POSIT: Proactive Operation Scheduling for Interactive Transactions in Main Memory Database Systems

Donghui Wang, Peng Cai, Weining Qian, Aoying Zhou

School of Data Science & Engineering
East China Normal University
donghuiwang@stu.ecnu.edu.cn, {pcai,wnqian,ayzhoug}@dase.ecnu.edu.cn

ABSTRACT

To improve the performance for high contention workloads, modern main memory database systems (e.g. Calvin or H-store) usually assume that all operations in a transaction are known in advance, and thus generate a conflict-free schedule. However due to the scalability and flexibility of the Multi-Tier architecture, in many OLTP applications, instead of using stored procedure, their business logics are still implemented by using interactive JDBC/ODBC-like API. Operation scheduling in interactive transactions mostly uses the FIFO strategy, where the produced execution order is not pre-determined by transaction conflicts. In this paper, we present POSIT, a proactive operation scheduler to generate an efficient execution order for interactive transactions under high-contention workloads. POSIT uses multiple queues with different priorities to buffer transaction operations. Before an update operation is enqueued, the scheduler decides whether it needs to be prioritized or postponed, and pushes it into queues with proper priority. We demonstrate that POSIT introduces no overhead for low-contention workloads and early results are promising for high-contention workloads.

1. INTRODUCTION

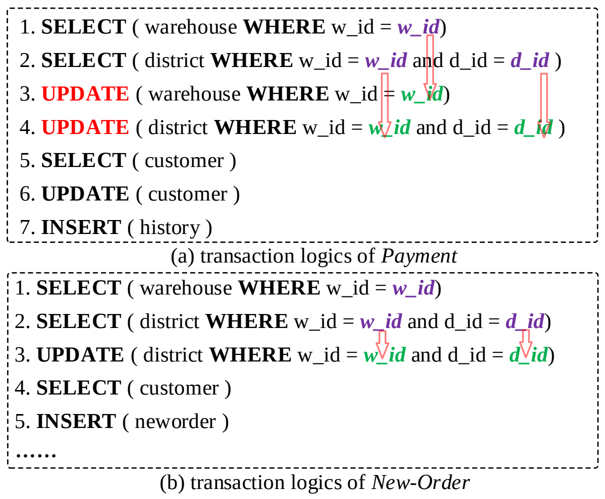
Main memory database systems hold the entire working dataset into memory. As a result, the transaction processing performance is not dominated by the slow disk I/O. To take full advantage of multi-core processors, modern main memory database systems (e.g. Calvin or H-store) usually assume transactions are executed in a stored-procedure manner. Based on this assumption, these systems could generate a conflict-free schedule by figuring out the full read/write sets ahead of time for conflicting transactions. Although transactions executed by stored procedures can achieve a great performance, transaction logics in many real-world OLTP applications are still implemented by using JDBC/ODBC-like API because of the scalability and flexibility of the Multi-Tier architecture. In this scenario, clients or middlewares execute a transaction by repeatedly sending a transaction operation to a database server and then waiting the response until this transaction is committed or aborted, referred to as interactive transactions. A recent survey [11] reported that 54% responders never or rarely use stored procedures in their applications. Upon realizing that, we re-examine the transaction processing of interactive transactions.

The front-end application as a client repeatedly sends transaction operations to a database server. The server first buffers the received operations from different clients in one or several queues. These operations are dequeued using a FIFO strategy, and then are concurrently executed by transaction engines. [19] has designed a scalable execution engine for interactive transactions. However, in spite of an efficient transaction engine, under high contention workloads, the scheduling order produced by FIFO strategy naturally consists of a lot of conflicts. That motivates us to review the overlooked scheduling problem in interactive transaction processing.

Operation scheduling is essential especially for interactive transactions, whose executions contain multiple network communications. Compared to stored-procedure model, an interactive transaction has a higher transaction latency. This leads to more access conflicts because of prolonged time of holding locks on hot data. As an execution of conflicting operation may result in dead locks or aborting transactions, it would be better to proactively schedule conflicting operations to produce an efficient execution order.

The main challenge of scheduling interactive transactions is that, a transaction is invoked by sending SQLs one-by-one to a database server, and in this case, the full read and write set of this transaction can not be acquired in advance. Therefore, we cannot directly find conflicting transactions before a conflicting operation is executed or fails to be granted a lock. In this paper we present POSIT, to the best of our knowledge, the first system prototype to adopt proactive operation scheduling for interactive transactions. By analyzing transactional logs offline, POSIT collects and groups a set of SQL template sequences where each sequence denotes executed SQLs of a transaction, and then mines the correlation of accessed record keys in different SQL statements for each group. At runtime, POSIT tracks hot data, and uses mined correlation to predict whether next operations of a started transaction will access hot records. In the end, POSIT uses a contention-aware operation scheduling algorithm to allow transactions to be executed in an efficiently interleaved fashion.

WOODSTOCK '97 El Paso, Texas USA
© 2007 ACM. ISBN 0-12345-67-8/90/01...$15.00
DOI: 10.475/123_4



1. **SELECT** (warehouse **WHERE** w_id = ***w_id***)
2. **SELECT** (district **WHERE** w_id = ***w_id*** and d_id = ***d_id***)
3. **UPDATE** (warehouse **WHERE** w_id = ***w_id***)
4. **UPDATE** (district **WHERE** w_id = ***w_id*** and d_id = ***d_id***)
5. **SELECT** (customer)
6. **UPDATE** (customer)
7. **INSERT** (history)

(a) transaction logics of *Payment*

1. **SELECT** (warehouse **WHERE** w_id = ***w_id***)
2. **SELECT** (district **WHERE** w_id = ***w_id*** and d_id = ***d_id***)
3. **UPDATE** (district **WHERE** w_id = ***w_id*** and d_id = ***d_id***)
4. **SELECT** (customer)
5. **INSERT** (neworder)

……

(b) transaction logics of *New-Order*

Figure 1: The previously executed SQL statements can predict which records will be updated by next UPDATE statements. In Payment and New-Order transactions of TPC-C, green rowkeys in UPDATE statements can be inferred by purple rowkeys in the previous SELECT statements.

2. POSIT OVERVIEW

Illustrating Examples for Operation Prediction. We observe that the primary keys of records updated by follow-up operations can be found from previous operations in a transaction. Figure 1 presents illustrating examples with the SQLs of two main OLTP transactions in TPC-C [3]: Payment and New-Order. In lines 1 and 2 of Figure 1(a), a row in the WAREHOUSE table is selected with the parameter value of W_ID, and a row in the DISTRICT table is selected with W_ID and D_ID, respectively. It's observed that, in line 3, the row in the WAREHOUSE is updated using the same W_ID value to line 1. This is a simple case that the primary key of a record to be updated can be predicted by the first SQL because they use the same rowkey for W_ID. Similarly, in line 4 the update on the row in the DISTRICT table can also be predicted as it uses the same W_ID and D_ID values to those of line 2. In the Payment transaction, update operations in lines 3 and 4 often incur conflicts among concurrent transactions. Although these transactions are implemented by client/server database APIs, conflicting operations are able to be predicted using the previously executed SQL statements. This motivates us to use predictable conflicting operations to generate an effective schedule for high contention workloads.

An Example for Operation Scheduling. In Figure 2, we present a motivating example to show how operation scheduling can yield better performance for conflicting transactions. Figure 2(a) demonstrates that conflicting transactions are easy to block each other under the FIFO strategy of scheduling received SQL operations. We assume the database server adopts two-phase locking as concurrency control protocol. Tx_1(Payment) updated W_1 and tries to update D_1, which leads to a conflict with Tx_2 (New-Order). Thus, the lock manager will block Tx_1 until the lock of D_1 is released. On the other hand, Tx_3 (Payment) also contends with Tx_1 for W_1 and will be blocked until Tx_1 commits. In this demonstrating examples, however, no write conflicts exist between Tx_2 and Tx_3, and Tx_1 conflicts with both of them. By taking advantage of predicted operations and tracked hot records, a proactive scheduler should predict that executing Tx_1 early is more likely to stall Tx_2 and Tx_3. In Figure 2(b), *the SQL UPDATE W_1 is deferred because it knows the next SQL of Tx_1 will update tuple D_1 which is doomed to failure since Tx_2 has acquired the lock of D_1*. As a result, delaying UPDATE W_1 in Tx_1 allows Tx_3 and Tx_2 to execute in parallel. In the method of handling received transactional operations by FIFO strategy, conflicting transactions are blocked or aborted after failing to request locks. POSIT aims to react to conflicts which will happen in concurrent transactions, and achieves more parallelism. Section 3.3 describes more details of our proposed scheduling algorithm. It should be noted that the early version of POSIT assumes transactions are executed in the isolation level of Read Committed (RC), which is the widely used isolation level in a practical setting.

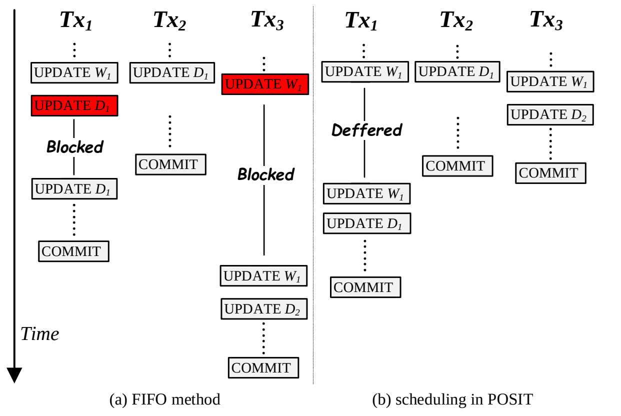


Figure 2: An example that illustrates the performance benefit with scheduling. UPDATE W_1 means updating warehouse table with the primary key W_ID equivalent to W_1.

POSIT Architecture. The framework of POSIT includes TxnPattern-Extractor, Temp-Tracker and Op-Scheduler (Figure 3). TxnPattern-Extractor is to reverse-engineer transaction logics based on offline transactional logs, and build the prediction model only for conflicting SQL statements in each kind of transactions. Temp-Tracker is to track popular records under the consideration of time factor. In real applications, hot records might change to cold as time goes on. Op-Scheduler is triggered to generate an effective schedule when hot records are accessed by concurrent transaction. Scheduled operations are enqueued onto queues with different priorities (e.g. High, Normal and Low priorities) of a working thread.

3. IMPLEMENTATION

3.1 TxnPattern-Extractor

Transaction patterns could be extracted from the tracking logs produced by DBMS. Each transaction log contains a sequence of SQL statements, which is started with START TRANSACTION and ended with COMMIT or ABORT. We first converts all raw SQL statement sequences appeared

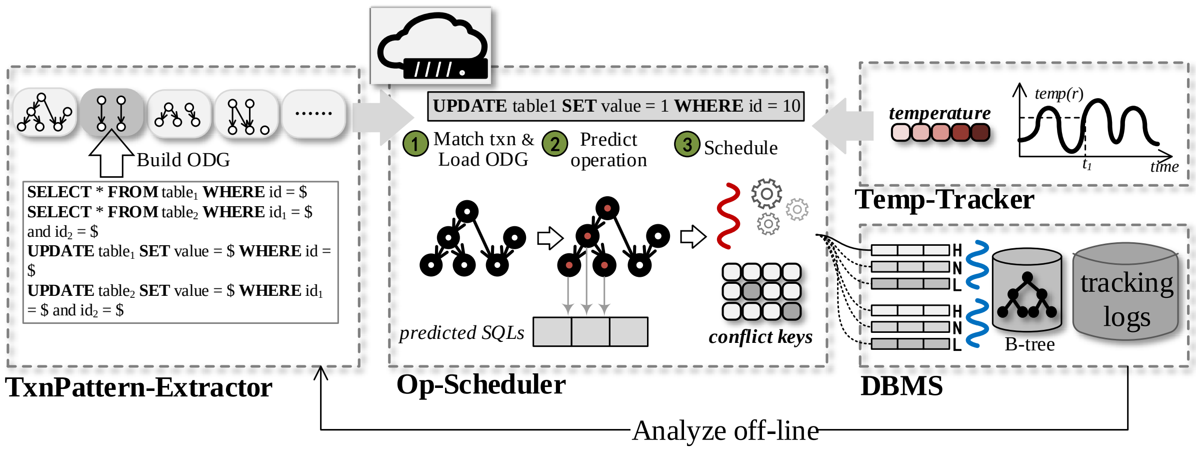

UPDATE table1 SET value = 1 WHERE id = 10
1 Match txn & Load ODG
2 Predict operation
3 Schedule
Build ODG
SELECT * FROM table1 WHERE id = $
SELECT * FROM table2 WHERE id1 = $ and id2 = $
UPDATE table1 SET value = $ WHERE id = $
UPDATE table2 SET value = $ WHERE id1 = $ and id2 = $
predicted SQLs
conflict keys
temperature
temp(r)
t1
time
Temp-Tracker
H
N
L
B-tree
tracking logs
TxnPattern-Extractor
Op-Scheduler
DBMS
Analyze off-line

Figure 3: POSIT contains three main components: TxnPattern-Extractor, Temp-Tracker and Op-Scheduler.

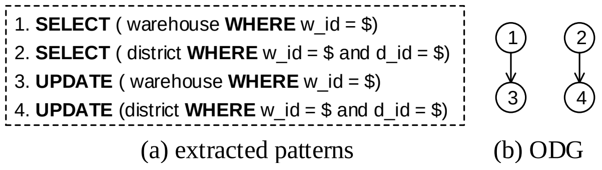

1. SELECT (warehouse WHERE w_id = $)
2. SELECT (district WHERE w_id = $ and d_id = $)
3. UPDATE (warehouse WHERE w_id = $)
4. UPDATE (district WHERE w_id = $ and d_id = $)
(a) extracted patterns
1
3
2
4
(b) ODG

Figure 4: TxnPattern-Extractor for *Payment*.

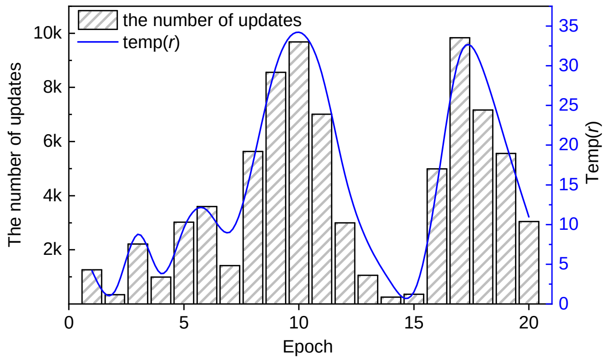


Figure 5: The actual number of updating a record and its temperature tracked by Temp-Tracker, where the probability of sampling is 1%, the cooling factor (p_1) is set to 0.7 and heating factor (p_2) is set to 0.3.

in transactional logs into a collection of SQL-template sequences by replacing parameter values in SQLs with variables. Then, the collected SQL-template sequences are grouped. If the size of each group is greater than a predefined threshold, the SQL-template sequence in this group is regarded as one kind of transaction in the current workload. Actually, POSIT is mainly concerned with these conflicting SQLs that really affect performance at runtime. We can distinguish them from always-succeeded SQLs by counting lock failures. After that, TxnPattern-Extractor considers how to infer the records accessed by conflicting SQLs according to their previous SQLs. In the early implementation of POSIT, we build Operation Dependency Graph (ODG) from a set of SQL statement sequences with parameter values for each kind of transaction. In ODG, a node represents a SQL. If there is an edge from a to b, it indicates b is a high conflicting SQL and its parameter values can be directly inferred from a.

Taking *Payment* of TPC-C for example again, Figure 4(a) presents an extracted group of SQL template sequence from transactional logs of *Payment*, where the third and fourth SQLs are identified to incur conflicts. Figure 4(b) shows the ODG that the third SQL uses the same parameter value with the first SQL. Similarly, the fourth SQL depends on the second SQL. At runtime, TxnPattern-Extractor first maps a transaction to a grouped transaction pattern, and then predicts which records are accessed by next operations. It's should be noted that extracting transaction pattern and building ODG are done offline without affecting transaction processing performance.

Also note that a transaction may contain control logics of branching or looping, in our initial implementation, POSIT identifies operational sequences that enter different branches as different transactions although they maybe belong to the same transaction. Besides, in this paper, we only mine direct mapping functions between parameter values of conflicting operations and those of their dependent, previous operations in ODG. Actually, this mapping function can be represented with a general linear or non-linear predictive model, which is trained by using parameter values from previous operations as input features, and parameter values of conflicting operations as predicted output. We will research it more deeply in future works.

3.2 Temp-Tracker

POSIT tracks popular records at runtime, where Temp-Tracker begins to heat up the temperature of a record which was modified frequently in a period of time. If the access frequency decreases, its temperature should be cooled down. We use a lock-free hash table *temp* to maintain the temperature statistics of hot spots in current workload. Since frequent updates to temperature statistics might potentially hurt performance, write operations are online sampled with a low probability (e.g. 1%) for updating temperature statistics.

With the passage of time, the temperature of all hot records is gradually cooling. Inspired by Newton's law of cooling [5], the temperature of a record has exponential decay, and it will rise up if this record receives many updates in a period of time. Temp-Tracker uses epoch as time unit, and a global epoch number is freshed periodically. A hot record keeps the last epoch in which its temperature is updated. The following formula is used for computing temperature.

$$temp(r) = \begin{cases} p_1 * \frac{temp(r)}{e^{curEpoh - lastEpoh}} + p_2, & \text{if } curEpoh \neq lastEpoh \\ temp(r) + p_2, & \text{if } curEpoh = lastEpoh \end{cases}$$

For a sampled update operation on record r, the formula first checks whether the current epoch (i.e. global epoch number) is equal to the last sampled epoch for updating r's temperature. If not so, $temp(r)$ should be decayed. The decay coefficient is decided by epoch difference $curEpoh - lastEpoh$ and cooling factor p_1. Besides, this update operation adds increments to $temp(r)$ with heating factor p_2. Some expired hot records may not be updated in recent epochs, so their temperatures will not be triggered to calculate and remains high. Garbage collection goes through the *temp* object at regular intervals, and cleans out these expired hot records. Figure 5 demonstrates the actual number of updating a record and its temperature tracked by Temp-Tracker.

3.3 Op-Scheduler

The basic intuition of Op-Scheduler is that transactions of accessing hot records should be scheduled. The SQL request of a transaction holding hot locks should be prioritized, and the SQL request doomed to failure should be deferred. The workflow of Op-Scheduler is decomposed into two stages: (1) predict next SQLs according to the previously arrived SQLs and transaction patterns extracted by TxnPattern-Extractor; (2) schedule the concurrent SQL by considering which data will be accessed by next SQLs and data's temperatures. In the prediction stage, a transaction is assigned to an extracted transaction pattern which has maximal prefix matching with arrived SQL sequences of this transaction. The ODG of this kind of transaction is used to predict which records will be updated by next, predicted SQLs.

To make system execute SQLs in an order determined by Op-Scheduler, we allocate multi-priority queues for worker threads. Tasks on the higher priority queue are more likely to be processed earlier and those on the lower priority queue are more likely to be processed later. In our implementation, each thread has three queues with high, normal, low priorities respectively. It's noted that starvation problem is avoided because SQL tasks with low priority just have relatively lower probability of being dequeued.

Another key data structure is *conflict_keys*. It's implemented by a lock-free hash table for tracking whether a hot record is going to be accessed by a transaction that has already acquired locks of other hot records. If Tx_1 has updated a hot record W_1 and its predicted SQLs contain another hot record D_1, then it registers to the conflict_keys by marking D_1 as true.

The main steps of POSIT scheduling algorithm are to decide whether the execution of a SQL task needs to be prioritized or deferred. In the following case, a SQL should be prioritized and pushed to a high priority queue: if its corresponding transaction has acquired one or several locks with

Algorithm 1: POSIT Scheduling Algorithm

```
   /* Scheduler handles packets from network and
      schedules them to different queues.                */
1  const $H$; // temperature threshold
2  function handle_packet
       Input: $pkt$
3      $ctx$ = get_ctx($pkt$.transid());
       // Prioritize a SQL task
4      if $need\_prioritization(ctx)$ then
5          push($pkt$, WORKER, HIGH_PRIORITY);
       // Defer a SQL task
6      else if $pkt.lock\_type == WRITE$ and
        $need\_deferral(ctx, pkt.key())$ then
7          push($pkt$, WORKER, LOW_PRIORITY);
       // Default scheduling
8      else
9          push($pkt$, WORKER, NORMAL_PRIORITY);
   /* Judge whether a SQL task needs to be
      prioritized.                                       */
10 function need_prioritization
       Input: $ctx$
11     for $r$ in $ctx.acquired\_locks$ do
12         if $temp(r) > H$ then
13             return $true$;
14     return $false$;
   /* Judge whether a SQL task needs to be
      deferred.                                          */
15 function need_deferral
       Input: $ctx$, $key$
       // first case:
16     for $r$ in $ctx.predicted\_SQLs$ do
17         if $temp(r) > H$ and ($r.locked$ or
            $conflict\_keys.find(r)$) then
18             return $true$;
       // second case:
19     if $conflict\_keys.find(key)$ then
20         return $true$;
21     return $false$;
```

high temperature (lines 5,6 and function *need_prioritization* in Algorithm 1). It is benefit for reducing conflicting probability by quickly finishing transactions that hold hot locks.

In other cases, the scheduler should defer executing a SQL and push it into a low priority queue (lines 7,8 and function *need_deferral*): (1) when one or more records to be accessed by predicted SQLs have been locked by other transactions, the transaction would be blocked or aborted by next SQLs. Thus giving up processing this SQL could bring more chances to other transactions (see an example in Figure 1); (2) when the record to be updated by this SQL is going to be accessed by a started transaction that has held hot locks, this SQL is preferred to delay and avoids blocking that started transaction because we hope transactions holding hot locks to finish as quickly as possible.

Finally, if this SQL is neither a conflicting task nor belongs to a contentious transaction, Op-Scheduler pushes it into a normal priority queue. Conflict keys and predicted_SQLs need to be updated after acquiring a hot lock successfully.

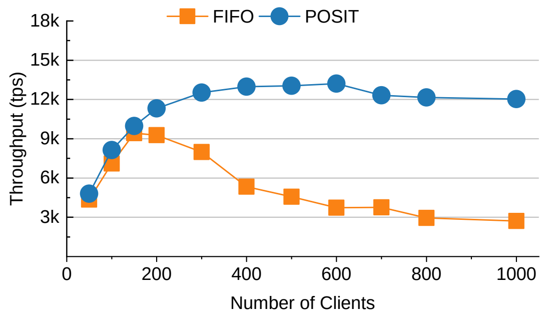


Figure 6: Throughput under various number of clients.

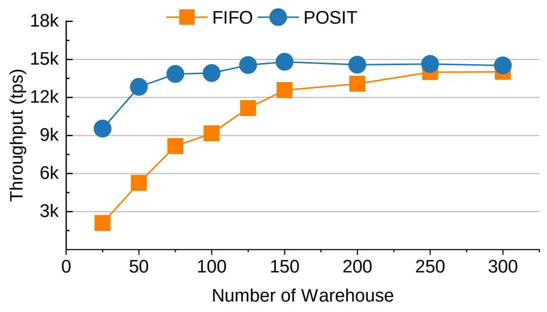


Figure 7: Throughput under various number of warehouses.

4. PRELIMINARY RESULTS

We evaluate the performance of POSIT by implementing a main memory database prototype with 19864 lines C++ code. The main memory database system provides a simplified database driver API to support interactive transactions. All clients connect to the database server using Gigabit Ethernet. This prototype adopts two-phase locking protocol and the workload is running under read committed isolation level. The locking implementation is based on a highly scalable, lightweight row locking scheme. The locking information are decentralized in each record with extra 64 bits as a hidden column. This hidden column contains transaction ID and read/write locking flags, and thus the lock request or release for a record is implemented by a single atomic operation. As we know, this kind of simple yet effective lightweight locking scheme is used by an open source version of Oceanbase [2], developed by Alibaba and served for Double 11 event [1]. Oceanbase created the maximal TPS record of online shopping in the world [1, 2]. Ren Kun and his co-authors optimized lightweight row locking under one-shot transaction model for high contention workloads [13]. It needs a transaction to acquire all locks before this transaction starts to execute.

Two scheduling methods are compared in the following experiments: (1) the widely used FIFO, where SQL tasks are processed according to their arrival orders; (2) Our POSIT, which proactively schedules SQL tasks for conflicting transactions.

We first run standard TPC-C benchmark with 50 warehouses by varying the number of clients. Figure 6 shows experimental results. Overall, POSIT performs better than FIFO. With the increasing of clients, the performance of FIFO first increases and then decreases. This is because access conflicts happen more frequently with more clients. POSIT achieves an increasing throughput with more clients and maintains stable performance with more than 300 clients. When using fewer clients, conflicts are rare, and POSIT has the same performance to FIFO. When conflicts increase, POSIT can identify hot spots and prioritize or defer the execution of conflicting SQLs to obtain more parallelism.

Next we use 300 clients and increase the number of warehouses to decrease workload contentions. Results are presented in Figure 7. POSIT could get a 4.5 times performance improvement than FIFO under high contentions (i.e. 25 warehouses). Using more warehouses, throughput of both methods scales linearly and POSIT could achieve peak performance earlier since it's proactive in avoiding conflicts. Under the workload (300 warehouse) with low contentions, we observe that the performance of POSIT and FIFO is very close to each other. This means that the overhead introduced by POSIT is negligible.

5. RELATED WORKS

Operating system scheduling. Scheduling is a common problem on how to assign limited resources to a set of jobs. Actually, in the field of operating systems, it has been studied for decades. The main problem that scheduling resolves is to decide which outstanding job requests are to be allocated resources (i.e., CPU, I/O). Scheduling algorithms have various optimization targets such as maximizing throughput and minimizing wait time. There are many different scheduling algorithms in operating system like first in first out(FIFO), earliest deadline first(EDF), shortest job first(SJF) [6]. Corbato et al. [4] proposed the multilevel feedback queue algorithm. If a process uses too much CPU time, it will be moved to a low-priority queue. Otherwise, when a process waits too long in a low-priority queue, it will be moved to a higher priority queue.

However, except the native FIFO, other scheduling algorithms can not be directly applied to database systems to support interactive transactions. The reasons are that: (1) the task information (i.e., the whole processing time or the deadline of an interactive transaction) cannot be determined in advance. (2) scheduler in database should also concern about lock conflicts besides hardware contentions.

Database system scheduling. FIFO policy is the default scheduling of SQL requests in many databases like MySQL, SQL Server and DB2 . Based on the assumption that transaction logics are implemented by stored procedure, Calvin is able to generate conflict-free schedule [15]. Calvin yields deterministic execution order according to the read and write sets of transactions before their executions. The intelligent scheduling proposed in [18] scheduled conflicting transactions into the same queue. Under one-shot transaction model, it first predicted all read/write set of a incoming transaction. Then, the conflict possibility is measured by comparing read/write sets with those running transactions in each queue. All above scheduling works avoid executing two conflicting transactions concurrently. However, in the case of interactive transactions, we can not decide whether a transaction conflicts with other before its execution. Therefore, existing transaction scheduling methods are not suitable in the context of interactive transactions.

Scheduling is also useful in other database components. Tian et al. [16] proposed a contention-aware lock scheduling.

It allows the lock requested by a transaction which blocks many other transactions to be granted with high priority. Lock scheduling depends on the traditional centralized locking manager to provide the information of all lock requests.

By leveraging information about locking contentions, QURO [17] reorders transaction code. This can be regarded as a kind of operation scheduling, and its goal is to reduce the locking time on high-conflict records. However, this scheduling is done by rewriting application codes. The scheduling in POSIT is transparent to clients and the database server.

Workload prediction. Prediction has many applications in transaction or query processing [14, 7, 9]. Andy Pavlo et al. [12] used Markov model to forecast the behaviors (e.g. accessed partitions, abort or commit) of an arriving transaction. The predicted information is used to choose the optimal partition to run this transaction or disable undo logging for a non-aborting transaction. [14] used Markov model to learn OLAP query patterns, and then it enables the data for next queries to be prefetched. A Ganapathi et al. [8] used kernel canonical correlation analysis to predict performance metrics (e.g. running time, resource usage). QB5000 [10] forecasted the query arrival rate by using linear regression and recurrent neural network in order that optimal indexes could be chose for the target workload in real-time.

6. CONCLUSIONS

Most database systems use the FIFO strategy to schedule SQL statements for interactive transactions. It leads to that high-conflict transactions are easy to block each other or frequently aborted. In this paper, we present POSIT, a proactive scheduler for interactive transactions. By predicting next operations of a running transaction and tracking hot data, POSIT determines to prioritize or defer a SQL task to get better parallelism. Preliminary results demonstrate that, under high contention workloads, POSIT could gain an almost five-fold improvement in performance than the classical FIFO scheduling.

7. REFERENCES

[1] China singles day. https://www.chinainternetwatch.com/tag/double-11/.

[2] Oceanbase. https://github.com/alibaba/oceanbase.

[3] Tpc-c benchmark. http://www.tpc.org.

[4] Multilevel feedback queue — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Multilevel_feedback_queue&oldid=838557282, 2018.

[5] Newton's law of cooling. https://en.wikipedia.org/wiki/Newton%27s_law_of_cooling, 2018.

[6] Scheduling (computing) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Scheduling_(computing)&oldid=845258399, 2018.

[7] N. Du, X. Ye, and J. Wang. Towards workflow-driven database system workload modeling. In *Proceedings of the 2nd International Workshop on Testing Database Systems, DBTest 2009, Providence, Rhode Island, USA, June 29, 2009*, 2009.

[8] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 592–603, 2009.

[9] C. Gupta, A. Mehta, and U. Dayal. PQR: predicting query execution times for autonomous workload management. In *2008 International Conference on Autonomic Computing, ICAC 2008, June 2-6, 2008, Chicago, Illinois, USA*, pages 13–22, 2008.

[10] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 631–645, 2018.

[11] A. Pavlo. What are we doing with our lives?: Nobody cares about our concurrency control research. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, page 3, 2017.

[12] A. Pavlo, E. P. C. Jones, and S. B. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB*, 5(2):85–96, 2011.

[13] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. In *Proceedings of the 39th international conference on Very Large Data Bases*, 2013.

[14] C. Sapia. PROMISE: predicting query behavior to enable predictive caching strategies for OLAP systems. In *Data Warehousing and Knowledge Discovery, Second International Conference, DaWaK 2000, London, UK, September 4-6, 2000, Proceedings*, pages 224–233, 2000.

[15] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. 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, 2012.

[16] B. Tian, J. Huang, B. Mozafari, and G. Schoenebeck. Contention-aware lock scheduling for transactional databases. *PVLDB*, 11(5):648–662, 2018.

[17] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. *PVLDB*, 9(5):444–455, 2016.

[18] T. Zhang, A. Tomasic, Y. Sheng, and A. Pavlo. Performance of oltp via intelligent scheduling. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 19-22, 2018*, pages 1288–1291, 2018.

[19] T. Zhu, D. Wang, H. Hu, W. Qian, X. Wang, and A. Zhou. Interactive transaction processing for in-memory database system. In *Database Systems for Advanced Applications - 23rd International Conference, DASFAA 2018, Gold Coast, QLD, Australia, May 21-24, 2018, Proceedings, Part II*, pages 228–246, 2018.