Scheduling OLTP Transactions via Learned Abort Prediction

Yangjun Sheng
Carnegie Mellon University
Pittsburgh, PA
yangjuns@andrew.cmu.edu

Anthony Tomasic
Carnegie Mellon University
Pittsburgh, PA
anthony.tomasic@gmail.com

Tieying Zhang
Alibaba Database Group
tieying.zhang@alibaba-inc.com

Andrew Pavlo
Carnegie Mellon University
Pittsburgh, PA

ABSTRACT

Current main memory database system architectures are still challenged by high contention workloads and this challenge will continue to grow as the number of cores in processors continues to increase [23]. These systems schedule transactions randomly across cores to maximize concurrency and to produce a uniform load across cores. Scheduling never considers potential conflicts. Performance could be improved if scheduling balanced between concurrency to maximize throughput and scheduling transactions linearly to avoid conflicts. In this paper, we present the design of several intelligent transaction scheduling algorithms that consider both potential transaction conflicts and concurrency. To incorporate reasoning about transaction conflicts, we develop a supervised machine learning model that estimates the probability of conflict. This model is incorporated into several scheduling algorithms. In addition, we integrate an unsupervised machine learning algorithm into an intelligent scheduling algorithm. We then empirically measure the performance impact of different scheduling algorithms on OLTP and social networking workloads. Our results show that, with appropriate settings, intelligent scheduling can increase throughput by 54% and reduce abort rate by 80% on a 20-core machine, relative to random scheduling. In summary, the paper provides preliminary evidence that intelligent scheduling significantly improves DBMS performance.

CCS CONCEPTS

• **Information systems → Database transaction processing**; • **Computing methodologies → Machine learning**;

KEYWORDS

machine learning scheduling

ACM Reference Format:
Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. 2019. Scheduling OLTP Transactions via Learned Abort Prediction. In *International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM'19), July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3329859.3329871

1 INTRODUCTION

Transaction aborts are one of the main sources of performance loss in main memory online transaction processing (OLTP) database management systems (DBMS) [23]. Current architectures for OLTP DBMS use random scheduling to assign transactions to threads. Random scheduling achieves uniform load across CPU cores and keeps all cores occupied. For workloads with a high abort rate, a large portion of work done by CPU is wasted. In contrast, the abort rate drops to zero if all transactions are scheduled sequentially into one thread. No work is wasted through aborts, but the throughput drops to the performance of a single hardware thread. Research has shown that statistical scheduling of transactions using a history can achieve low abort rate and high throughput [25] for partitionable workloads. We propose a more systematic machine learning approach to schedule transactions that achieves low abort rate and high throughput for both partitionable and non-partitionable workloads.

The fundamental intuitions of this paper are that (i) the probability that a transactions will abort can be accurately modeled through machine learning, and (ii) given that an abort is predicted, scheduling can avoid the conflict without loss of response time or throughput. Several research works [1, 12, 16, 17] perform exact analyses of aborts of transaction statements that are complex and not easily generalizable over different workloads or DBMS architectures. Our approach uses machine learning algorithms to model the probability of aborts. Our approach is to (i) build a machine learning model that helps to group transactions that are likely to abort with each other, (ii) assign each group of transactions to a first-in-first-out (FIFO) queue, and (iii) monitor concurrency across cores and adjust for imbalances.

2 SYSTEM ARCHITECTURE OVERVIEW

The overall environment (Figure 1) consists of three layers – the incoming stream of transaction requests, our scheduler, 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, (iii) the ability to log the SQL of two events that occur during transaction processing: a transaction abort and a transaction commit, and (iv) the ability to report real-time transaction response times. This environment and API implies our approach can be "bolted on" to any DBMS with little effort. Our system does however require that for a (non user) aborted transaction, the system can identify at least one transaction that 'caused' the abort. Typically the causing transaction holds a lock on a row or modified the row during the aborted transaction's execution.

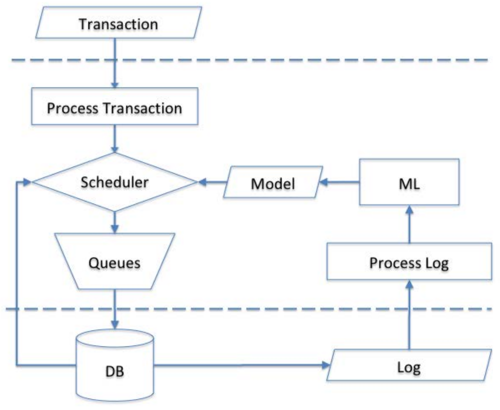


Figure 1: Functional Architecture – The Transaction component represents an incoming transaction. The Process Transaction component converts the transaction into its feature representation. The Scheduler component chooses the queue to place the incoming transaction. The Queues component represents the set of run queues. The DB component is a main-memory database system. The Log component collects transaction execution results for model training purposes. The Process Log component converts the log into its feature representation for training. The machine learning component trains the processed log to produce the Model. The dotted lines separate conventional components from our scheduler. The DB component also provides real-time transaction response time information to the scheduler.

3 SUPERVISED MACHINE LEARNING

Our approach is to learn a classifier that classifies two transactions, T_A and T_B, into Abort if they will abort each other when run concurrently (with high probability) or Commit if they do not conflict.

3.1 Training Data

Each transaction indirectly describes information about which tuples, attributes and tables it will read and write in the database. To access this information, some research uses the read/write set of a transaction, but this set of information is large and dynamically changing. Instead, the string representation of the SQL statements of a transaction are coded into a feature space. The SQL statements are a lightweight feature representation of the transaction.

3.1.1 Features. As described above, the primary goal of the model is to check if two transactions will conflict with each other. Therefore, the features should be derived from both transactions, T_1 and T_2. In particular, we want to define a function

$$trans(T_1, T_2) \rightarrow vector$$

to transform two transactions into a feature vector and train our model to classify this vector into Abort or Commit.

For example, in TPC-C, a transaction T might want to read rows where warehouse id is equal to 10. Then T has a reference `W_ID=10` and the string `W_ID=10` is considered as a feature of T. More precisely, any instance of *attribute operator value* in a **WHERE** clause of a **SELECT**, **DELETE** or **UPDATE** statement is a feature. All values of an **INSERT** are also features. We do not distinguish between reads and writes. If T both read and write rows `W_ID=10`, it has only one such string. The function $Features(string) \rightarrow F$ maps from a SQL string to a set of features F. Note that any particular transaction produces only a few features.

To produce a compact representation with a fixed size independent of the size of the transaction, we apply feature hashing, a fast and space-efficient way of converting an arbitrary number of features into indices in a fix-size vector. The function $Hash(T)$ constructs the union of the set of the features F of the statements of the transaction and generates a binary vector of fixed length k. Given two transactions T_1 and T_2, we now have two vectors $Hash(Features(T_1)) = V_1$

and $Hash(Features(T_2)) = V_2$. Let $V_3 = V_1 \& V_2$ be the binary logical AND of V_1 and V_2. The final feature vector is the concatenation of these three vectors,

$$trans(T_1, T_2) = V_1|V_2|V_3$$

The vector V_3 encodes tuples, columns or tables that will potentially be touched by both transactions T_1 and T_2. In our experiments our feature vectors are 1k bits in length.

3.1.2 Canonical Features. Attribute names that appear in a schema are arbitrary, independent of the underlying domain concept. For instance, in TPC-C benchmark, the column `W_ID` in `WAREHOUSE` table and `D_W_ID` in `DISTRICT` table both represent warehouses in this database. However, their string representations in SQL are not the same (`W_ID=1` and `D_W_ID=1`). While they express the same entity, based on the hashing function described in previous section, these two strings hash to different indices in the feature vector (baring collision). Research has shown that canonical features are more favorable than literal strings for representing transactions [25]. Therefore, we adopt canonical features converting string representations of attributes to a canonical version.

Suppose a TPC-C *NewOrder* transaction contains following SQL statements:

```
SELECT W_NAME, W_STREET_1, W_STREET_2, W_CITY,
FROM WAREHOUSE WHERE W_ID = 10
...
SELECT D_NAME, D_STREET_1, D_STREET_2, D_CITY
FROM DISTRICT WHERE D_W_ID = 10 AND D_ID = 4
...
```

Then the feature representation for these two SQL statements is:

```
W_ID = 10,..., W_ID = 10, D_ID = 4, ...
```

That is, each of these three strings is hashed and the corresponding bit is turned on in the feature vector. This representation discards most of the meaning of the SQL query and concentrates on representing only the read/write footprint of the transaction. The representation is invariant to the ordering of expressions in conjunctions and disjunctions. More complex SQL expressions, such as range expressions, are reserved for future work.

3.1.3 Training. Training data is gathered by observing the log while the system is running. Every time a transaction commits or aborts, the event is logged as one of two possible abstract triples:

- (Aborted Transaction, Conflicting Transaction, label `abort`)
- (Committed Transaction, Any Concurrent Transaction, label `commit`)

Sampling the log under random scheduling generates a distribution of transactions and abort/commit independently of our subsequent scheduling decisions. Concretely, if T_1 is aborted by T_2, then we log the pair $(trans(T_1, T_2), 1)$ into our log where 1 indicates an abort. To obtain vectors in the commit set, we used the following approach: when T_1 commits, we randomly pick a transaction, T_2, that is running currently with T_1 and add $(trans(T_1, T2), 0)$ into our log. The log serves directly as the training and test set for choosing a learning algorithm.

Note that only one conflicting transaction is chosen to record in the log. However, since the choice is random, the log implicitly records a random sample of the distribution of conflicting transactions, exactly in proportion to their occurrence in the workload.

3.1.4 Model Evaluation. We will assume that the hypothesis space is linear and defer non-linear models for future work. Our learning algorithm needs to be cheap to train and very fast to evaluate on a pair of transactions. Logistic regression is both cheap to train and can evaluate an example in $O(n)$ time where n is the number of 1 bits in the feature vector, using a sparse representation. Example evaluation time is proportional to the sum of the string lengths of the two transactions. In practice n is small (less than 20) and the evaluation consists of multiplication of each 1 feature vector bit by a scalar and summing to produce a final score. Another advantage is that the score for logistic regression can be interpreted as the probability that the transaction pair will produce an abort. The trained model generates parameters for a function

$$M(T_1, T_2) \rightarrow P(commit/abort)$$

that predicts abort probabilities for any transaction pair (T_1, T_2). In [15] we report a set of experiments that show this model has an accuracy of > 95% on a balanced workload.

3.2 Scheduling Algorithms

Our task in this section is to design an algorithm using M that assigns a transaction, T_{new}, into a queue in a way that avoids aborts and increases throughput compared to random scheduling.

Consider a transaction T_k already queued and waiting to execute. Suppose $M(T_{new}, T_k)$ has low abort probability. A low abort probabilities does not make it clear which queue to assign T_{new} since some other concurrent transaction may have a high abort probability. Suppose $M(T_{new}, T_k)$ predicts a high abort probability. Then if these two transactions run concurrently, with high probability at least one abort will occur, perhaps two (depending on the details of the concurrency control system). Assigning transaction T_{new} into a different queue than T_k may succeed as long as the database does not try to execute them at the same time. Concurrent execution depends in this case on there being the same amount of work in front of each queued transaction, so that they both arrive to the front of the queue at the same time, and are processed concurrently. However, predicting the total work in a queue is difficult. The simplest heuristic places T_{new} into the same FIFO queue after T_k so they never execute concurrently.

With this heuristic in mind, the algorithm computes

$$Enqueue(T_{new}, queue(\underset{T_i}{\operatorname{argmax}} M(T_{new}, T_i))$$

where $queue(T)$ is a function that returns the queue of transaction T. That is, for a new transaction T_{new}, search for T_{high} in some queue with the highest abort probability $M(T_{new}, T_{high})$, then assign T_{new} to the same queue as T_{high} (algorithm 1).

Algorithm 1 Assign(T_{new})

```
p = −1
q = −1
for $T \in \mathbb{T}$ do
    if $M(T_{new}, T) > p$ then
        $p = M(T_{new}, T)$
        $q = queue(T)$
    end if
end for
if $p > -1$ then
    Assign $T_{new}$ to FIFO $q$
else
    Assign $T_{new}$ randomly
end if
```

3.2.1 Search Scheduling Algorithm. We can search queues in two different ways: Breadth First Search (BFS) and Depth First Search (DFS). Extensive analysis and experimentation have shown that DFS performs poorly compared to BFS. DFS spends much more time searching for high probability transactions. See [15] for details about the implementation. The worst case computation cost, in terms of model evaluations, for both algorithms is $O(n)$ where n is the number of waiting transactions. This cost is paid by every queued transaction.

The combination of the model evaluation and the heuristic of scheduling aborts into the same queue will tend to group high probability conflicting transactions into the same queue. We can exploit this behavior to reduce the computation cost of finding the best queue to schedule a transaction collapsing the feature vectors of the transactions in the queue to a centroid that represents the "average" transaction.

3.2.2 Balanced Vector Scheduling Algorithm. The Vector scheduler creates a representative transaction R_i for the history of transactions in each queue q_i. R_i represents transactions in q_i by averaging the feature vectors of the transactions enqueued into q_i. That is, the features of transaction are bit vectors but the centroid representing each queue is a vector of floats. The model M remains unchanged for training purposes. The difference is in instance evaluation. To determine the best queue for a new transaction T_{new} the algorithm computes

$$Enqueue(T_{new}, \underset{i}{\operatorname{argmax}} = M(T_{new}, R_i))$$

See [15] for details about the implementation. The computational cost in terms of model evaluations is $O(n)$ where n is the number of queues in the system.

3.3 Supervised Scheduler Summary

In summary, the transactions assignment execution time of BFS is $O(n)$ where n is the number of queued transactions, less the early stop criterion of finding a younger transaction or finding a high abort probability transaction. The transaction execution time of Balanced Vector Assign is $O(kd)$ where k is the number of queues, d is the length of the feature vector (in a dense representation of floats), plus some additional cost for detecting imbalances in response times. The vector centroids are updated with each newly queued transaction (but not on transaction commit) and thus represent a history of transactions.

4 UNSUPERVISED MACHINE LEARNING

The algorithms in the prior section used supervised learning to construct a model and then applied that model to the scheduling problem in various ways. An alternative is to discover clusters of transactions that are likely to abort with other transactions in the same cluster if run concurrently. Unsupervised clustering algorithms are used to form k clusters, one for each queue. When a new transaction arrives, the algorithm assigns the transaction to the cluster (queue) closest to the new transaction. Each cluster represents a group of similar types of aborts.

4.1 Clustering Model

Each abort instance provides a clustering example since the abort suggests clustering the two transactions, the aborted transaction and the transaction that caused the abort, into the same cluster. Our model essentially learns a clustering model from these aborts. In this model, committed transactions do not write to the log because commit information is not necessary.

4.1.1 Features. Given two transactions T_1, T_2 such that T_1 aborted due to T_2, then (T_1, T_2) is an instance of abort recorded in the log. $F(\cdot)$ extracts features from (T_1, T_2) to form a feature vector as follows. Let $V_1 = Hash(T_1)$ and $V_2 = Hash(T_2)$. Then $F(\cdot)$ is the logical `AND` of these two vectors:

$$F(T_1, T_2) = V_1 \& V_2$$

The assumption here is that V_1 and V_2 encode the salient abort features of T_1 and T_2, respectively, and $V_1 \& V_2$ encodes evidence for the features that caused T_1 to be aborted. For example, suppose we set the vector size to be 8 and we have two transactions.

T_1 : `WAREHOUSE_ID=1 ITEM_ID=123 USER_ID=10`

T_2 : `WAREHOUSE_ID=1 ITEM_ID=456 USER_ID=20`

Assume that the hash functions hashes the string `WAREHOUSE_ID=1` to index 2 (starting from 0) (without collision) and suppose the results are

$$Hash(T_1) = V_1 : 00100101$$
$$Hash(T_2) = V_2 : 10100010$$

Then,

$$F(T_1, T_2) = V_1 \& V_2 = 00100000$$

which is a vector that represents aborts caused by sharing the attribute `WAREHOUSE_ID=1`.

4.1.2 Training. Training data is gathered by observing the log while the system is running under random scheduling as in supervised learning. However, only log abort instances are logged. In particular, if a transaction aborted is by another transaction, we log

- (Aborted Transaction, Conflicting Transaction)

4.1.3 Clustering Algorithm. We use the k-means scheduling algorithms with a Euclidean distance function $D(V, W)$. Although the decision boundary of k-means is linear, this algorithm converges relatively quickly in practice but more importantly can evaluate new instances quickly, on the order of O(k) centroid comparisons. Each centroid, in a spare representation, is of length f, the number of features in the centroid. If $V = (v_1, v_2, ..., v_n)$ and $W = (w_1, w_2, ..., w_n)$, then

$$D(V, W) = \sqrt{\sum_{i=1}^{n}(v_i - w_i)^2}$$

4.2 Scheduling Algorithm

After running the clustering algorithm, we obtain k centroids, denoted by $C_1, C_2, ...C_k$. The next question is to design a scheduling algorithm that schedules new transactions. Given a new transaction T_{new}, compute the Euclidean distance between T_{new} and $C_1, ...C_k$ and pick the "closest" cluster C_c and assigned T_{new} to queue c.

The reader may notice that this method for the evaluation of an instance is an abuse of the normal machine learning framework because the training data distribution is different from the new instance data distribution. Each cluster is designed to represent a type of abort produced by a pair of transactions, but the new instance represents only a single transaction. Computing the distance between a transaction and an abort seems to be meaningless. Intuitively however, the feature vector of T_{new} contains the features associated with an abort, along with some other features. Empirically, if the distance $D(V_{new}, C_i)$ is small, then T_{new} is more likely to have abort type i. Since each cluster represents a collection of transactions, then T_{new} is more likely to abort with transactions in group i.

In addition, as with the balanced workload above, the balanced clustering scheduler (algorithm 2) tracks the response time of each queue. The *Balanced Cluster Assign* scheduling algorithm performs O(k) model instance evaluations for each new transaction that is scheduled.

Algorithm 2 BalancedClusterAssign(T_{new})

```
min_dist ← $\infty$
idx ← $-1$
**for** $i := 0$ to $n - 1$ **do**
    dist ← $D(V_{new}, C_i)$
    **if** dist < min_dist **then**
        min_dist ← dist
        idx ← $i$
    **end if**
**end for**
rt ← $GetResTimeHistory(q_{idx})$
rt_avg ← $GetResTimeAvg(rt)$
rt_std ← $GetResTimeStd(rt)$
**if** rt - rt_avg > rt_std **then**
    min_rt ← $\infty$
    **for** $i := 0$ to $n - 1$ **do**
        rt_i ← $GetResTimeHistory(q_i)$
        **if** rt_i < min_rt **then**
            min_rt ← rt_i
            idx ← $i$
        **end if**
    **end for**
**end if**
$Enqueue(q_{idx}, T_{new})$
```

4.3 Unsupervised Scheduler Summary

In summary, the transactions assignment execution time of Balanced Cluster Assign is $O(kf)$ where k is the number of queues, f is the number of 1 features, plus some additional cost for detecting imbalances in response times. The centroids are static and thus do not have contention issues with multiple schedulers.

5 EXPERIMENTAL FRAMEWORK

The experimental framework focuses on isolating external factors from the system to highlight any differences in the policies. Experiments are run on a server that is otherwise idle. The server used for experiment contains 20 cores from Intel Xeon Silver 4114 CPU @ 2.20GHz with 125GB of memory. Hyper-threading technology is not used. One thread is pinned to work-flow computations exclusively. The remaining 19 worker threads both assign and execute transactions. In this framework, each worker thread is a scheduler and a transaction executor and the thread is dedicated to one queue of transactions. This arrangement allows us to measure the cost of scheduling algorithms but potentially introduces high contention between schedulers. The experimental design is an open queue environment where the arrival rate is used to simulate real world scenario where transactions arrive in DBMS at certain rate. Given an arrival rate, each worker thread creates and assigns enough transactions to meet the arrival rate requirements.

All experiments are run on the Peloton system, an open source, stored procedure, main-memory database system.

Experimental Execution

1. Initialize the random number generator to a new unique seed.
2. Initialize and start the database for the benchmark.
3. Execute transactions for 20 seconds and schedule transactions by random assignment to warm up the system.
4. Collect transactions execution results into log file during warm-up phase in Phase I.
5. Learn a model from a sample of the log file.
6. Turn on measurements, then schedule and execute transactions for 300 seconds under a given policy.
7. Wait for all transactions to finish execution.
8. Stop the database.

We repeat the above three phases 3 times and report the average of measurements across the 3 experiment runs.

6 RESULTS AND DISCUSSION

We compare four schedulers (Random, BFS, Balanced Vector and Balanced k-Means Scheduler) over three benchmarks (TPC-C, TATP and Epinions). The Random Scheduler serves as the baseline for comparison. Experimentally, we step-wise increase the transaction arrival rate to the point where the maximum execution speed of the system is equal to the arrival rate, and examine the transaction throughput, response time, transactions distribution, scheduling cost and workload balance over threads. We also analyze the underlying factors in success and failure of different scheduling algorithms.

We examine the response time and throughput of all five schedulers by gradually increasing the arrival rate until the system reaches or exceeds its maximum execution speed (fig. 2). The fact that throughput becomes worse when the arrival rate is too high is because every worker thread is also a scheduler and it is forced to assign enough transactions to meet arrival rate requirement. If the arrival rate is too high, then more work is put on transaction assignment instead of execution.

The Balanced k-Means scheduler is the winner of these five algorithms. It has a higher maximum throughput and at a given arrival rate, k-Mean scheduler cuts the response time by about 10%. See [15] for additional results and discussion.

7 RELATED WORK

Several recent works improve the performance of locking system in main memory databases [13, 18]. This line of work is complementary to our approach. Overall our work is most closely related to the general area of self driving databases [10].

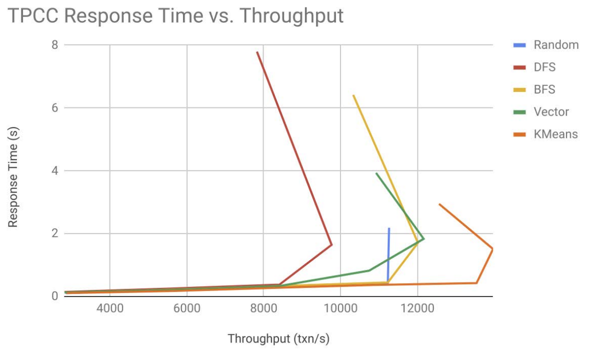
Partitioning Data. One technique to deal with contention partitions the data to reduce aborts [16, 17, 21]. Adopting partitioning as a core assumption, along with several additional design decisions, improves performance. However, performance problems still remain[23] and partitioning is problematic because some database schemas are not easily partitionable [3, 11, 12, 18]. In an informal sense, partitioning is data centric since the partitioning is generally based on the actual read/write sets of transactions. In contrast our work is transaction centric since it focuses on the read/write set predicted from the transaction statement without direct reference to the data.

Another OLTP architectural technique to avoid aborts focuses on careful implementation of ACID transaction properties to achieve improved performance [6, 19, 24]. Systems based on this approach offer improved performance over all database schemas and do not suffer from the constraints of partitioning. However, these systems still suffer under high contention workloads.

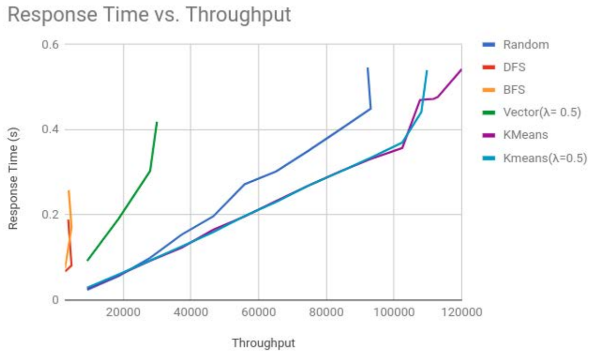
Semantic analysis. A deterministic approach to the analysis of code is followed in [7]. The approach merges multiple database calls in an application code loop back into a single SQL query. The approach identifies dependencies between database calls and extracts the independent statements to be run in parallel to decrease latency. A similar approach, extended to use optimization, is followed in QURO [22], to reorder statements to reduce transaction conflict under a two-phase locking protocol. However, reordering requires complex analysis for query dependency.

Deterministic analysis of transactions to detect aborts [1, 17] has the advantage of deriving deeper precise knowledge of the transactions structure at a higher cost of analysis. In contrast, we choose a lightweight, imprecise analysis to detect evidence related to transaction aborts.

TPC-C:



TATP:



Epinions:

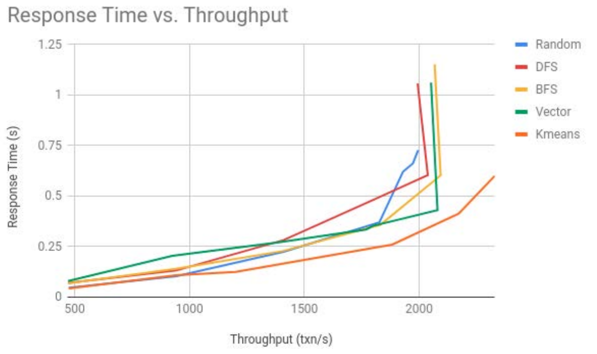


Figure 2: Response Time vs. Throughput – Each point in the graph represents an experiment run that generates a (throughput, response time) point. Lines connect the same scheduler as arrival rate increases. The spikes in the graph correspond to the arrival rate exceeding the processing rate of the system for the given scheduler. The "backwards bend" of a line occurs when the system becomes overloaded as the arrival rate swamps the system.

Zhang et al. [25] explores scheduling in main memory databases by considering different lightweight analysis of transactions and using a static representation for scheduling. We borrowed the notion of canonical representations of attributes from this work. In contrast, we use machine learning algorithms to learn distributions of aborts. In effect our results show that a machine learning algorithm can learn distributions to produce more intelligent and better performing schedulers.

In [4], the performance impact of scheduling based on the size of static HTTP requests to a server is investigated. The paper improves server response time by scheduling draining network socket buffers according to Shortest Remaining Processing Time queuing policy instead of Round Robin or random. Priorities on the queues are set by the number of bytes left to read from a static file. The paper and our work both use scheduling to improve performance based on a feature that characterizes the work remaining to be done for a an operation. The main difference is that the feature is a property of the transaction in the paper, where as in our work the feature is a signal generated by the transaction processing system.

Ic3 [14, 20] constructs a dependency graph and maintains the dependency graph for running transactions to provide efficient serializability. 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.

Scheduling One area of application program analysis focuses on improvement performance (network latency in particular) through heuristic program analysis [7]. The work is complementary to ours since it is focused on transformation of the application to improve performance as opposed to our scheduling approach.

Data and functionality partitioning. In H-Store [16], a stored procedure main memory database system, the database is chopped into partitions and a thread controls a partition at any point in time. The transactions execute serially for a given partition. HyPer [5] also processes transactions using a single-thread. It supports both OLTP and OLAP workloads (the latest version of Hyper uses optimistic multi-versioning [8]). In DORA [9], a transaction is split into sub-transactions and a sub-transaction is assigned to a single partition to execute.

Cheung et al. propose a technique of automatic code partitioning using program analysis [2]. By code partitioning they mean the partitioning of the application program between the application and stored procedures on the DBMS. The approach performs a program analysis of the application and then chops up the program into pieces with respect to network latency. Some pieces are executed on the application server and other pieces are executed on the database server. In effect, store procedures are automatically defined through program analysis.

8 CONCLUSIONS

In this paper, we conjectured that (i) the patterns of transaction aborts could be learned through the use of machine learning models, and (ii) those models could be used to improve the scheduling of transactions to reduce aborts and increase throughput. The paper systematically explored both supervised and unsupervised models for machine learning to create intelligent schedulers. In summary, the paper provides preliminary evidence that improved performance via intelligent scheduling is indeed possible in main memory database systems.

REFERENCES

[1] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[2] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C. Myers. 2012. Automatic Partitioning of Database Applications. *Proc. VLDB Endow.* 5, 11 (July 2012), 1471–1482. https://doi.org/10.14778/2350229.2350262

[3] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 48–57. https://doi.org/10.14778/1920841.1920853

[4] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. 2003. Size-based Scheduling to Improve Web Performance. *ACM Trans. Comput. Syst.* 21, 2 (May 2003), 207–233. https://doi.org/10.1145/762483.762486

[5] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, Washington, DC, USA, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[6] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. *Ratio* 10, 3 (2016), 10–2.

[7] Amit Manjhi, Charles Garrod, Bruce M. Maggs, Todd C. Mowry, and Anthony Tomasic. 2009. Holistic Query Transformations for Dynamic Web Applications. In *Proceedings of the IEEE 25th International Conference on Data Engineering*.

[8] Thomas Neumann, Tobias Muhlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 677–689. https://doi.org/10.1145/2723372.2749436

[9] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 928–939. https://doi.org/10.14778/1920841.1920959

[10] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*.

[11] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2213836.2213844

[12] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2018. Improving High Contention OLTP Performance via Transaction Scheduling. https://arxiv.org/abs/1810.01997v1

[13] Kun Ren, Alexander Thomson, and Daniel J Abadi. 2012. Lightweight locking for main memory database systems. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 145–156.

[14] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst.* 20, 3 (Sept. 1995), 325–363. https://doi.org/10.1145/211414.211427

[15] Yangjun Sheng, Tyieying Zhang, Andrew Pavlo, and Anthony Tomasic. [n. d.]. Scheduling OLTP Transactions via Machine Learning. http://arxiv.org/abs/1903.02990

[16] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. 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)*. VLDB Endowment, 1150–1160. http://dl.acm.org/citation.cfm?id=1325851.1325981

[17] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[18] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. 2018. Contention-aware lock scheduling for transactional databases. *Proceedings of the VLDB Endowment* 11, 5 (2018), 648–662.

[19] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[20] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1643–1658. https://doi.org/10.1145/2882903.2882934

[21] Jay-Louise Weldon. 2013. *Data base administration*. Springer Science & Business Media.

[22] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment* 9, 5 (2016), 444–455.

[23] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[24] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, Vol. 8. VLDB Endowment, 209–220.

[25] Tieying Zhang, Anthony Tomasic, Yangjun Sheng, and Andrew Pavlo. 2018. Performance of OLTP via Intelligent Scheduling. In *34th IEEE International Conference on Data Engineering (ICDE '18)*.