Concurrency Control Based on Transaction Clustering

Xuebin Su
Harbin Institute of Technology
xvebinsu@gmail.com

Hongzhi Wang
Harbin Institute of Technology
Peng Cheng Laboratory, Shenzhen, China
wangzh@hit.edu.cn

Yan Zhang
Harbin Institute of Technology
zhangy@hit.edu.cn

***Abstract*—Concurrency control is a mechanism that database systems provide to allow multiple transactions to be executed at the same time while enforcing isolation. The concurrency control algorithm is key to performance of a database system. However, different concurrency control algorithms have different strengths and weaknesses, making each of them fits only for some types of workloads, while performs unsatisfactorily for others. As a result, the user will have to make assumptions about the workloads before choosing the concurrency control algorithm to achieve the best performance. To overcome this limitation, we propose a scheme, called transaction clustering, to decide the best isolation mechanism for any given pair of transactions automatically. Based on transaction clustering, we further develop the Clustering-based Concurrency Control algorithm, or C3 for short, which combines the pessimistic and the optimistic concurrency control algorithms to get the best of both worlds while mitigating their performance bottlenecks at the same time. Both theoretical and experimental studies show that, for high-conflict workloads, the performance of the C3 algorithm can be significantly better than the performance of both the pessimistic and the optimistic algorithms that C3 is based on.**

***Index Terms*—database, transaction, concurrency control, clustering**

I. Introduction

Concurrency control is a mechanism that database systems provide to allow multiple transactions to be executed at the same time while preserving data consistency. Due to its importance, it has been studied for several decades, and many concurrency control algorithms have been proposed. Based on how transaction conflicts are resolved, many concurrency control algorithms can be divided into two categories: pessimistic algorithms and optimistic algorithms. These two categories of algorithms are very different in principle. Therefore, they have different strengths and weaknesses, making each of them fits only for some types of workloads, while performs unsatisfactorily for others. To understand this, we compare the two categories of algorithms in terms of the cost of lock contention and the abort rates of transactions below.

- **Cost of lock contention:** Pessimistic algorithms require transactions hold locks during the whole life cycle of the transaction while optimistic algorithms acquire locks only when transactions commit. As a result, the time of holding locks for pessimistic algorithms would typically be much longer than that for optimistic algorithms and therefore lock contention would be much more severe. Research shows that lock contention is a main performance bottleneck for pessimistic algorithms [13]. For highly-contended workloads, transactions using pessimistic algorithms do almost nothing but waiting for locks.
- **Abort rate:** For pessimistic algorithms, the main cause of transaction abort is deadlocking, which happens rarely. In contrast, optimistic algorithms abort transactions much more frequently due to validation failure. Research shows that for highly-contended workloads, almost all transactions are aborted by the OCC algorithm [11], and the high abort rate is a main bottleneck for the OCC algorithm [13].

In summary, the strength and weakness of the two categories of concurrency control algorithms are listed in Table I.

Concurrency control algorithms	Pessimistic	Optimistic
Cost of lock contention	High	Low
Abort rate	Low	High

TABLE I: Strength and Weakness of the Two Approaches

Goals and challenges: From the discussions above, we observe that the two approaches are highly complementary to each other. The weakness of one algorithm is exactly the strength of the other. Therefore, we hope to find a way to combine these two approaches into a novel concurrency control algorithm to inherit their strengths while overcoming their weaknesses. More specifically, we want the novel algorithm to achieve the following three non-trivial goals:

- **Mitigating the bottlenecks of both approaches:** The primary goal of this idea is to avoid performance bottlenecks of both the pessimistic and the optimistic algorithms. However, due to the large discrepancy in principle between the two approaches, their performance bottlenecks are very different. Therefore, it is challenging to design one mechanism to mitigate the bottlenecks of the two approaches.
- **Enforcing isolation:** It is vital for concurrency control algorithms to ensure isolation of transactions. However, when trying to combine multiple concurrency control algorithms, simply allowing multiple transactions running concurrently under different concurrency control algorithms can violate the isolation property and cause data inconsistency [12]. Therefore, it requires non-trivial effort to design a mechanism to enforce isolation while still allow each transaction to freely choose the best isolation mechanism against other transactions.
- **Minimizing performance penalty:** Combining multiple concurrency control algorithms comes with performance penalty compared to using a single pessimistic or optimistic algorithm. The penalty comes mainly from two sources: (1) choosing the best isolation mechanism for transactions or (2) enforcing isolation by imposing more control on each transaction. For (1), an intuitive algorithm is to consider all other transactions for a given transaction, and choose the best isolation mechanism for each pair, like in [8], which would take $\mathrm{O}(n)$ time where n is the

number of transactions in the system, which makes the system hard to scale. For (2), a straightforward solution is to add another layer of concurrency control algorithm to coordinate transactions running under independent different concurrency control algorithms [12], which might introduce unnecessary extra synchronization overhead.

Contributions: To achieve our goals, we propose the Clustering-based Concurrency Control (C3) algorithm that combines the pessimistic and the optimistic algorithms and chooses the best isolation mechanism for each transaction against other transactions. This is done fully automatically when each transaction begins with low cost. More specifically, in response to the three main challenges above, we make the following three contributions:

- To mitigate bottlenecks of both the pessimistic and the optimistic approaches, we propose a novel similarity metric for transactions to cluster them and a novel criterion to determine how any pair of transactions should isolate with each other based on whether they belong to the same cluster. By clustering and partitioning the set of transactions, contention among transactions can be significantly reduced. As a result, the C3 algorithm performs much better than traditional pessimistic and optimistic algorithms under high-contention workloads.
- To enforce isolation, we propose to force each transaction to perform validation checking whether there is any conflict between it and other transactions in the C3 algorithm. Even though this step would introduce extra overhead, it is necessary to ensure data consistency. Moreover, since validation can be done in parallel, it would unlikely become a scalability bottleneck.
- To reduce performance penalty, we first propose to use Locality-Sensitive Hashing (LSH) to cluster transactions and determine the best isolation mechanism for each transaction such that the average running time for processing one transaction is $\mathrm{O}(1)$, meaning that it is independent from the total number of transactions being processed in the system. This makes C3 an efficient and practical concurrency control algorithm.

II. The C3 Algorithm

In this section, we present the Clustering-based Concurrency Control (C3) algorithm. We first introduce the basic idea of C3, then we describe how it works briefly and analyze its correctness and effectiveness. Finally, we discuss the design assumptions and trade-offs behind C3.

A. The Basic Idea of C3

Based on the observation in Section I that the pessimistic concurrency control algorithm and the optimistic concurrency control algorithm are highly complementary to each other, the most fundamental idea of C3 is to combine the two algorithms to exert their strengths and avoid their weaknesses. In C3, this is done by choosing the best isolation mechanism for each transaction against other transactions.

So, How to determine which isolation mechanism, locking or conflict validation, is the best? Generally, we have the following common wisdom on choosing the concurrency control algorithm:

Proposition 1 (Common Wisdom)**.** *When the conflict among transactions is severe in a system, pessimistic concurrency control algorithms should be adopted to avoid aborting transactions frequently, whereas when the conflict is mild, optimistic concurrency control algorithms should be adopted to minimize the cost of locking.*

This common wisdom is described in many widely recognized textbooks [4] and also guides recent research [10], [11]. From the common wisdom, we conclude that the conflict rate of transactions is a key criterion to choose concurrency control algorithms.

Based on this observation, C3 first considers all the running transactions as data and performs data clustering [6] to divide them into different groups such that transactions that conflict with each other severely are placed in the same *cluster* and transactions that seldom conflict with each other are divided in to different clusters. After clustering, transactions in the same cluster have a high conflict rate with each other such that they should be isolated using the pessimistic approach, while transactions in different clusters have a low conflict rate with each other such that they should be isolated using the optimistic approach. This idea is visualized as Figure 1 where PCC and OCC denote the Pessimistic and the Optimistic Concurrency Control algorithm respectively, each filled circle represents a transaction, and each dashed circle represents a cluster.

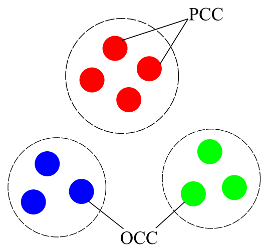

Fig. 1: A Schematic Diagram Showing the Idea of C3

B. How C3 works

Specifically, following the basic idea introduced above, the C3 algorithm takes the following three big steps in order when processing one transaction:

1) **Transaction clustering:** After the transaction begins, it finds the cluster that the current transaction belongs to.
2) **Cluster locking:** To isolate the current transaction against other transactions in the same cluster pessimistically, the transaction first requires the corresponding lock before accessing a data record. Whether it can obtain the lock depends on whether there exists any other transaction in the same cluster that is holding the lock.
3) **Conflict validation:** To isolate the current transaction against other transactions in different clusters optimistically, after data access and before the transaction commits, it validates whether there exists any read-write conflict between itself and any other transaction.

To show the whole picture of the algorithm, consider a minimal example that a transaction t accessing only one data record x. The whole workflow of processing the transaction t is shown Figure 2. Note that steps that are not directly related to concurrency control are omitted such as logging and writing to the database.

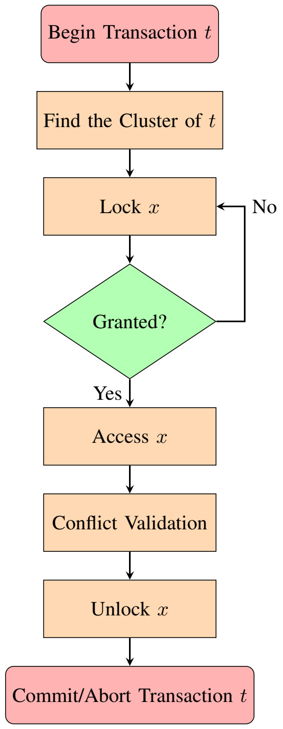

Fig. 2: Workflow of Processing a Minimal C3 Transaction

Next, we will introduce the three big steps, i.e., transaction clustering, cluster locking, and conflict validation, in more details.

Transaction Clustering: As introduced above, the C3 algorithm first clusters transactions based on how likely that two transactions would conflict with each other, which can be estimated by the Jaccard similarity [6] between the *working sets* of the two transactions where the working set of a transaction is defined as the set of all data records accessed by the transaction [13]. This is because if two transactions access more data records in common, the probability that the two transactions conflict will be higher, and so will the Jaccard similarity between their working sets.

Therefore, we propose to use the Jaccard similarity of two transactions' working sets as the similarity metric to cluster transactions in order to achieve the goal that transactions conflict severely are in the same cluster.

After defining the similarity metric, we consider the algorithm for clustering transactions. The clustering algorithm is required to be incremental rather than batched, i.e., for each incoming transaction, it needs to efficiently find the cluster that the transaction belongs to. One naive implementation is to check the incoming transaction against all other running transactions to find those whose similarity is above a threshold. This takes $\Theta(n)$ time where n is the total number of running transactions, which makes the database system hard to scale to handle many transactions.

To make the C3 algorithm more scalable, we use Locality-Sensitive Hashing (LSH) [7] to cluster transactions. Specifically, since the Jaccard similarity is used as the similarity metric, we use the MinHash algorithm [1], [7] to hash the working set of a transaction to a vector L hash values. Each hash value is generated by a randomly-chosen and mutually-independent hash function. It is proven that the probability that a MinHash function generates equal hash values for two sets is equal to their Jaccard similarity [7]. According to this property, when two sets are similar, the Jaccard similarity is high, and the probability that their MinHash values are equal is also high.

Therefore, we use MinHash vectors for identifying transaction clusters such that transactions in the same cluster has at least one MinHash vector in common generated from their working sets. The size of the MinHash vector L is chosen to be large enough such that the Jaccard similarity of two transactions' working sets is large and the conflict probability of the two transactions is high. To increase the recall, we use K MinHash vectors for a transaction and claim that two transactions belong to the same cluster as long as they have at least 1 of K vectors in common. This is the basic idea of the transaction clustering algorithm, which is formally stated as Algorithm 1.

Algorithm 1: The Transaction Clustering Algorithm

```
Input: Incoming transaction $t$ with $K$ MinHash vectors
       $(k_1, k_2, \ldots, k_K)$, and a MinHash table $M$ storing
       the mappings from any MinHash vector $k$ to a set
       of transactions such that each transaction has a
       MinHash vector equal to $k$
Output: The set $C$ of transactions that in the same
        cluster as $t$
1 $C \leftarrow \emptyset$
2 foreach $k$ in $(k_1, k_2, \ldots, k_K)$ do
3     $C_k \leftarrow$ The set of transactions that $k$ maps to in $M$
4     $C \leftarrow C \cup C_k$
5 return $C$
```

The MinHash table can be implemented using any associative container. If it is implemented using a hash table, the average running time of finding the cluster for an incoming transaction is $\Theta(KL)$ where L is the length of a MinHash vector and K is the number of MinHash vectors for a transaction. KL is the number of total MinHash values computed for clustering for each transaction. Since the running time is constant to the number of running transactions, using Algorithm 1 in C3 makes it more scalable compared to using the naive approach.

Cluster Locking: Cluster locking is a novel locking mechanism used to isolate the current transaction against other transactions in the same cluster. A transaction is required to acquire the *cluster lock* before accessing any data record and whether it succeeds depends on how transactions are clustered.

To implement cluster locks, we need to first design the lock table. Similar to 2PL, which has the following two-level structure:

- Similar to the lock table of 2PL, the first level maps any data record x to a table containing all the lock-holding transactions, representing the lock of x.
- Different from the lock table of 2PL, the second level is a MinHash table introduced above containing information on how transactions are clustered, rather than being an array of lock-holding transactions.

By searching the lock table, a transaction can be informed directly whether there is any transaction that belong to the same cluster and that is holding the lock. If there isn't, the lock can be granted to the requesting transaction. Otherwise, the lock is granted if and only if the requested locking mode is compatible with the current locking mode. The lock compatibility matrix of the C3 algorithm is exactly the same as the one of the 2PL algorithm.

With the lock table, we now introduce the algorithm of locking a data record, which is formally stated as Algorithm 2

Algorithm 2: The Lock Acquisition Algorithm

```
Input: The lock table $T$, the requesting transaction $t$ with
       locking mode $m$ and data record $x$
Output: TRUE if the lock is granted, otherwise FALSE
1  if $x$ is not locked then
2      Add transaction $t$ with its MinHash vectors to $M$
3      return TRUE
4  Retrieve the MinHash table $M$ and the current locking
     mode $m'$ of $x$ in $T$
5  Find the cluster $C$ of $t$ using Algorithm 1
6  if $C \setminus \{t\} \neq \emptyset$ and $m$ is not compatible with $m'$ then
7      return FALSE
8  else
9      Add transaction $t$ with its MinHash vectors to $M$
10     return TRUE
```

The lock acquisition algorithm of C3 is similar to the one of 2PL, except that in 2PL, all lock-holding transactions would conflict with the requesting transaction, while in C3, only transactions that in the same cluster would conflict with the requesting transaction while other transactions are ignored.

The unlocking algorithm of C3 is simple, Similar to the one of 2PL, it simply deletes the requesting transaction with its MinHash vectors from the corresponding MinHash table.

In conclusion, with cluster locking, we realize the goal that transactions in the same cluster are isolated using a pessimistic algorithm. Each time before a transaction access a data record, it contends the lock with other concurrent transactions in the same cluster and would block until the lock is granted.

Conflict Validation: Conflict validation is used to enforce isolation in C3. Even though transactions in the same cluster are isolated with locks, access by transactions from different clusters are not under control, which might cause data inconsistency.

To enforce isolation, the C3 algorithm requires any transaction to undergo the same conflict validation phase as in OCC when it commits, testing whether there is any read-write conflict against other transactions. This is shown in Figure 2 as the "Conflict Validation" step. If there is conflict, the transaction must be aborted.

By validation, no matter how transactions are clustered and no matter whether transactions acquire locks successfully, isolation and data consistency will always be guaranteed.

C. Analysis of C3

In this section, we show that the C3 algorithm is both correct and effective theoretically.

Correctness: The correctness property of the C3 algorithm is formally stated as Theorem 2.

Theorem 2 (Correctness of C3)**.** *No matter how transactions are clustered, the C3 algorithm is correct as long as the validation process identifies conflicts correctly.*

We omit the proof due to the limit of space. To see why it holds, note that as long as there exist any conflict that might cause data inconsistency, the current transaction would fail to pass the validation phase and would be aborted and the database would not be modified.

By Theorem 2, it is trivial to ensure the correctness of C3 since the only thing to do is to borrow the validation process directly from any correct version of OCC without modification.

Effectiveness: To show the effectiveness of the C3 algorithm, we argue that it can mitigate the bottlenecks of both the pessimistic and the optimistic approaches by reducing both lock contention and the abort rate, which follows the theorem below immediately.

Theorem 3 (Effectiveness of C3)**.** *For any transaction* t *, let* C_t *be the set of transactions that belongs to the same cluster as* t *and* $C_t^{\complement}$ *be the complement of* C_t *containing transactions in clusters different from* t*'s. Then it is not possible for* t *to have any lock contention with any transaction in* $C_t^{\complement}$ *or to cause any other transaction in* C_t *to abort.*

We omit the proof again due to the limit of space. To understand why Theorem 3 is correct, note that t would not cause any transaction in the same cluster to be aborted since as long as they conflict with each other, one of them must wait for the cluster lock before accessing the data, and that t would not have lock contention with transactions in different clusters since those transactions are ignored when t acquires the cluster lock by Algorithm 2.

By Theorem 3, we conclude that using the C3 algorithm can reduce both lock contention rate and the abort rate compared to using pure pessimistic and optimistic concurrency control algorithms, thus mitigating the performance bottlenecks of the two algorithms at the same time.

D. Design Assumptions

Note that C3 algorithm relies on three key assumptions to take maximum effect.

1) The working set of a transaction is known prior to executing the transaction. This is required for transaction clustering.
2) For the pessimistic algorithms, we assume that the main performance bottleneck is waiting due to lock contention, rather than the locking and the unlocking processes.
3) For the optimistic algorithms, we assume that the main performance bottleneck is the large amount of wasted work due to high abort rate, rather than the cost of the validation process.

Assumption 1) can be satisfied by using stored procedures [9], or making prediction upon seeing the first data record that the incoming transaction wants to access based on the Write-Ahead Log using a sequence model such as Markov models or RNNs [5].

Assumption 2) and 3) are valid for high-conflict workloads. Under such workloads PCC algorithms and OCC algorithms waste most of the time in either waiting for locks or aborting transactions [13].

III. Experimental Evaluation

In this section, we conduct experiments to evaluate the performance of the C3 algorithm. We first introduce the setup, i.e., how we implement the C3 algorithm and the experiment environment briefly. Then we show the effectiveness of the C3 algorithm under high-conflict workloads. Finally, we evaluate impact of performance penalty for the C3 algorithm under low-conflict workloads.

A. Setup

Implementing C3: We implement the C3 algorithm on RocksDB, a high-performance key-value database that is widely adopted by many companies. RocksDB supports both pessimistic and optimistic concurrency control using 2PL with deadlock detection and the Multi-Version OCC (MVOCC) algorithm respectively [3]. Our implementation of C3 is based on the 2PL algorithm provided by RocksDB to reuse the lock management code as much as possible but with modifications on the lock table to support transaction clustering and add the conflict validation process of MVOCC before commit. We set the lock timeout to be smallest possible, which is 1ms, such that we can use the timeout ratio of transactions as the indicator of the severity of lock contention. Likewise, the validation failure ratio of transactions is used as the indicator of the severity of transaction abortion.

Candidates: We compare the C3 algorithm against the two pessimistic and optimistic algorithms it is based on, i.e., 2PL and (multi-version) OCC. This is to demonstrate the effectiveness of the idea of transaction clustering by showing that the C3 algorithm can mitigate the bottleneck of the two algorithms.

Workloads: The workloads we use in the experiments is YCSB [2]. We use the skewness parameter α of the Zipfian distribution in YCSB to control the conflict severity. To evaluate performance in terms of conflict severity, we use both the low-conflict workloads with $\alpha = 0.2$ and the high conflict workloads with $\alpha = 0.8$, In low-conflict workloads, data access is less concentrated than in high-conflict workloads. In terms of operation types, we generate both the read-intensive workloads with read-write ratio $80/20$, and the write-intensive workloads with read-write ratio $20/80$.

Environment: All the experiments are perform on a 16-core and 32-thread machine. Each transaction is executed in a native operating system thread. Each thread is protected from race condition using operating system mutexes and condition variables.

B. Experiment Results

In the experiments, we use the throughput as the main performance indicator and we also examine the timeout ratio and the validation failure ratio to see the severity of lock contention and transaction abortion.

High-Conflict Workloads: In the high-skewness write-intensive workload, the conflict rate among transactions is high since write operations conflict with both read and write operations. We show the results under the high-conflict workload as Figure 3. From the figure, we observe that when the number of thread is large, the throughput of the C3 algorithm is significantly larger that those of both the pessimistic and the optimistic algorithms. That means the C3 algorithm outperforms both the pessimistic and the optimistic algorithms. This is because in the high-conflict workload, the 2PL algorithm spends most of the time waiting for locks and the OCC algorithm spends most of the time aborting transactions, which can be seen form the fact that the timeout ratio of 2PL and the validation failure ratio of OCC are both high (≥ 0.6 when the number of running transactions is 64), which are the main bottlenecks of the two algorithms. On the other hand, the timeout ratio of C3 is much lower than that of 2PL and the validation failure ratio of C3 is much lower than that of OCC. This means that C3 can effectively mitigate the bottlenecks of the two algorithms at the same time and thus achieving higher performance.

Low-Conflict Workloads: As shown in Figure 4, Figure 5, and Figure 6, for the high-skewness read-intensive workload and the low-skewness workloads, the throughput of the C3 algorithms is close to those of the other two algorithms. This is because in those cases, the conflict among transactions is much less severe compared to that in the high-skewness write-intensive workload such that the performance gain due to mitigating lock contention and frequent validation failure is insignificant. also show that the performance penalty of the C3 algorithm is small. This is because under low-conflict workloads, the performance penalty introduced by transaction clustering would become a major factor that lower the performance of the C3 algorithm.

In summary, we show in this section by experiments that (1) under high-conflict workloads, the C3 algorithm outperforms significantly both the pessimistic and the optimistic algorithms by mitigating lock contention and validation failure; and (2) under low-conflict workloads, the effect of performance penalty of C3 compared to either 2PL or OCC is not significant and the performance of all the three algorithms are similar to each other. With the facts above, we claim that we have achieved the first and the third goals stated in Section I.

IV. Conclusions and Future Work

In this paper, the we show that the Clustering-based Concurrency Control (C3) algorithm mitigates the performance bottlenecks of both the pessimistic and the optimistic algorithms such that under high-conflict workloads, the C3 algorithm outperforms the two algorithms significantly. Meanwhile, the extra overhead introduced in C3 compared to either the pessimistic algorithm or the optimistic algorithm is low such that for low-conflict workloads, the performance lose of C3 compared to the two algorithms is very limited.

For future work, one interesting and important problem is how to extend the C3 algorithm to support range queries, which is an essential part of SQL. In this paper, we only focus on only the read and the write operations since they are the most fundamental operations and it is easy to estimate the similarity of transactions' working sets. For range queries, elements in a transaction's working set is not given directly, but described by predicates. Therefore, the Jaccard similarity cannot be directly applied to compute the similarity. Instead, it is required to estimate the similarity between two predicates, which might involve the techniques of cardinality estimation [4].

V. Acknowledgements

This paper was partially supported by NSFC grant U1866602, 61602129, 61772157, CCF-Huawei Database System Innovation Research Plan DBIR2019005B. Hongzhi Wang and Xuebin Su contributed to the work equally and should be

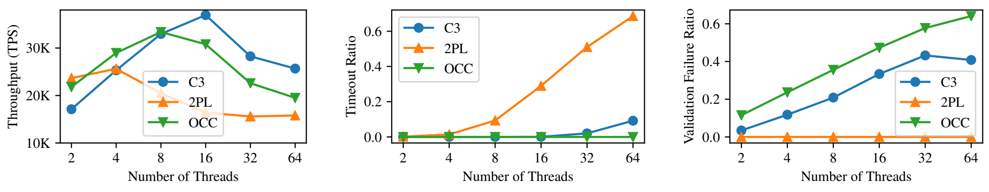
Fig. 3: Performance Evaluation under High-Skewness Write-Intensive Workload

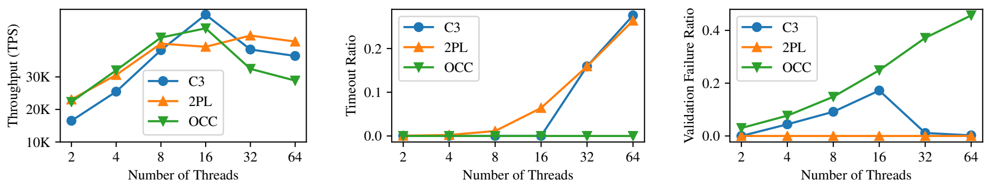
Fig. 4: Performance Evaluation under High-Skewness Read-Intensive Workload

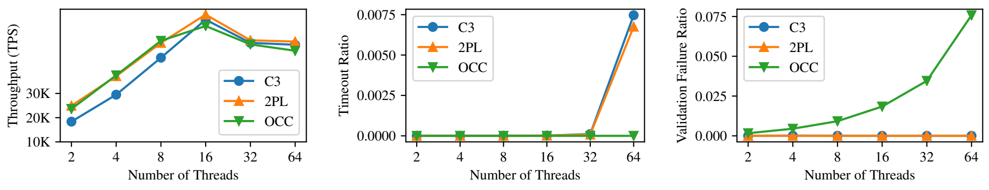
Fig. 5: Performance Evaluation under Low-Skewness Write-Intensive Workload

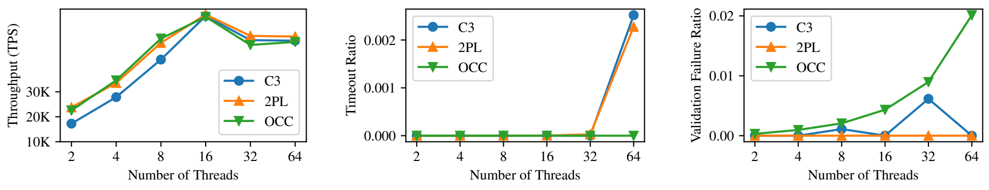
Fig. 6: Performance Evaluation under Low-Skewness Read-Intensive Workload

regarded as co-first authors. Hongzhi Wang is the corresponding author of this work.

References

[1] A. Z. Broder. On the resemblance and containment of documents. In B. Carpentieri, A. D. Santis, U. Vaccaro, and J. A. Storer, editors, *Compression and Complexity of SEQUENCES 1997, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997, Proceedings*, pages 21–29. IEEE, 1997.

[2] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In J. M. Hellerstein, S. Chaudhuri, and M. 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.

[3] Facebook. Transactions · facebook/rocksdb wiki, 2020. https://github.com/facebook/rocksdb/wiki/Transactions.

[4] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[5] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[6] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques, 3rd edition*. Morgan Kaufmann, 2011.

[7] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[8] Y. Sheng, A. Tomasic, T. Zhang, and A. Pavlo. Scheduling OLTP transactions via learned abort prediction. In R. Bordawekar and O. Shmueli, editors, *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, pages 1:1–1:8. ACM, 2019.

[9] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1150–1160. ACM, 2007.

[10] D. Tang, H. Jiang, and A. J. Elmore. 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*. www.cidrdb.org, 2017.

[11] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endow.*, 10(2):49–60, 2016.

[12] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance ACID via modular concurrency control. In E. L. Miller and S. Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 279–294. ACM, 2015.

[13] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, 2014.