the result?
- How do the number and size of KV pairs affect the result?
- How does the read/write ratio affect the result?

4.1 Impact of Network Stack

Our overall observation is not surprising: for system processing large KVs, the bottleneck is likely to be network bandwidth; for system processing small KVs, the bottleneck is likely to be the CPU to process network packets. However, for the latter case, we can batch multiple small KVs in a single request: in this case, the bottleneck is either the network bandwidth or the in-memory engine.

We quantitatively measure the throughput of network stacks of different systems, by removing their internal data processing. We particularly focus on HERD, which uses RDMA as its network stack, and MICA, which uses DPDK as its network stack; it's well-known that the classic TCP or UDP stack is less efficient than these two.

Our experiments show that the throughput of HERD's network stack is almost identical to the throughput of HERD, which means the network stack is the bottleneck. As shown in Figure 6d (this figure includes data processing but disabling data processing generates a similar figure), the maximal throughput is around 26M ops/sec and starts to decrease when the KV size is larger than 128 bytes; for large KVs, read throughput is slightly lower than write throughput since write is mainly performed by the clients with RDMA WRITE over the Unreliable Connection protocol and read must be processed and responded with RDMA SEND over the Unreliable Datagram protocol by the server. Our profiling confirms that RDMA SEND in the server responding path is the bottleneck of YCSB read experiments. MICA's network stack is slower, unless with batching. Since MICA and HERD share the same key-value engine, the rest of this section only presents the results of HERD.

The overhead of processing packets per request can briefly classify different works: TAPIR has the lowest throughput since it uses only one thread per machine, uses the relatively slow UDP stack, and needs to send multiple packets per transaction; Star uses the slow TCP stack, but it uses multiple threads and does not need to send packets for 2PC, and thus can get much higher throughput per node. MICA and HERD further improve throughput with efficient network stacks.

4.2 Impact of Skewness

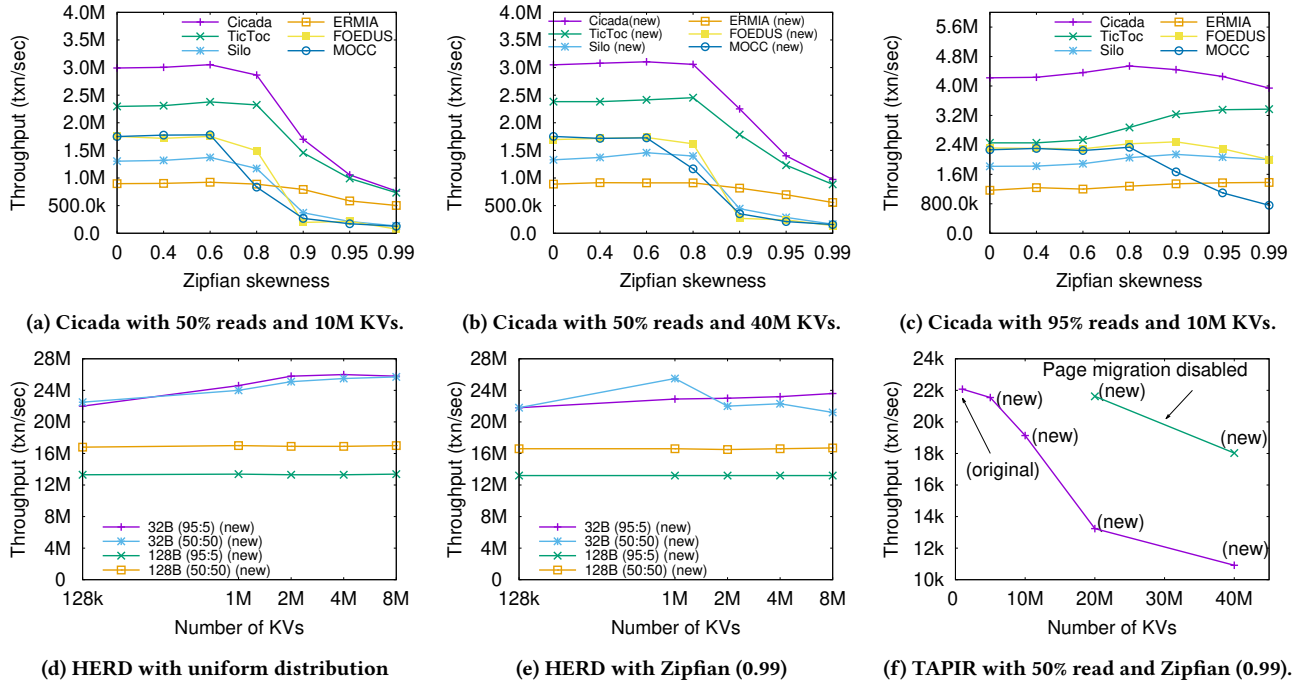
High skewness will cause YCSB to frequently access a small number of hot keys, creating a high level of contention. In practice, however, **its impact on throughput varies, depending on system design and implementation.**

For systems like Silo and Cicada, which allow any thread to access any KVs and use lock or versioning to perform concurrency control, skewness has a significant impact on throughput as shown in Figure 6a to Figure 6c. For HERD, if we compare Figure 6d and Figure 6e, we can see skewness does not have a significant impact on throughput. The reason is as follows: a HERD server starts multiple processes and lets each process be responsible for a number of key ranges; a client will send a request to the HERD process that is responsible for the key of the request. In this way, HERD actually avoids concurrency control. Under such a design, more skewness in the workload will not cause more contention but instead will cause more load imbalance among different processes [70]. However, such load imbalance does not cause a significant performance degradation due to the following reason: by default a HERD server starts six processes and we find using one process can achieve about 25% of the throughput of using six processes, which means each process has some extra processing capability to handle load imbalance.

Although the latter design (i.e., tie key ranges to threads or processes) provides better performance than the former (i.e., use concurrency control) under a skewed workload, it's unclear how the latter design can support transactions that may access multiple keys, which can be supported by the former design.

4.3 Impact of Number of KVs

In theory, contention level may be affected by the number of KVs. In practice, however, we find such impact is not significant: as shown in Figure 6d and Figure 6e, HERD's throughput is not very sensitive to the number of KVs, since HERD does not have contentions in the first place (Section 4.2); more interestingly, comparing Figure 6a



(a) Cicada with 50% reads and 10M KVs.

(b) Cicada with 50% reads and 40M KVs.

(c) Cicada with 95% reads and 10M KVs.

(d) HERD with uniform distribution

(e) HERD with Zipfian (0.99)

(f) TAPIR with 50% read and Zipfian (0.99).

Figure 6: YCSB throughput numbers of different systems under different settings. Lines or points with (new) tags are those not existed in the original articles.

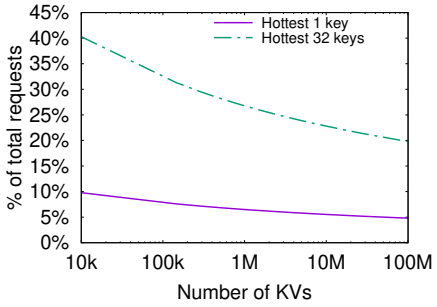


Figure 7: Contention level of different number of KVs with Zipfian distribution 0.99.

and Figure 6b , one can see Cicada's throughput is not very sensitive to the number of KVs as well, despite that Cicada's throughput is actually sensitive to contention level. The reason is as follows:

For workload with low skewness (e.g., uniform distribution), since the experiments we ran use at least 128K KVs, the contention level is very low. Therefore, although changing the number of KVs will change the contention level significantly, the contention level will not reach a level that matters to throughput, unless we use, say, tens of KVs. For highly skewed workload (e.g. Zipfian 0.99), the contention level is high, but it does not change proportionally with the number of KVs. Figure 7 shows how the frequency to access hot keys changes with the total number of KVs: changing from one million KVs to 100 million KVs will only decrease the frequency to access the hottest key from 6.5% to 4.8% and decrease the frequency to access the hottest 32 keys from 27% to 20%.

Therefore, in both cases, **the change of contention level caused by the change of the number of KVs usually has no significant impact on performance**.

On the other hand, similar as in TPC-C, the number of KVs may have other impacts: for example, for TAPIR, as shown in Figure 6f, we observe a quite significant drop in throughput when adding more KVs, and *perf* shows it is caused by page migration in Linux kernel. After disabling page migration, the throughput increases significantly, though still slightly lower than the one with a small number of KVs. Similarly, we observe Silo and Star experience modest throughput degradation when adding more key-value pairs.

4.4 Impact of Read/Write Ratio

For a skewed workload, Read/Write ratio has an impact on contention level since Read operations on the same key can execute concurrently. Its actual impact is similar as described in Section 4.2: for HERD, the impact is small (i.e., comparing Figure 6d and Figure 6e), because HERD does not have concurrency control; for systems with concurrency control, the impact is larger (e.g., comparing Figure 6a and Figure 6c).

The Read/Write ratio may have other impacts as well. For example, for Star, the 90%-read workload can achieve about 2.8M ops/second but the 50%-read workload can achieve only 0.8M ops/second, since Star needs to replicate Writes; for the same reason, TAPIR's throughput under the 50%-read workload (20K ops/second) is lower than that under the 95%-read workload (25K ops/second).

4.5 Summary of Tuning YCSB

Similar to TPC-C, we can tune YCSB to test almost every key component: if we use large KVs, the bottleneck is likely to be network bandwidth; if we use small KVs, the bottleneck is likely to be the packet rate of the network stack; batching requests or increasing contention level will move the bottleneck to the in-memory engine but in different ways: batching with low contention is likely to stress test key lookup, DRAM speed, etc; high contention is likely to stress test concurrency control. This work did not test any persistent KV stores, and it is possible that storage I/Os are bottlenecks for persistent KV stores under YCSB.

If either network bandwidth or packet rate is the bottleneck, a higher write ratio will incur more packets if the system needs to replicate writes but has no significant impact if the system does not replicate writes.

If contention is the bottleneck, a higher write ratio or a higher skewness will incur a higher contention. Unlike the number of warehouses in TPC-C, the number of KVs does not have a significant impact here, as shown in the prior analysis.

In addition, the design choice of whether to use concurrency control or to tie key ranges to processes/threads has a significant impact. As discussed in Section 4.2, the latter performs better under a highly skewed workload, and its throughput can exceed the speed of even RDMA or DPDK stacks, making these stacks the bottleneck; the former has a significant performance degradation under a highly skewed workload, in which case its throughput may be lower than the RDMA or DPDK stacks. However, it's unclear whether the latter design can support transactions accessing multiple KVs.

5 OPEN QUESTIONS AND SUGGESTIONS

5.1 What settings shall we use?

As shown in the previous sections, one can tune TPC-C and YCSB to test almost every key component in a computer system, and they can interact with system features and hardware settings in complicated ways. As a result, perhaps not surprisingly, experiment settings do have a strong impact on evaluation results and even the conclusions in some cases, which creates a well-known risk that the conclusion of a work may not hold under different settings.

We observe two major directions to alleviate this problem: the first is to encourage extensive experiments under a variety of settings; the second is to limit the values of experiment parameters. However, neither is perfect and in the rest of this section, we discuss each approach in detail and propose concrete suggestions to move forward. Finally we discuss improvement we can achieve in the short term as well. While they don't fully address the problem, we hope that they can motivate a broad discussion in our community.

Extensive testing. Encouraging researchers to run experiments under a variety of settings is a natural idea to address the problem. However, it has its own questions:

First, so far, our community has no consensus about what parameters and what values to use in these standard benchmarks. If we want to encourage extensive testing, **a discussion about which parameters should be tuned and what values should be tested is perhaps necessary**.

Second, the experiment time and cost could grow high if the experiment has many tunable parameters. That said, this may be the necessary cost for a good research. To alleviate this problem, it may be valuable to develop methods to identify "important" points in the parameter space to avoid a full blind scan.

Finally, in our experience, we observe an obstacle to extensive testing is that many systems cannot work properly under a setting that is not reported in their corresponding articles. Therefore, **making our implementation reliable under different settings is a concrete step we can take to improve the situation**. Note that today many communities have emphasized reproducibility through artifact evaluation [1–4], but so far it only focuses on reproducing results reported in an article and has nothing to do with other settings: to encourage extensive testing, it may be helpful to introduce a badge of "supporting extensive testing" in artifact evaluation.

Restricting parameter values. On the other hand, a number of impactful benchmarks, such as vanilla TPC series, SortBenchmark [5], and HPL [69], have strict specifications about how to choose the parameter values. While restricting parameter values certainly makes it easier to run experiments and compare different works, it may discourage research in new areas or new applications, whose settings may be different from these default settings.

Nevertheless, we believe knowing realistic parameter values in production systems is valuable and such study should be carried out periodically since these values may change over time. We have seen multiple studies about workloads in key-value stores [10, 62, 91], which studied read/write ratio, the distribution of data size, data skewness, etc: such information can be integrated into YCSB seamlessly. However, we also observe a pity in such studies since they miss the parameter about degree of batching, which, as shown in our YCSB study, can largely determine whether the bottleneck is the network stack or the in-memory engine: two of the studies mentioned above [10, 91] do not report this parameter at all and the remaining one [62] reports this parameter at the client side, which leaves to the readers to infer how a client-side batch may be distributed to multiple servers. Therefore, **we suggest our community to more explicitly clarify "what parameters matter", so that future studies will not miss them**. On the other hand, these studies have revealed several parameters that are not included in YCSB, such as cache miss rate and temporal change of data object size, which may help us to improve YCSB-like benchmarks.

Besides, **we are not aware of any similar studies about transaction processing systems, which should be of great value to our community**. The only relevant work we are aware of is Pavlo's survey and keynote [66], which is helpful but lacks detailed information like contention level, etc.

Better explanation of experiments settings. Apart from the two long-term efforts discussed above, we believe there are concrete improvements we can do in the short term: 1) Instead of just documenting experiment settings (e.g., our experiments use 8 warehouses and 32 threads), an article may provide an explicit explanation about the implication of these settings, e.g., what components they stress. This would be particularly helpful for non-expert readers. In addition, it would be helpful for our communities to create "tutorials" for popular benchmarks to explain the implications of

different settings. 2) An article should be precise and explicit about the conditions of its major conclusions. For example, "A is 10 times faster than B when data can fit into DRAM and the contention level is higher than 8" is more precise than "A is up to 10 times faster than B". A precise condition would help readers to understand the scope of an article and compare different works. 3) When not using a standard setting, the authors may present a justification about the setting.

5.2 Other questions

In addition, our study has raised some other related open questions:

Shall we continue using TPC-C to testing concurrency control? As discussed previously, vanilla TPC-C, with its wait time, is essentially an I/O benchmark and thus is not suitable for testing concurrency control mechanisms. Tuning TPC-C to remove wait time and use a small number of warehouses can introduce a high contention, but introduces other problems like data set size is too small or there is no locality. Therefore, though popular, TPC-C is perhaps not the ideal benchmark to test concurrency control.

TPC-E [81] has addressed some of these issues: it introduces a realistic data skew; it introduces a higher contention assuming all data can be kept in DRAM (otherwise it is still an I/O intensive benchmark), but its adoption is slow probably due to its complexity [16, 80]. For example, in TPC-E, 10 out of 12 types of transactions involve lookups and scans through non-primary indexes [80], which will pose challenges to systems that do not support scans or require the prediction of read/write set.

Again, we consider this as an open question: ultimately we will need studies from production systems to answer this question. In the short term, we propose a temporary solution by combining the ideas of TPC-C and YCSB: we can use Zipfian distribution to create a few hot warehouses in TPC-C; we don't have evidence that it represents any realistic workload, but at least it should allow us to create a high contention and a hotspot within a large dataset.

Are we emphasizing too much on a specific setting? As shown in Table 2, most works we studied focus on a specific parameter space—running transactions as stored procedures with a high degree of contention. While this scenario is certainly interesting, focusing too much on it is concerning especially since we lack any studies to confirm that this is the dominant scenario: actually Pavlo's survey [66] indicates the opposite by showing stored procedures are not frequently used for various reasons. At least we argue we should not overlook other scenarios.

6 RELATED WORK

Benchmarks. Different communities have proposed many well-received benchmarks to measure different systems, including the TPC series [85] and the SmallBank benchmark [8] to test database systems, the YCSB [20] benchmark to test NoSQL systems, the fio tool [33] to test file systems, the SPEC series [75] to test architecture, HPC, cloud and storage systems, the HPL [69], HPCG [26], and Graph500 [36] benchmarks to test supercomputer systems, the MLPerf [56, 71] and DAWNBench [19] benchmarks to test AI systems, etc. While this work focuses on TPC-C and YCSB, it would be interesting to perform a similar study on other benchmarks.

Reproducibility. In recent years, our community started paying much attention to the reproducibility of research papers. For instance, many major conferences in our field, like VLDB [4], SIGMOD [1], OSDI [3], SOSP [6], and EuroSys [2], have strong commitments to the reproducibility of research. In the meantime, many research studies focused on reproducing existing studies and exposing the reasons behind the performance variability. For example, a recent study [87] assessed the impact of network variability on cloud-based Big Data workloads and provided guidelines to reduce the volatility of performance. Fursin shared his experience of reproducing 150+ research papers and potential solutions about reproducibility [34]. Ursprung [73] presented a provenance collection system to improve the reproducibility of data science workloads.

Compared to these efforts, our work does have a strong component in reproducing prior results, but we mainly focus on comparing works under a variety of settings, some of which are beyond the ones used in the original articles. As discussed in Section 5, to support such extensive comparison, we suggest to introduce the idea of "supporting extensive testing" into artifact evaluation.

Extensive study. A number of works did extensive experiments to understand the impact of different concurrency control features [39, 90]. Since they also use TPC-C and YCSB, their observations overlap with ours to some extent (e.g., contention level matters). However, because they have a different goal compared to this work, there are a few key differences: first, for an apple-to-apple comparison, they re-implement different concurrency control features under the same framework; this work, instead, tries to understand whether changing experiment settings will change the conclusion of the original work and thus has to re-produce original works. Second, they focus more on tuning concurrency control features but less on tuning benchmark parameters and our work has the opposite focus. Finally, our work covers additional systems which explore different designs (e.g. Janus, HERD, etc). Among the works we studied, Cicada [50] did an extensive comparison, but only on in-memory database engines.

7 CONCLUSION

This paper studies how sensitive are evaluation results to experiment settings, by reproducing 11 systems under TPC-C and YCSB, measuring them under different settings, and analyzing the reasons for the change of performance numbers. By quantitatively illustrating the extent of this long-standing problem, this paper tries to motivate a broader discussion about whether and how we should address this problem. Though this paper does not propose a complete solution to this problem, it proposes concrete suggestions we can take to improve the state of the art.

ACKNOWLEDGMENTS

We thank all reviewers for their insightful comments and Spyros Blanas for his suggestions on early versions of this work. We thank Shuai Mu, Yi Lu, Xingda Wei, and Tinggang Wang for their help to reproduce prior works. This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1908020, CCF-2118745, and CCF-2132049.

REFERENCES

[1] ACM SIGMOD Reproducibility. https://reproducibility.sigmod.org/.

[2] EuroSys Call for Artifacts. https://sysartifacts.github.io/eurosys2021/.

[3] OSDI Call for Artifacts. https://www.usenix.org/conference/osdi21/call-for-artifacts.

[4] PVLDB Reproducibility. http://vldb.org/pvldb/reproducibility/.

[5] SortBenchmark. http://sortbenchmark.org/.

[6] SOSP Call for Artifacts. https://sysartifacts.github.io/sosp2021/call.html.

[7] Alibaba-Cloud. TPC-C Result (Alibaba Cloud Elastic Compute Service Cluster). http://www.tpc.org/tpcc/results/tpcc_result_detail.asp?id=120051701.

[8] Mohammad Alomari, Michael J. Cahill, Alan D. Fekete, and Uwe Röhm. The Cost of Serializability on Platforms that Use Snapshot Isolation. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 576–585. IEEE Computer Society, 2008.

[9] Aria Source Code. https://github.com/luyi0619/aria.

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In Peter G. Harrison, Martin F. Arlitt, and Giuliano Casale, editors, *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64. ACM, 2012.

[11] Philip A Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[12] N. Budhijara, K. Marzullo, F. Schneider, and S. Toueg. The Primary-backup Approach. In S. Mullender, editor, *Distributed Systems*. Addison-Wesley, 2nd edition, 1993.

[13] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *CDCCA*, 1992.

[14] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[15] Ohio Supercomputer Center. Ohio Supercomputer Center, 1987.

[16] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. Tpc-e vs. tpc-c: Characterizing the new tpc-e benchmark via an i/o comparison study. *SIGMOD Rec.*, 39(3):5–10, February 2011.

[17] Cicada Source Code. https://github.com/efficient/cicada-exp-sigmod2017.

[18] Cloudlab. https://www.cloudlab.us/.

[19] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Re, and Matei Zaharia. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. https://cs.stanford.edu/ deepakn/assets/papers/dawnbench-sosp17.pdf, 2017.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[21] Artifacts of this Work. https://github.com/sam1016yu/DB-Exp-Sensitivity.

[22] Hardware Settings Used in this Work. https://github.com/sam1016yu/DB-Exp-Sensitivity/blob/main/hardware_config.md.

[23] Akon Dey, Alan D. Fekete, Raghunath Nambiar, and Uwe Röhm. YCSB+T: Benchmarking Web-scale Transactional Databases. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, pages 223–230. IEEE Computer Society, 2014.

[24] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254. ACM, 2013.

[25] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013.

[26] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. High-performance Conjugate-gradient Benchmark: A New Metric for Ranking High-performance Computing Systems. *The International Journal of High Performance Computing Applications*, 30(1):3–10, 2016.

[27] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In Ratul Mahajan and Ion Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[28] DrTM Source Code. https://github.com/SJTU-IPADS/drtm.

[29] Jose M Faleiro and Daniel J Abadi. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.*, 8(11), 2015.

[30] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.*, 10(5), 2017.

[31] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 371–384. USENIX Association, 2013.

[32] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1477–1488. IEEE, 2020.

[33] fio - flexible i/o tester. https://github.com/axboe/fio.

[34] Fursin, Grigori. Reproducing 150 Research Papers and Testing Them in the Real World: Challenges and Solutions. https://learning.acm.org/binaries/content/assets/leaning-center/webinar-slides/2021/grigorifursin_techtalk_slides.pdf, 2021.

[35] GAM Source Code. https://github.com/ooibc88/gam.

[36] Graph500 Committee. Graph500 benchmark. http://graph500.org.

[37] James N Gray. Notes on Data Base Operating Systems. In *Operating Systems*, pages 393–481. Springer, 1978.

[38] H2. The H2 Home Page. http://www.h2database.com.

[39] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.*, 10(5):553–564, January 2017.

[40] Stephen Hemminger et al. Network Emulation with NetEm. In *Linux conf au*, volume 5, page 2005. Citeseer, 2005.

[41] HERD Source Code. https://github.com/efficient/HERD.

[42] Janus Source Code. https://github.com/NYU-NEWS/janus.

[43] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 295–306. ACM, 2014.

[44] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: Fast Memory-optimized Database System for Heterogeneous Workloads. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1675–1687. ACM, 2016.

[45] Hideaki Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 691–706. ACM, 2015.

[46] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[47] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[48] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. High-performance Concurrency Control Mechanisms for Main-memory Databases. *Proc. VLDB Endow.*, 5(4):298–309, 2011.

[49] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In Ratul Mahajan and Ion Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 429–444. USENIX Association, 2014.

[50] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 21–35. ACM, 2017.

[51] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a Non-2pc Transaction Management in Distributed Database Systems. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1659–1674. ACM, 2016.

[52] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. On the Design and Scalability of Distributed Shared-data Databases. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 663–676. ACM, 2015.

[53] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.*, 13(12):2047–2060, July 2020.

[54] Yi Lu, Xiangyao Yu, and Samuel Madden. STAR: Scaling Transactions through Asymmetric Replication. *Proc. VLDB Endow.*, 12(11):1316–1329, July 2019.

[55] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 183–196. ACM, 2012.

[56] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. MLPerf Training Benchmark, 2020.

[57] Memcached. http://memcached.org.

[58] MICA Source Code. https://github.com/efficient/mica.

[59] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In Andrew Birrell and Emin Gün Sirer, editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 103–114. USENIX Association, 2013.

[60] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 517–532. USENIX Association, 2016.

[61] MySQL. http://www.mysql.com.

[62] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 385–398. USENIX Association, 2013.

[63] OLTPBench. https://github.com/oltpbenchmark/oltpbench.

[64] Oracle. TPC-C Result (SPARC SuperCluster with T3-4 Servers). http://www.tpc.org/tpcc/results/tpcc_result_detail.asp?id=110120201.

[65] John Ousterhout, Parag Agrawal, David Erickfson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

[66] Andrew Pavlo. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *SIGMOD 17*, page 3, 2017.

[67] Daniel Peng and Frank Dabek. Large-scale Incremental Processing using Distributed Transactions and Notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 251–264. USENIX Association, 2010.

[68] perf: Linux Profiling with Performance Counters. https://perf.wiki.kernel.org.

[69] Petitet, A. and Whaley, R. C. and Dongarra, Jack and Cleary, A. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. https://www.netlib.org/benchmark/hpl/.

[70] Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention Management with Deterministic Concurrency Control. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 180–194. ACM, 2021.

[71] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 446–459. IEEE Press, 2020.

[72] Kun Ren, Alexander Thomson, and Daniel J Abadi. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *Proc. VLDB Endow.*, 7(10):821–832, 2014.

[73] Lukas Rupprecht, James C. Davis, Constantine Arnold, Yaniv Gur, and Deepavali Bhagwat. Improving Reproducibility of Data Science Pipelines through Transparent Provenance Capture. *Proc. VLDB Endow.*, 13(12):3354–3368, August 2020.

[74] Silo Source Code. https://github.com/stephentu/silo.

[75] Standard Performance Evaluation Corporation. Standard performance evaluation corporation. https://www.spec.org/.

[76] Star Source Code. https://github.com/luyi0619/star.

[77] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[78] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin Code Repository. https://github.com/yaledb/calvin.

[79] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12. ACM, 2012.

[80] Pınar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. From A to E: Analyzing TPC's OLTP Benchmarks: The Obsolete, the Ubiquitous, the Unexplored. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 17–28. ACM, 2013.

[81] Transaction Processing Performance Council. The TPC-E home page. http://www.tpc.org/tpce/.

[82] Transaction Processing Performance Council. TPC Benchmark C Standard Specification Revision 5.11. http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[83] Transaction Processing Performance Council. TPC-C - All Results - Sorted by Performance. http://tpc.org/tpcc/results/tpcc_results5.asp.

[84] Transaction Processing Performance Council. The TPC-C home page. http://www.tpc.org/tpcc/.

[85] Transaction Processing Performance Council. Tpc-homepage. http://www.tpc.org.

[86] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-memory Databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 18–32. ACM, 2013.

[87] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is Big Data Performance Reproducible in Modern Cloud Networks? . In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 513–527. USENIX Association, 2020.

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

[89] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 87–104. ACM, 2015.

[90] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.*, 10(7):781–792, March 2017.

[91] Juncheng Yang, Yao Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-memory Cache Clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 191–208. USENIX Association, 2020.

[92] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. Tictoc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1629–1642. ACM, 2016.

[93] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Tapir Code Repository. https://github.com/UWSysLab/tapir.

[94] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 263–278. ACM, 2015.