Scheduling Online Repartitioning in OLTP Systems

Kaiji Chen
University of Southern Denmark
chen@imada.sdu.dk

Yongluan Zhou
University of Southern Denmark
zhou@imada.sdu.dk

Yu Cao
EMC Labs China
EMC Corporation
yu.cao@emc.com

ABSTRACT

Previous studies on automatic database partitioning mostly focus on optimizing the (re)partitioning scheme for a given database and its query workload, while overseeing the problem about how to efficiently deploy the partition scheme onto the database system, which is, however, often non-trivial and challenging, especially in a distributed OLTP system where repartitioning is expected to take place online without interfering the user transactions. In this paper, we propose SOAP, a system framework for scheduling online database repartitioning for OLTP workloads. SOAP aims to minimize the time frame of executing the repartition operations while guaranteeing the correctness and performance of user transactions. It models and groups the repartition operations into repartition transactions, and then mixes them with the normal transactions for holistic scheduling optimization. SOAP utilizes a cost-based approach to prioritize the repartition transactions, and leverages a feedback model in control theory to determine in which order and at which frequency the repartition transactions should be scheduled for execution. When the system is under heavy workloads, selected repartition operations would piggyback onto the normal transactions to mitigate the repartitioning overhead. We have built a SOAP prototype on top of PostgreSQL and running at Amazon EC2, and conducted a comprehensive experimental study validating SOAP's significant performance advantages.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Distributed Databases and Transaction Processing

General Terms

Algorithms, Design, Performance, Experimentation

Middleware'14:Industry short papers, December 08-12 2014, Bordeaux, France
ACM 978-1-4503-3219-4/14/12 ...$15.00.
http://dx.doi.org/10.1145/2676727.2676731.

Keywords

Distributed Database Systems, Online Data Partitioning, Transaction Scheduling

1. INTRODUCTION

Automatic database partitioning has been extensively researched in the literature. Several studies have been done to make OLTP system scaling when incoming workload or data changed during runtime [2, 1, 3]. Furthermore, most DBMSs offer advisory tools for database partitioning design. The idea of these tools analyze the workload at a given time and suggest a (near-)optimal repartition scheme in a cost-based or policy-based manner, with the expectation that the system performance can thereby be maintained at a consistently high level. It is then the DBA's responsibility to deploy the derived repartition scheme onto the underlying database system, which however often posts great challenges to the DBA. On one hand, the repartitioning operations should be executed fast enough so that the new partition scheme can start to take effect as soon as possible. However, granting high execution priorities to the repartitioning operations will inevitably slow down or even stall the processing of normal transactions in the database system. On the other hand, the repartitioning procedure should be as transparent to the users as possible. In other words, the normal transactions' correctness must not be violated and the processing performance should not be significantly affected. Obviously, even skilled DBAs may not be able to easily figure out the best way to deploy repartition schemes, especially when the workload changes over time with bursts and peaks. As a result, automatic partition scheme deployment satisfying the above requirements is highly desirable. Surprisingly, few previous researches, if any, have been devoted to this important research problem.

In this paper, we focus on the problem about how to optimally execute the database repartition decisions (i.e. a set of repartition operations), either derived from the automatic partition design tools or from a DBA, in a distributed OLTP system where the database repartitioning is expected to take place online without interrupting and disrupting the normal transaction processing. We propose SOAP, a system framework for scheduling online fine-grained database repartitioning for OLTP workloads. SOAP aims at minimizing the time frame of executing the repartition operations while guaranteeing the correctness and performance of the concurrent normal transaction processing.

SOAP models and groups the repartition operations into repartition transactions, and then mixes them with the nor-

mal transactions for holistic scheduling optimization. There are two basic strategies for SOAP to use to schedule the repartition transactions. The first strategy is to maximize the speed of applying the repartitioning plan and submit all the repartition transactions to the waiting queue with a priority higher than the normal transactions. The second strategy schedules repartition transactions only when the system is idle. Both of the basic strategies lack the flexibility to find a good trade-off between the two contradicting objectives, i.e. maximizing the speed of executing repartition transactions and minimizing the interferences to the processing of normal transactions. To achieve this flexibility, SOAP interleaves the repartition transactions with normal transactions, and leverages a feedback model in control theory to determine in which order and at which frequency the repartition transactions should be scheduled for execution.

In the feedback-based method the repartition transactions have the same priority as the normal ones, hence they will contend with normal transactions for the locks of database objects and significantly increase the system's workload. To mitigate this issue, SOAP utilizes a piggyback-based approach, which injects repartition operations into the normal transactions that access the same tuples. As these transactions would acquire the locks of these tuples anyway, the overhead of acquiring and releasing locks as well as performing the distributed commit protocols can be saved. Moreover, the degree of lock contention can be reduced as the number of transactions decreases.

To summarize, we make the following significant contributions with this work:

- To the best of our knowledge, we are among the first to specifically study the problem of online deploying database partition schemes in OLTP systems.
- We propose a feedback model which realizes dynamic scheduling of the repartition operations.
- We also propose piggyback executing selected repartition operations within the normal transactions so as to further mitigate the repartitioning overhead.
- We have built a SOAP prototype on top of PostgreSQL, and conducted a comprehensive experimental study on Amazon EC2 that validates SOAP's significant performance advantages.

The rest of this paper is organized as follows. In Section 2, we describe the generic SOAP system architecture In Section 3, we elaborate SOAP's feedback-based and piggyback-based approaches of online scheduling repartition operations. Section 4 presents the experiment set-up and experimental results of a SOAP prototype on an Amazon EC2 cluster. We conclude in Section 5.

2. SOAP SYSTEM OVERVIEW

In this section, we describe the generic SOAP system architecture, as well as how SOAP realizes online fine-grained database repartitioning for OLTP workloads.

2.1 SOAP System Architecture

Figure 1 shows a SOAP-enabled distributed database architecture providing OLTP services. The clients submit user transactions through a *transaction manager (TM)*, which

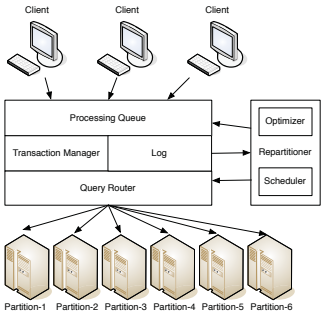


Figure 1: The generic SOAP system architecture

can be either centralized or distributed. TM takes care of the processing lifecycle of transactions, and guarantees their ACID properties with certain distributed commitment protocol and concurrency control protocol. The *query router* maintains the mappings between data partitions and their resident nodes, with which it routes the incoming transaction queries to the correct nodes for execution. All the submitted transactions will be associated with a scheduling priority and then put into a *processing queue*, where higher-priority transaction will be executed first, while the FIFO policy will be applied to a priority tie. The priority setting rules are customizable.

The transaction manager, query router and processing queue are common components in most OLTP systems, while SOAP introduces a new component *repartitioner* to coordinate its online fine-grained database repartitioning for OLTP workloads. In the following subsection, we describe how the repartitioner works.

2.2 Online Database Repartitioning

In this paper, we consider the scenarios where the transaction type and frequency of OLTP workloads will change over time, so that periodic database repartitioning is required in order to maintain the system performance.

The repartitioner determines when and how the OLTP database should be repartitioned. The optimizer component of it periodically extracts transaction frequency and tuple visiting graph from the workload history, and estimates the system throughput and latency in near future based on historical log data. If the estimated system performance is under a predefined threshold, the optimizer will derive a repartition plan in a cost-based manner. The repartition plan is at the granularity of tuples. We assume each tuple is associated with a globally unique identification key. We also assume tuple replication will be only for the purpose of high availability, yet make no assumptions about the replication strategy utilized. Tuple replicas will be distributed over distinct data partitions, and the query router will determine for a transaction which replica of a specific tuple should be visited.

The optimizer will generate three types of repartition operations together with its potential cost reduction to each of the normal transactions that could benefit from the new tuple partition plan of it, i.e. *new replica creation*, *replica deletion* and *tuple migration*.

- *New Replica Insertion*: insert a new replica of a specific tuple into a data partition containing no other replicas of the same tuple.

- *Replica Deletion*: for a tuple with multiple replicas, delete one specific replica.
- *Tuple Migration*: relocate a tuple between two partitions; the procedure is realized by first inserting a new replica of this tuple into the destination partition and then deleting the original replica from the source partition.

To perform the repartition operations, the scheduler packages them into repartition transactions using the information provided by optimizer. The repartition transaction will be scheduled by the repartitioner and submit the system at a chosen time. It utilizes a cost-based approach to set the repartition transactions' execution orders, and leverages a feedback model in control theory to determine at which frequency the repartition transactions should be scheduled for execution. As such, the processing of repartition transactions and normal transactions may be interleaved. In other words, during the database repartitioning, the normal transaction processing will keep going on, and an online repartition transaction scheduling algorithm, which will be elaborated in Section 3, attempts to minimize the time frame of executing the repartition operations while guaranteeing the correctness and performance of the concurrent normal transaction processing.

The repartitioner accesses the system logs, manipulates the processing queue and updates routing rules in the query router during and after the database repartitioning. With the piggyback-based execution method (refer to Section 3 for details), the repartitioner may need to modify the normal transactions by inserting additional repartition operations, and the transaction manager will coordinate the processing of the modified normal transactions.

3. ONLINE REPARTITION SCHEDULING

The scheduling of repartitioning operations has to be done in an online fashion. Besides the incoming workload is hard to predict, there are many system factors that will cause the system performance to fluctuate over time, such as variations of network speeds/bandwidth, transaction failures, interferences from other programs running on the servers. Therefore, we study how to implement an online scheduler that can continuously adapt to the system's current actual workload and performance.

3.1 Generating and Ranking Repartition Transactions

For each repartition operation, the scheduler will take a list of the potential cost reduction for each normal transaction that will benefit from the new tuple position. This list is obtained from the repartition optimizer. By collecting all such information for every repartition operation we need to execute, we will generate an inverted index for each affected normal transactions and sort it by the total cost reduction of all the repartition operations in its index entry. Repartition transactions will be generated by simply grouping all the index entry of each normal transaction in the potential cost reduction descending order. We will use the generated repartition transactions as our schedule unit in the remainder of this section.

In all subsequent scheduling algorithms, we have to first decide the execution order of repartition transactions and schedule the more "beneficial" ones before those less "beneficial". To do so, we need to estimate the cost and benefit executing such transactions. To estimate the cost of a transaction with different partitioning plans, we follow the approach in [1]. Suppose the cost of running transaction T_i with a partition plan where all the tuples accessed by T_i are collocated in a single partition is C_i, then the cost of T_i with a plan where T_i has to access more than one partition is $2C_i$.

With regard to the benefit of a repartition transaction, most fine-grained data partitioning algorithms, such as [1, 2, 3], have a cost model to estimate it. Suppose the cost of an arbitrary normal transaction T_i's with partition plan $\mathcal{P}$ is $C_i(\mathcal{P})$, then the benefit of a repartition transaction T_j, denoted as B_j can be defined as $\Sigma_{\forall T_i} f_i(C_i(\mathcal{O}) - C_i(\mathcal{P}))$, where f_i is the frequency of T_i. Finally, we can define the benefit density of T_j as B_j/C_j and then we can schedule the repartition transactions in descending order of their benefit densities.

3.2 Basic Solution

In general, there is a tension between the two objectives in our scheduling: (1) executing the repartitioning queries as soon as possible to improve the current partitioning plan, (2) avoiding interferences to the normal transactions and making the repartition process transparent to end users. In this subsection, we propose two baseline solutions, each favoring one of the objectives.

Apply-All. This strategy is to maximize the speed of applying the repartitioning plan and submits all the repartitioning transactions to the waiting queue with a priority higher than the normal transactions. As mentioned earlier, the system will schedule the transactions in descending order of their priorities, this strategy is equivalent to pausing the processing of normal transactions and performing the repartitioning queries immediately. Depending on the number of repartitioning transactions, the normal transactions may need to wait for a rather long time, which is usually unacceptable.

After-All. To minimize the interference in normal transactions, we can use a lazy strategy where repartitioning transaction will only be scheduled when the system is idle. We can achieve this by giving all the repartitioning transactions a priority lower than the normal ones. By doing so, the normal transactions will almost not be affected by the repartitioning transactions and the repartitioning could be done transparently. Due the this advantage, the state-of-the-art approach for online repartitioning adopted this strategy [3]. However, there is a downside of this approach: the repartition may be performed too slowly, especially when the system workload is high and there is very little idle time. Under this situation, the high workload could actually be alleviated by adopting the new (and better) partitioning plan and this strategy fails to take advantage of this.

3.3 Feedback-based Approach

As discussed earlier, the aforementioned basic solutions lack the flexibility to find a good trade-off between the two contradicting objectives. To achieve that, one can schedule some additional repartition transactions on top of those scheduled by the After-All strategy. These additional transactions will be assigned with the same priority as the normal transactions so that they have the chance to be executed faster. We call such transactions as high priority repartition transactions to distinguish them with those low priority ones scheduled by the After-All strategy. To limit the im-

pact over the normal transactions, we can limit the number of high priority repartitioning transactions.

However, such a seemingly simple idea is rather challenging to realized in practice. Note that the number of high priority repartition transactions that we can execute without significant disturbance of the normal transactions heavily depends on the system's current workload and capacity. In reality, the system's workload may have temporary skewness and fluctuations even if it appears to be uniformly distributed for some long period [2]. Furthermore, the system's capacity is also subject to variations caused by external factors, such as external workload imposed on the same server or other virtual servers running on the same physical machine or cluster rack in a cloud computing environment. A desirable solution should be able to detect such short-term variations of system workload and capacity and promptly adapt the scheduling strategy accordingly. To achieve this goal, we model our system as an automatic control system and take use of the feedback control concept in control theory to design an adaptive scheduling strategy.

Control theory deals with the behaviors of complex dynamic systems with inputs and present output values. A controller is engineered to generate proper corrective actions so that system error, i.e. the differences between the desired output value, called setpoint (SP), and the actual measured output value, called process variable (PV), are minimized.

A commonly used controller is the Proportional-Integral-Derivative controller (PID controller). Let $u(t)$ be the output of the controller, then the PID controller can be defined as follows:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \qquad (1)$$

where K_p, K_i and K_d are the proportional, integral and derivative gains respectively, and $e(t)$ is the system error at time t. The system error can be minimized by tuning the three gains so that the controller will generate proper outputs. Simply put, K_p, K_i and K_d determines how the present error ($e(t)$), the accumulation of past errors ($\int_0^t e(\tau)d\tau$) and the predicted future error ($\frac{d}{dt}e(t)$) would affect the controller's output.

The system for scheduling repartition transactions can be modeled as a PID controller as follows. We can use the ratio of the total cost of the high-priority repartition transactions to that of the normal transactions as the SP for the PID controller. By stabilizing this ratio, we can constrain the total workload imposed by the high priority repartition transactions at a desired level so that they would have limited impact over the latency of the normal transactions and in the mean time maximize the speed of applying the repartitioning plan.

To capture the fluctuations of the system's workload and capacity, we divide the time into small intervals and measure the aforementioned ratio for every interval. The actual ratio that is measured would be the PV of the PID controller and hence the error can be computed as $SP - PV$. The output of the controller is the ratio to be used to calculate the number of high-priority repartition transactions that we should schedule in the coming interval.

To tune the parameters of the PID controller, we will take an online heuristic-based tuning method formally known as the Ziegler—Nichols method[5].

Finally, we enforce a limit on the maximum number of high-priority repartition transactions scheduled in each time interval to avoid significant impacts caused by sudden changes of system workload and capacity, which the PID controller will take some time to stabilize its outputs. Putting such a limit is essentially a conservative approach to avoid too much interferences during the period that the PID controller is stabilizing its behavior.

3.4 Piggyback-based Approach

In the feedback-based method, the repartition transactions have the same priority as the normal transactions, hence they will content with normal transactions for the locks of database objects and significantly increase the system's workload. We propose a piggyback-based approach, which injects repartition operations into the normal transactions that access the same tuples. As these transactions would acquire the locks of these tuples anyway, we can save the overhead of acquiring and releasing locks as well as performing the distributed commit protocols. Moreover, we can reduce the degree of lock contention by reducing the number of transactions.

This algorithm will examine the tuples that are accessed by the incoming normal transaction and if some of them need to be relocated, it will piggyback the repartitioning operations into the transaction as extra data modification queries. This essentially leads to a repartition-on-demand strategy where tuples will be repartitioned only when they are going to be accessed.

One important issue of the piggyback-based approach is that the extra queries and processing time would incur a higher failure rate of the transactions caused by the lock contentions. The failed transactions will be rolled back by the DBMS and they have to be resubmitted for processing. Such failures would often occur when the system has a high overall workload within some short period of time. To avoid repeated transaction failures and further congestions of the system, upon a failure of the piggybacked transaction, we will only resubmit the original version of the transaction.

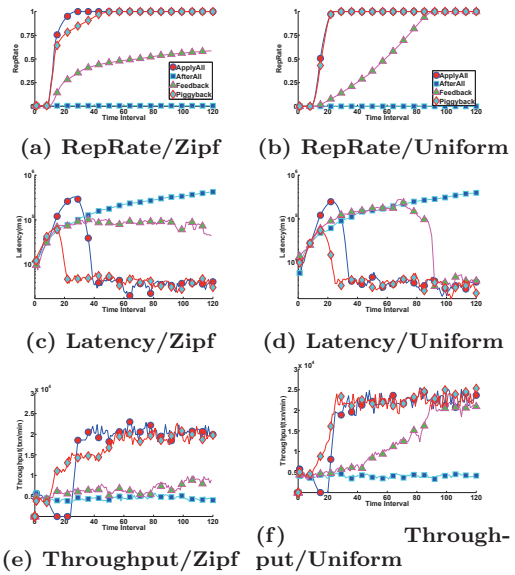
4. EVALUATION

In this section, we will first provide some details of our system implementation and experiment configuration in Section 4.1. The experimental results under different workload conditions are presented and discussed in Section 4.2 and Section 4.3.

4.1 Experimental Configuration

System Implementation and Configuration. We have used PostgreSQL 9.2.4 as the local DBMS system at each data node and JavaSE-1.6 platform for developing and testing our algorithms. We have developed a query router using a lookup table to route each query to its target tuples. We have also implemented a query parser that could read the query and extracted the globally unique identifier of the target tuples, which will be used for query routing and applying our online repartition strategies. For transaction management, we take use of Bitronix[4], an implementation of Java Transaction API 1.1 version adopting the XAResource interface to interact with the DBMS resource managers running on the individual data nodes and using 2-Phase Commit (2PC) protocol.

Our evaluation platform is deployed on a Amazon EC2 cluster consisting of 5 data nodes corresponding to 5 data



(a) RepRate/Zipf (b) RepRate/Uniform

(c) Latency/Zipf (d) Latency/Uniform

(e) Throughput/Zipf (f) Throughput/Uniform

Figure 2: Experiment Results for High Workload

partitions respectively. Each data node runs an instance of a PostgreSQL 9.2.4 server, which is configured to use the *read committed* isolation level and have a limitation of 100 simultaneous connections. The node configuration consists of 1 vCPU using Intel Xeon E5-2670 processor and 3.75 GB memory with an on-demand SSD local storage ruuning 64-bit Ubuntu 13.04. The query router is run on an another EC2 instance with the same setting.

Workloads and Datasets. We create a table containing 500000 tuples. We generate two types of workload distribution to simulate two different scenarios: a uniform distribution with 30000 distinct transactions and a Zipf distribution with 23457 distinct transactions. We generate the workload with a Zipf distribution using the parameter $s = 1.16$ so that the workload follows the 80-20 rule. Each normal transaction will contain 5 queries. Each query access one unique tuple and is either a read-only or a write query with equal probability.

We use a Poisson distribution to determine how many normal transactions to submit to the system for each interval, which is set to be 20 seconds. Each run of the experiment will last for 45 minutes and the normal transactions are submitted to the system by the distribution at the beginning of each time interval. We generate a high and a low workload as follows. *Lowload* has an average system utilization as 65% before the repartitioning, which is measured by the percentage of time that the system spend on processing the normal transactions. *Highload* simulates a system overloaded situation, where the incoming workload is 30% higher than the system capacity. Under this situation, it is more urgent to adopt the repartitioning plan to reduce the effective incoming load. Furthermore, for each situation, we set the percentage of tuples we need to repartition as 60%.

Algorithm Settings. We compare all the algorithms we discussed in the previous sections. We use three performance metrics for comparison: the system *Throughput*, which is counted as the maximum number of normal transactions that the system can process per unit of time; the processing *Latency*, which is the time between a transaction is submitted and the time its processing is finished; the *RepRate*, which is the percentage of repartition operations that have been performed so far. In line with the workload generation, we divide the time into 20 seconds intervals and run the system for 10 intervals to warm it up before we start the repartitioning. Furthermore, the feedback-based approach uses 20 seconds as the monitoring interval.

For the feedback model parameter used in each of the experiment, we use $SP = 1.05$ for Zipf *HighLoad* and $SP = 1.25$ for Uniform *Highload* workload, $SP = 1.03$ for all the *LowLoad* workload. All the experiments will have a same controller parameter $K_p = 1$, $K_i = 0$ and $K_d = 0$.

4.2 Performance Under High Load

Recall that under the high workload setting, we have set the initial workload to be higher than the system's capacity but it should become lower than the system's capacity after applying the repartition plan as the normal transactions would consume less resources with the new partition plan. Therefore, it is necessary for the system to be able to process the repartition transactions as soon as possible.

The experimental results are presented in Figure 2. As we discussed earlier, ApplyAll would stall all the normal transactions and execute all the repartition transactions before we resume the normal processing. This should result in the fastest deployment of the new partition plan. This is verified by Figure 2a. However as one can see from Figure 2e and Figure 2c, using this approach will experience a period that the system has a very low throughput and very high processing latency caused by the stalling of the normal transactions.

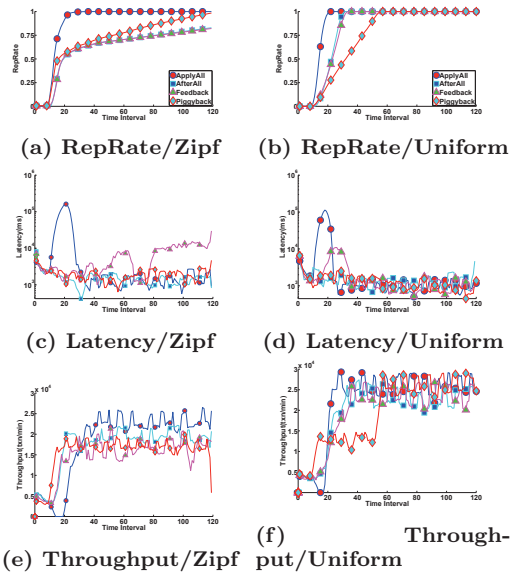
On the contrary, AfterAll basically cannot execute any repartition transactions due to the lack of system idle time, hence it cannot take advantage of the new data partition plan. The Feedback approach will enforce the scheduling of some repartition transactions, hence can make some progress in deploying the new partition plan (Figure 2a). Accordingly, the system's throughput and processing latency will improve gradually.

As we analyzed in the earlier sections, the Piggyback approach can effectively reduce the cost of executing repartition operations. This is especially important when the system is under high workload and has little extra resources for repartitioning the data. Furthermore, the high arrival rate of normal transactions provides abundant opportunities for the repartition operations to piggyback. The results in Figure 2 verify our analysis. In comparing to ApplyAll, Piggyback does not incur any sudden dropping of system performance while it is able to quickly execute the repartition plan.

We also perform the experiments with a workload under a uniform distribution. The results are presented in the right column of Figure 2. The difference from the workload with a Zipf distribution is that we will not gain a lot of improvement by executing a small portion of repartition transactions.

Similar to the previous experiments, since the workload is more than the system could handle, AfterAll could barely execute any repartition transactions improve the system's performance.

For the Feedback method, we set a higher SP value under uniform workload to examine its performance when more repartition transactions are enforced to be submitted to the system. The system finally finish the repartition in time and make the system able to process all the incoming nor-



(a) RepRate/Zipf (b) RepRate/Uniform

(c) Latency/Zipf (d) Latency/Uniform

(e) Throughput/Zipf (f) Throughput/Uniform

Figure 3: Experiment Results for Low Workload

mal transactions without queuing. In comparing the results in the previous experiments, a higher SP here is actually beneficial when the number of repartitioning transactions is relatively small and Feedback has the chance to finish them in a good time.

Similar to the Zipf workload, Piggyback performs the best in all cases. It can achieve a speed that is almost identical to the *ApplyAll* approach.

4.3 Performance Under Low Load

In the low load experiments, we expect the system has more idle time and the repartition process could be done more aggressively to take use of these available resources. For the Zipf workload, since there exist some transactions that have a higher frequency in the workload, the resource contention under the same load level will be higher than the Uniform workload.

ApplyAll performs similarly as under high load situation. But, since there are fewer normal transactions, there are also fewer transactions that are queued up during the repartitioning period and we will have a shorter time for the system to achieve its maximum performance after the repartitioning period.

As the system has enough idle time now, AfterAll could submit quite some repartition transactions now. AfterAll has the minimum interference to the normal transactions when the system is able to handle the normal transaction load like the situations in Figure 3c and hence it could be the algorithm that can achieve the lowest average latency.

Feedback will add more repartition transactions to the system than just filling up the idle period. So we can see in Figure 3c that it partitions the tuples accessed by some high frequency transactions and render the load decreasing more quickly than AfterAll. Adding the extra repartition transactions will increase the processing latency of the normal transactions. This extra latency is a trade-off against the repartitioning speed. Figure 3f and Figure 3d show that Feedback has both a higher throughput and a higher latency than AfterAll before it completes the repartitioning.

Piggyback is a passive repartitioning method and the overhead of this method is proportional to the number of piggybacked transactions. When the workload is very low, like the condition in Figure 3e, it may not be able to repartition the database as fast as in the high load situation. But the latency it incurs is also lower with a lower workload.

With the Uniform workload, the degree of resource contention among the normal transactions is lower than that with the Zipf workload. AfterAll could finish the repartitioning more quickly than with the Zipf workload. Since the frequency of each normal transaction is low, the repartition may take some time to get into effect. We could find in Figure 3f that Piggyback has a stair shape curve. This is because, the repartitioning operations that piggybacks on a transaction could only take effect when the same transaction is submitted to system again.

5. CONCLUSION

In this paper, we studied the problem of online repartitioning of a distributed OLTP database. We identify that the two basic solutions are very rigid and miss the opportunities to find good trade-offs between the speed of repartitioning and the impact on the normal transactions. We then propose to use control theory to design an adaptive methods which can dynamically change the frequency that we submit repartition transactions to the system. As putting the repartition queries into extra transactions may further increase the system's resource contention especially when the system has a high workload, we also proposed a piggyback-based method to mitigate the repartitioning overhead, which however do not perform well when the system has a low workload and there is few transactions to piggyback on. Our hybrid approach intelligently integrates the two approaches and is able to combine their strengths while avoiding their problems. Based on the experiments of running our prototype on Amazon EC2, we can conclude that Hybrid is the overall best approach and achieves a great performance improvement in comparing to the two basic solutions used in most existing systems.

6. REFERENCES

[1] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[2] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *SIGMOD Conference*, pages 61–72. ACM, 2012.

[3] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 430–441, New York, NY, USA, 2013. ACM.

[4] Brett Wooldridge. Bitronix jta transaction manager, 2013.

[5] JG Ziegler and NB Nichols. Optimum settings for automatic controllers. *trans. ASME*, 64(11), 1942.