Adaptive Transaction Scheduling for Highly Contended Workloads

Demo ID: 699

DASFAA 2019

Abstract. Future servers will be equipped with thousands of CPU cores. Traditional transaction scheduling mechanism—which is a key component in database systems—slows down the performance of concurrency control greatly in such environments for highly contended workload. Obviously, to address this issue, there are two effective methods: (1) avoiding concurrent transactions that access the same high-contention tuple at the same time; (2) accelerating the execution of these high-contention transactions. In this demonstration, we present a new transaction scheduling mechanism, which aims to achieve the above goals. An adaptive group of *first-class* queues is introduced, each queue is allocated to a specified worker thread and takes charge of transactions accessing specified high-contention tuples. We implement a system prototype and demonstrate that our transaction scheduling mechanism can effectively reduce the abort ratio of high-contention transactions, which improves the system throughput dramatically.

1 Introduction

Nowadays, with the rapid growth of data traffic in many applications, database servers need to face numerous online transactions at the same time. Fortunately, the servers equipped with thousands of CPU cores have the ability to handle a huge number of requests. However, existing databases cannot scale up to thousands of cores, especially when the workload is highly contended [3]. This is because that traditional transaction scheduling mechanism randomly chooses a worker thread to execute a new transaction. In the highly contended workload, the probability that two concurrent transactions in different worker threads access a same tuple is very high. This leads to a result that one of both conflicted transactions may be aborted or blocked, which has a negative impact on system performance. To address this issue, there are two effective methods as follows:

1. Avoid concurrent transactions that access the same high-contention tuple.
2. Accelerate the execution of the high-conflict transactions.

Zhang et al. provided an intelligent scheduling to put a new arrived transaction into the queue with the highest likelihood of conflict [4], which aims to achieve the first goal. However, this method does not accelerate the execution of any transaction. MySQL enterprise edition utilizes the primary queue to accelerate the execution of a likely-conflicted transaction [1]. But it is not designed

for the first point. Although H-Store [2] is regarded as a solution to achieve both goals, its distributed feature leads to its struggle for load-balance.

In this work, we propose a new transaction scheduling mechanism, which aims to achieve the above two goals at the same time. First, we utilize the *first-class* queues, which are responsible for handling transactions that access the specified high-contention tuples. As a result, we can prevent the concurrent transactions from accessing the same high-contention tuple at a time point. Second, we can increment the number of *first-class* queues. Since each *first-class* queue is allocated to a specified worker thread, the computing resources can be added for the high-contention transactions. Consequently, the execution time of a high-contention transaction is decreased effectively, which can reduce the probability of conflict. In the next sections, we will introduce our system architecture and the key techniques and provide a demo for our system.

2 System Architecture and Key Techniques

The overall architecture we designed is shown in Figure 1, which consists of two main components: statistics module and transaction scheduling module. The statistics module is responsible for collecting the historical information of transactional operations, which is used to acquire: (1) which tuple is high-contention; (2) which two high-contention tuples in a transaction unit are accessed frequently. We provides the details of this module in Section 2.1.

In the scheduling module, there is a transaction scheduler, which is responsible for forwarding a transaction to a queue. In our system, these queues are classified into two categories: one *public* queue and a variable number of *first-class* queues. We will introduce the adaptive strategy of varying the number of *first-class* queues in Section 2.2. It should be noted that each *first-class* queue is assigned to a specified worker thread, and it is in charge of a set of high-contention tuples. The intersection of tuple sets of any two different *first-class* queues is empty. We will describe how to distribute the high-contention tuples to different *fist-class* queues in Section 2.3. If a new transaction wants to access a high-contention tuple, it is forwarded to the *first-class* queue which takes charge of this tuple. Otherwise, the transaction—which does not contain any high-contention tuples—would be forwarded to the *public* queue. If a worker thread is not assigned to a first-class queue, it will get and execute a transaction from the *public* queue. Otherwise, it can only handle the transactions in its own *first-class* queue.

As shown in the left picture of Figure 1, there are two new transactions T_1 and T_2. Since T_2 wants to access the high-contention tuples c and e, it is forwarded to the *first-class* queue f_1, which is in charge of these tuples. On the other hand, the transaction T_1—which does not access any high-contention tuples—is forwarded to the *public* queue.

2.1 Statistics information

In our system, each tuple has a implicit field *temperature* (abbr. *temp*), which is used to track the access number of this tuple in the past period of time. If a tuple is accessed by a transaction, we increment its *temp*. On the other hand,

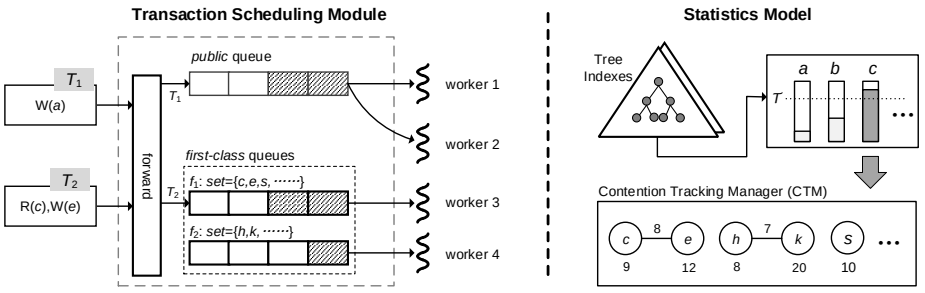


Fig. 1. Illustration of our system architecture.

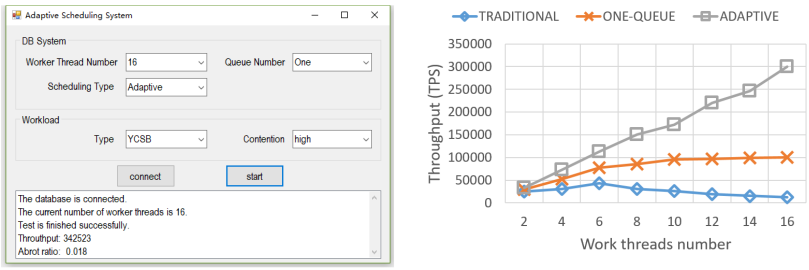
if the tuple is not accessed by any transaction over a specified period of time, we decrement its *temp*. A tuple, whose *temp* exceeds a pre-configured threshold $\mathcal{T}$, is added to the contention tracking manager (CTM). Note that CTM also tracks the number of the *correlation* of any two tuples. If two tuples in CTM are accessed by a transaction, we increment the *correlation* of these two tuples. As illustrated in the right picture of Figure 1, the *correlation* of tuples c and e is 8, which illustrates that c and e are accessed simultaneously by eight transactions. When a transaction accesses a tuple in CTM and finds its *temperature* is less than $\mathcal{T}$, it removes the tuple from CTM.

2.2 Adaptive adjustment of the number of first-class queues

The number of *first-class* queues is determined adaptively by two aspects: (1) the total number of transactions accessing the high-contention tuples; (2) the degree of emergency for handling the high-contention transactions. Therefore, we introduce an equation to acquire the *first-class* queues' number N_f: $N_f = (\lambda \cdot n_f \cdot (N - N_f))/n_p$, where n_p and n_f are the number of transactions entering the *public* queue and the *first-class* queues respectively, N is the total number of the worker threads, and λ is the emergency factor that is decided by the abort ratio of transactions. Note that with the increase of abort ratio, λ, which is always ≥ 1, becomes larger in order to allocate more resources for the high-contention transactions. In our system, once the workload is changed, the number of *first-class* queues are varied to adapt to the new data access model.

2.3 Element allocation for the first-class queues

Recall that each *first-class* queue has its own set of high-contention tuples. In other words, a transaction only enters into the *first-class* queue whose set contains the high-contention tuples accessed by the transaction, and it is only executed by the specified worker thread. The number of *first-class* queues has been acquired in the above section. Therefore, we need a reasonable approach to allocate the high contention tuples to the different *first-class*, which leads to the result that (1) each worker thread executes transactions in the load-balanced situation; (2) a transaction enters into the *first-class* queue whose set contains as many of high-contention tuples accessed by the transaction. Consequently, we first order the tuples in CTM according to their *temps*, and then allocate the tuple and its correlative tuples to a *first-class* queue in a round-robin manner.



(a) User interface. (b) Throughput.

Fig. 2. Experimental demonstration. The right picture shows the throughput of different transaction scheduling methods.

3 Evaluation

We have implemented a system prototype to demonstrate the effectiveness of our proposed method. The system prototype provides a user interface, which is illustrated in Figure 2(a). We can choose the worker number, scheduling type (e.g., ADAPTIVE is our proposed method), the workload type (e.g., YCSB, TPC-C) and the contention of the corresponding workload (e.g., high, low). Then we can push the START button to run the workload in the pre-configured database system. When the test is finished, the results are printed in the console.

ALL experiments are running in a server with 2-socket Intel Xeon E5606 @2.13GHz. Experimental results are shown in Figures 2(b). Note that TRADITIONAL is used to denote the database that does not adopt any optimization method for the highly contended workload, and ONE-QUEUE is used to represent the database that adopts a single queue to handle high-contention transactions. Experimental results show that our method ADAPTIVE outperforms the other transaction scheduling schemes, which conforms to our expectation. Consequently, our method is suitable for the highly contended workload in the modern hardware equipped with multiple CPU cores.

4 Conclusion

Traditional transactional scheduling mechanism does not perform well for the highly contended workload in the modern server equipped with multi-sockets. In this demonstration, we present a new adaptive mechanism for transaction scheduling and show that this method has a good performance.

References

1. MySQL Enterprise Thread Pool. https://dev.mysql.com/doc/refman/8.0/en/thread-pool.html.
2. R. Kallman, H. Kimura, J. Natkins, et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.
3. T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, 2016.
4. T. Zhang, A. Tomasic, Y. Sheng, and A. Pavlo. Performance of OLTP via intelligent scheduling. In *ICDE*, pages 1288–1291, 2018.