Performance of OLTP via Intelligent Scheduling

Tieying Zhang[*], Anthony Tomasic[†], Yangjun Sheng[‡], and Andrew Pavlo[§]
[*]Alibaba Inc. tieying.zhang@alibaba-inc.com
[†]Carnegie Mellon University tomasic@cs.cmu.edu
[‡]Carnegie Mellon University yangjuns@andrew.cmu.edu
[§]Carnegie Mellon University pavlo@cs.cmu.edu

***Abstract*—Current architectures for main-memory online transaction processing (OLTP) database management systems (DBMS) typically use random scheduling to assign transactions to threads. This approach achieves uniform load across threads but it ignores the likelihood of conflicts between transactions. If the DBMS could estimate the potential for transaction conflict and then intelligently schedule transactions to avoid conflicts, then the system could improve its performance. Such estimation of transaction conflict, however, is non-trivial for several reasons. First, conflicts occur under complex conditions that are far removed in time from the scheduling decision. Second, transactions must be represented in a compact and efficient manner to allow for fast conflict detection. Third, given some evidence of potential conflict, the DBMS must schedule transactions in such a way that minimizes this conflict. In this paper, we systematically explore the design decisions for solving these problems. We then empirically measure the performance impact of different representations on a standard OLTP benchmark.**

***Keywords*—online transaction processing, scheduling.**

I. Introduction

A significant source of performance degradation in OLTP DBMS is transaction conflict on shared data. Research on this problem has led to many variations of optimistic concurrency control (OCC) and pessimistic concurrency control (e.g., two-phase locking – 2PL) protocols. As the number of active transactions in the DBMS increases, the likelihood of conflict between transactions increases because transactions are assigned to queues without regard to conflicts. The key insight of this paper is simple: two transactions that are likely to conflict should not be run concurrently. Our approach is to model when two transactions might conflict and then schedule these transactions into the same run queue.

II. Challenges

The main challenge for our proposal consists of modeling the complex behavior of a database engine that results in transaction aborts.

A second challenge is to effectively aggregate incoming transactions into groups that are likely to conflict with each other. Given these groups, we can heuristically schedule transactions of each group linearly into an associated FIFO queue, thus insuring that the members of the group execute serially and (with high probability) do not generate an abort.

The third challenge is the *cost* of operations in this approach: the cost of maintaining the model, the cost of grouping, and the cost of scheduling transactions. In this paper, our target processing environment is main memory databases with OLTP workloads. Therefore we limit our design space to operations that are at most linear with respect to the length of a transaction statement.

III. Design

Consider the SQL update statement from the TPC-C benchmark (Figure 1). The SET and WHERE clauses of the statement provide some evidence to the circumstances of a commit or abort, in the following sense: any other transaction that references the same data is a candidate source of transaction conflict. That is, any other transaction that contains (e.g., `s_i_id=2`) may conflict with the transaction of Figure 1. In this paper, a *reference* is a sequence of attribute, operator, and value (or a boolean combination) that appears in SELECT, UPDATE, INSERT, and DELETE statements.

Thus, the circumstances of a transaction is defined as the statements and input parameters of the transaction. In particular, we *represent a transaction by the set of its references*. Although this representation of the circumstances is not as precise as, e.g., the read-write set of a transaction, the representation has the advantage of being compact and cheap to compute.

Finally, with respect to grouping, we realized that a group represents transactions that historically have conflicted with each other, and that a key simplification would be to *assign a group to a queue*. With these insights, the basic design framework becomes clear. When a new transaction arrives, references are extracted. Then the system computes the best group for a transaction from these references. Finally the transaction is assigned to the queue associated with the group.

```
UPDATE stock SET s_quantity=$1
 WHERE s_i_id=$2 AND s_w_id=$3
```

Fig. 1: A SQL Update Statement from the TPC-C benchmark – The variables $1, $2, and $3 are instantiated at execution time.

IV. System Architecture

The overall environment (Figure 2) consists of three layers – the incoming stream of transaction requests, our system, and a main-memory database system. The API between the system and the database is simple, consisting of three functions: (i) the ability to capture incoming transactions, (ii) the ability to queue a transaction in a specific run queue of the database, and (iii) the ability to log two events that occur during transaction processing: a transaction abort, and a transaction commit. This environment and API implies our techniques can be "bolted on" to any DBMS with little effort.

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00132

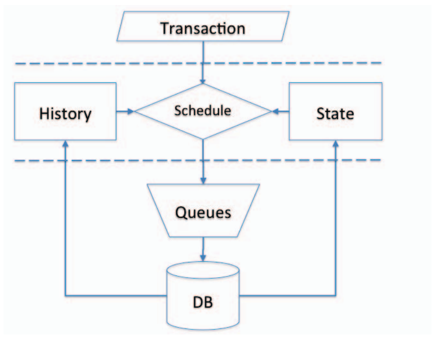

Fig. 2: Functional System Architecture – The Transaction component represents an incoming transaction. The History component represents the historical model of aborted transactions. The State component represents the mapping of transaction references to queues. The Schedule component chooses the queue to place the incoming transaction. The Queues component represents the set of run queues. The DB component is a main-memory database system. The dotted lines separate conventional components from the new system.

A. *Overview*

Our system consists of three components: (i) a *History* component that extracts and aggregates information each time the system detects an abort or commit, (ii) a *State* component that maps transaction references to queues, and (iii) a *Schedule* component that incrementally performs the grouping operation.

History Component: The History component is an aggregated historical summary of the references extracted from the database when a transaction aborts or commits. The History collects aggregated statistics depending on abort and commit events in the database. The History is implemented as a hash table. The key is a reference and the value is a pair (abort, commit) of integer counters that count the number of transactions that have either aborted or committed that contain the associated reference. A counter is incremented every time a transactions aborts or commits. A counter is never decremented. The History is operational during randomized scheduling to capture the true distribution of transaction commits and aborts. During intelligent scheduling, the History component serves as a data repository.

State Component: The State component records for each reference the number of transactions in each queue that contain the reference. The State is implemented as an key-value hash table. The key is reference and the value is an array of integers indexed by queue number. An array value records the number of times the associated reference appears in the transaction queue. The values are incremented when a transaction is queued. For each reference, we track the arrival rate per second, the total transactions in the system, and the queue with the maximum count (Table I).

Schedule Component: Given the History and the State and an incoming transaction, it computes a score for each queue and then schedules the incoming transaction into the best queue.

The three other components, Transaction, Queues, and DB, are standard components present in every database system. The Transaction component in the figure represents an incoming transaction. The Queues component is the set of run queues of the Database ('DB') component.

B. *Stealing*

In our implementation, an idle thread *steals* a transaction from the queue of another thread (in random fashion).In other words, in low utilization environments, a transaction may not be executed by the thread in which it was originally queued. This rule prevents the thread from sitting idle. Of course, a stolen transaction will immediately start running concurrently with the transaction running from the non-idle thread. Since these transactions may conflict, this behavior is not ideal from our perspective.

Note that once a transaction is assigned to a specific queue, it eventually must be executed by the thread associated with this queue, unless it is stolen. And once the transaction is stolen by a thread, the stealing thread always executes the transaction to completion, because aborts are immediately retried without the possibility for another thread to steal a transaction retrying after an abort.

C. *Bookkeeping*

The History and State components use libcuckoo, a thread-safe and lock-free data structures [3], but the components do not globally synchronize with the Schedule component in a transactional sense. Thus the Schedule component may make a scheduling decision based on (slightly) stale information. This trade-off for performance over accuracy is well worth it, since the cost of synchronization is high. The Schedule component is robust to minor inaccuracies in bookkeeping because both History and State components store an aggregated history of behavior that only slowly changes over time. Our implementation ensures that access to these structures is not a contention point in the system.

V. Policies

The overall process for scheduling a new arrived transaction into a queue consists of several steps: (i) extract the references from the transaction according to a *policy*, (ii) look up the references in the History to gather historical evidence on the conflict rates of references, (iii) produce a conflict likelihood score by combining this evidence with the historical state mapping of references to queue, and (iv) add the transaction to the queue with the highest likelihood of conflict.

A *policy* consists of a set of design choices. These choices are (i) the form of references that are extracted from the transaction and stored in the History and State components and (ii) rules about the assignment of transactions to queues in the Schedule component. In our evaluation, we run experiments with different policies to understand the consequences of various design choices. We introduce the policies explored in this paper with a running example.

Example: Suppose the system is in the state described in Table I. Consider the arrival of a new transaction (Figure 1) where `$1=6`, `$2=2`, and `$3=5`. The system extracts the references `s_quantity=6`, `s_i_id=2` and `s_d_id=5`. It then looks up the references in the History, and finds the number of aborts as 20, 0 and 20, respectively. The system then scores the likelihood of each queue containing a transaction that will conflict with the

#	current transactions
1	`UPDATE stock SET s_quantity=7 WHERE s_i_id=1 AND s_w_id=5`
2	`UPDATE stock SET s_quantity=7 WHERE s_i_id=1 AND s_w_id=5`
3	`UPDATE stock SET s_quantity=6 WHERE s_i_id=2 AND s_w_id=5`
4	`UPDATE stock SET s_quantity=6 WHERE s_i_id=2 AND s_w_id=5`

History

reference	abort	commit
s_quantity=6	20	60
s_quantity=7	40	20
s_i_id=1	20	20
s_i_id=2	0	20
s_w_id=5	20	20

State

	queue					
reference	1	2	3	R	T	Q
s_i_id=1	2	0	0	3.1	2	1
s_w_id=5	2	1	1	3.0	4	1
s_i_id=2	0	1	1	1.9	2	2
s_quantity=7	2	1	1	1.9	2	1
s_quantity=6	0	1	1	1.9	2	2
total	6	4	4			

TABLE I: Example – An example of an input sequence, History and State for the repeated execution of the SQL in Figure 1. Column 'reference' is the extracted representations of the transaction's statements. Column 'abort' is the historical count of the number of transactions containing a reference that have aborted. Column 'commit' is the count of the number of commits in the same way. A transaction may abort several times before committing. The 'queue' columns are the historical count of the transactions scheduled into the reference queue. Row 'total' is the total of the queue columns. The R column is the arrival rate of transactions with the associated reference. The T column is the total transactions for this reference in the system. The Q column is the queue that contains the largest number of references.

arriving transaction by summing the product of the abort count with the reference queue count in the State table. For queue one: $20 \cdot 0 + 0 \cdot 0 + 2 \cdot 2 = 24$, queue two: $20 \cdot 1 + 0 \cdot 1 + 2 \cdot 1 = 22$, and queue three: $20 \cdot 1 + 0 \cdot 1 + 2 \cdot 1 = 22$. The system then chooses the queue with the largest score (queue one), assigns the transaction in the wining queue, and then increments the state values to reflect this assigned transaction. An entire transaction is always placed onto a single run queue.

Consider the subsequent arrival of a new transaction that contains the reference `s_i_id=3`. The above score cannot be computed because the reference does not occur in the History. In this case, a *default* strategy is used: chose the queue with the smallest total number of references.

Count and Fraction policies In the Example, the strength of evidence for a reference to indicate a transaction abort is the historical count of the number of aborts. We term this design decision the *Count* policy. Intuitively, however, another metric could be used: the fraction of aborts over the total attempted (aborted + committed) transactions for a reference. We term this design decision the *Fraction* policy.

Sum and Max policies Revisiting the basic steps of determining a score for a transaction, consider the step of the score computation that sums across all the references in a transaction. The intuition behind this step is that transactions should weight the evidence from all the references that occur. We term this design decision the *Sum* policy. However, a different design choice at this point is to focus on a single reference "hot spot" that historically has the highest abort count across the workload and use only this reference in the score the transaction with the state table. This design decision is called the *Max* policy.

Comparing these two policies examines two opposing views of the evidence provided by a reference. The Max policy attempts to find the single best reference to use to group transactions into queues, somewhat like a data partition policy. In the Sum policy, all references are equally weighted to determine a particular queue.

Literal and Canonical policies For all policies so far references are literally extracted from each transaction. Those literal references are used by the History and State components. This design decision we call the *Literal* policy.

However, database administrators partition databases based on an underlying domain concept such as warehouse id. This concept is not captured by literal references because, e.g., foreign key references to the same underlying domain have different literal references. In this section we consider a canonical form of references where references are normalized to the underlying domain. For example, for the SQL of Figure 1 with parameters `$1=6`, `$2=2`, and `$3=5`, the *Canonical* policy refers the underlying domain, thus the canonical references are `s_quantity=6`, `i_id=2` and `w_id=5`.

The policy comparison of Literal versus Canonical explores two effects. First, the literal policy dimension results in a more fine-grained and precise representation, but also a sparser representation because there are many more unique literal references than canonical ones. Canonical references produces a smaller, more compact, less fine-grained representation. Second, the canonical policy requires additional implementation work. Literal references are straight forward to implement by examination of the parse tree. Implementing canonicalization requires deeper analysis to trace each reference to its underlying domain. The information required for the trace may be explicitly recorded in the schema (through, say, explicit domain statements) or a model can be constructed to determine likely candidates [1]. For our experiments, we simply hand-code the canonical references for a transaction.

Single and All policies In addition, we compared the extraction of single literals as references (the Single policy) versus using the entire condition as a reference (the All policy). For example, in Figure 1, the update has a boolean AND condition in the WHERE clause. With the All policy, this clause generates a single (literal) reference, e.g., `s_i_id=2 AND s_w_id=5` instead of two references. With the canonical and all policies, the single reference is `i_id=2 AND w_id=5`. The Single policy slices up complex conditionals and models each reference individually. This technique reduces the total number of references but does not model the boolean condition.

VI. Results and Discussions

The design space we explore contains multiple independent dimensions, so we ran a k-factor experiment to determine baseline performance compared to two other systems: (i) a vanilla policy with random assignment of transactions to threads (labeled 'Original' in the results) and (ii) a Hard Partition assignment of transactions to threads based on the warehouse id in the transaction. This result serves as the best case performance to match.

We ran all possible combinations of the policies under OCC and 2PL for TPC-C [9] (NewOrder and Payment) using Peloton [6] to compare policies. In this experiment, we throttled the system by fixing average response time to 500 ms and measuring throughput and abort rate at this response time. This experimental design roughly corresponds to the client response time limit in TPC-C. The results in Table II show that generally the best performing policy is a combination of Count, Max, Canonical, and Single.

Policy				TPCC-OCC		TPCC-2PL	
				tps	rate	tps	rate
Original				35.567	0.45	1,412	0.99
Hard Partition				43,137	0.01	39,338	0.29
Fraction	Sum	Literal	S	29,863	0.45	31,776	0.69
			All	28,670	0.44	36,146	0.51
		Canonical	S	29,265	0.44	31,715	0.66
			All	29,489	0.43	30,154	0.71
	Max	Literal	S	29,399	0.44	37,195	0.52
			All	28,659	0.45	4,986	0.97
		Canonical	S	29,489	0.44	29,655	0.72
			All	29,463	0.43	30,289	0.72
Count	Sum	Literal	S	29,587	0.44	16,899	0.89
			All	28,865	0.44	19,998	0.86
		Canonical	S	29,783	0.44	6,748	0.97
			All	28,990	0.44	16,787	0.91
	Max	Literal	S	27,899	0.41	28,881	0.71
			All	32,175	0.29	36,578	0.38
		Canonical	S	**40,612**	**0.07**	**38,125**	**0.29**
			All	33,763	0.24	31,590	0.71

TABLE II: 2^k Factor Experiment Results – The performance of an enumeration of policy combinations for TPC-C. Original means that transactions are simply randomly scheduled into queues as per usual. The column 'tps' reports transactions per second and column 'rate' reports abort rate. 'S' stands for Single. The OCC is under 19 warehouses and 2PL is under 30 warehouses.

VII. Related Work

Ic3 [10] chops transactions into pieces and analyzes the dependency between transaction pieces. Ic3 uses chopping technique, that is introduced by [7], to construct a dependency graph and maintains the dependency for running transactions. The main similarity between this work and our approach is that both analyze dependencies transactions to improve performance. However, our dependency structure is implied in the statistical relationship between references that occur in transactions, as opposed to an explicit dependency model.

TProfiler [2] is an instrumentation tool that allows a developer to quickly identify sources of performance variance. Using this tool, the authors introduce VATS as a intelligent scheduler of lock-granting that minimizes variances in MySQL. This scheduler also improves mean latency. In contrast, our work focuses on a different granularity of scheduling transaction based on an analysis of transactions before execution starts. In theory, the two approaches are entirely complimentary. Future work will permit a more detailed comparison of these approaches.

In [4], transaction reordering at a high level is shown to improve performance for materialized views and sequential scans. This work provides specific execution conditions for performance improvement where as our work simply examines similarities in the syntactic form of transactions.

In H-Store [8], data is partitioned through analyzing store procedure and each thread corresponds a partition. The transactions execute serially for a given partition. In DORA [5], a transaction is split up into sub-transactions and a sub-transaction is assigned to a single partition to execute.

VIII. Conclusion

In this paper, we provide preliminary evidence that scheduling of OLTP transactions offers both flexibility and improved performance. The key is that transaction scheduling can be improved by observing the history of transaction aborts and integrating this information into a design that improves transaction throughput, under high load, without significantly impacting response time. The design is remarkably similar to data partitioning, a technique that is the basis of an entire class of OLTP architectures, but without requiring a partition specification from a database administrator.

References

[1] I. T. Bowman. *Scalpel: Optimizing Query Streams Using Semantic Prefetching*. PhD thesis, University of Waterloo, 2005.

[2] J. Huang, B. Mozafari, G. Schoenebeck, and T. F. Wenisch. A top-down approach to achieving performance predictability in database systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 745–758. ACM, 2017.

[3] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.

[4] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Transaction reordering. *Data & Knowledge Engineering*, 69(1):29–49, 2010.

[5] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2):928–939, Sept. 2010.

[6] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, et al. Self-driving database management systems. In *CIDR*, 2017.

[7] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3):325–363, Sept. 1995.

[8] 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 *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[9] The Transaction Processing Council. TPC-C benchmark (revision 5.9.0), 2007. `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`.

[10] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1643–1658, New York, NY, USA, 2016. ACM.