Online Data Partitioning in Distributed Database 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

Most of previous studies on automatic database partitioning focus on deriving a (near-)optimal (re)partition scheme according to a specific pair of database and query workload and oversees the problem about how to efficiently deploy the derived partition scheme into the underlying database system. In fact, (re)partition scheme deployment is often non-trivial and challenging, especially in a distributed OLTP system where the repartitioning is expected to take place online without interrupting and disrupting the processing of normal 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 the concurrent processing of normal transactions. SOAP packages 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 rank the repartition transactions' scheduling priorities, 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 workload or resource shortage, SOAP takes a further step by allowing repartition operations to piggyback onto the normal transactions so as to mitigate the resource contention. We have built a prototype on top of PostgreSQL and conducted a comprehensive experimental study on Amazon EC2 to validate 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

Keywords

Distributed Database Systems, Online Data Partitioning, Transaction Scheduling

1. INTRODUCTION

The difficulty of scaling front-end applications is well known for DBMSs executing highly concurrent workloads. One approach to this problem employed by many Web-based companies is to partition the data and workload across a large number of commodity, shared-nothing servers using a cost-effective, distributed DBMS. The scalability of online transaction processing (OLTP) applications on these DBMSs depends on the existence of an optimal database design, which defines how an application's data and workload is partitioned across the nodes in a cluster, and how queries and transactions are routed to the nodes. This in turn determines the number of transactions that access data stored on each node and how skewed the load is across the cluster. Optimizing these two factors is critical to scaling complex systems: a growing fraction of distributed transactions and load skew can degrade performance by a factor of over ten. Hence, without a proper design, a DBMS will perform no better than a single-node system due to the overhead caused by blocking, inter-node communication, and load balancing issues.

Automatic database partitioning has been extensively researched in the past. As a consequence, nowadays, most DBMSs offer database partitioning design advisory tools. 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 always maintain a consistently high level. It is then the DBA's responsibility to deploy the derived repartition scheme into the underlying database system, which however often posts great challenges to the DBA. On the one hand, the repartition 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 normal transaction processing on the database system. On the other hand, the repartitioning procedure should be as transparent to the users as possible. In other words, the normal user transactions' correctness must not be violated and the processing performance should not be significantly influenced. Obviously, even skilled DBAs may not be able to easily figure out the best ways of deploying repartition schemes, especially when the workload changes over time

10.5441/002/edbt.2015.02

and has bursts and peaks. As a result, automatic partition scheme deployment satisfying the above requirements is highly desirable. Surprisingly, few previous studies have been devoted to this important research problem.

In this paper, we focus on the problem about how to optimally execute a database repartition plan consisting of a set of repartition operations in a distributed OLTP system, where the repartitioning is expected to take place online without interrupting and disrupting the normal transactions' processing. 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 the concurrent normal transaction processing.

SOAP models and groups the repartition operations into repartition transactions, and then mixes them with the normal transactions for holistic scheduling optimization. There are two basic strategies for SOAP to schedule the repartition transactions, which are similar to the techniques used in state-of-the-art database systems' online repartitioning solutions. The first strategy is to maximize the speed of applying the repartition 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 basic strategies lack the flexibility to find a good trade-off between the two contradicting objectives: maximizing the speed of executing repartition transactions and minimizing the interferences to the processing of normal transactions. As a result, SOAP interleaves the repartition transactions with normal transactions, and leverages feedback models 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 transactions, hence they will contend with normal transactions for the locks of database objects and significantly increase the system's workload, especially when the system is under heavy loads or resource shortage. To mitigate this issue, SOAP utilizes a piggyback-based approach, which injects repartition operations into the normal transactions. The overhead of acquiring and releasing locks as well as performing the distributed commit protocols incurred by a repartition transaction can be saved if the normal transaction that it piggybacks on will access the same set of database objects.

While the piggyback-based approach consumes less resources, it fails to take use of the available system resources to speed up the repartitioning process. This may leave some resources unused when the system workload is low and there are few transactions to piggyback on. Therefore, SOAP adopts a hybrid approach that is composed by a piggyback module and the feedback module. When the system workload does not use up all the system's resources, we can make use of the available resources to repartition the data before the actual arrival of transactions that will access them, meanwhile the piggyback-based approach will attempt to let the repartition transactions piggyback on the normal transactions when they arrive.

To summarize, we make the following 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 into OLTP systems.
- We propose a feedback model that realizes dynamic scheduling of the repartition operations.
- We also propose a piggyback approach to execute selected repartition operations within normal transactions 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, as well as how SOAP realizes online repartitioning for OLTP workloads. In Section 3, we elaborate SOAP's feedback-based, piggyback-based and hybrid 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 discuss the related works in Section 5 and then conclude in Section 6.

2. SOAP SYSTEM OVERVIEW

In this section, we describe the generic SOAP system architecture, as well as how SOAP realizes online 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 can be either centralized or distributed.Each submitted transaction will be given a global unique ID by the TM. TM takes care of the processing life-cycle of transactions and guarantees their ACID properties with certain distributed commit protocols and concurrency control protocols. The *query router* maintains the mappings between data partitions and their resident nodes, based on 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 transactions will be executed first, while the FIFO policy will be applied to break the tie. The rules of setting priorities 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 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 type of transactions and frequency of OLTP workloads could 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. Its optimizer component periodically extracts the frequency of transactions and their visiting data partitions from the workload history, and then estimates the system throughput and latency in the near

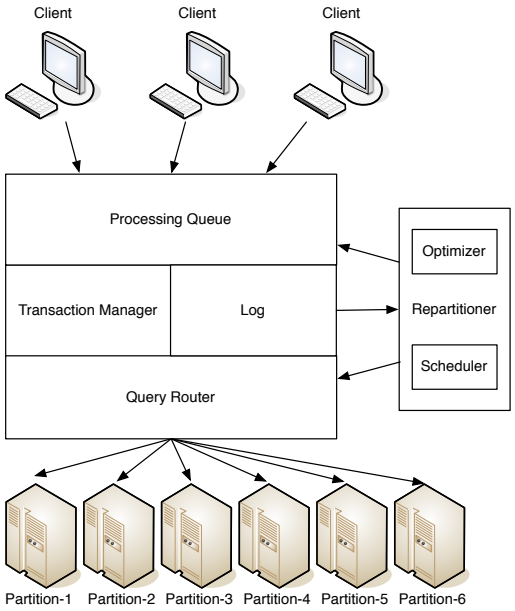


Figure 1: The generic SOAP system architecture

future based on the history. 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 could be at the granularity of moving individual tuple or tuples within some ranges or with some hash keys on their attributes. We assume each tuple contains enough information to be positioned by query router. We assume tuple replicas are only made 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 the normal transactions accessing the database objects repartitioned by each of them, i.e. *new replica creation*, *replica deletion* and *objects migration*.

- *New Replica Creation*: insert some new replicas of database objects originally stored in a data partition into an another one containing no other replicas of the same objects.
- *Replica Deletion*: for database objects with multiple replicas, delete the specific replica within one partition.
- *Objects Migration*: relocate some database objects between two partitions; the procedure is realized by first inserting new replicas of them into the destination partition and then deleting the original ones from the source partition.

To execute the repartition operations, the scheduler packages them into repartition transactions using the information provided by the optimizer. The repartition transaction will be scheduled by the repartitioner and submitted to the system at a chosen time. It utilizes a cost-based approach to determine 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 processing of normal transactions will keep going on, and an online scheduling algorithm, which will be elaborated in Section 3, attempts schedule repartition transaction so that the time frame of executing the repartition operations is minimized while guaranteeing the correctness and performance of the concurrent processing of normal transactions.

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

3. ONLINE REPARTITION SCHEDULING

The scheduling of repartition 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, and interferences from other programs running on the same server. Therefore, we study how to implement an online scheduler that can continuously adapt to the system's actual workload and capacity.

3.1 Generating and Ranking Repartition Transactions

In all the 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 achieve this, we need to estimate the cost and benefit of executing such transactions. To estimate the cost of a transaction under different partitioning plans, we follow the approach in [4]. Suppose the cost of running transaction T_i with a repartition 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 $2C_i$.

To estimate the benefit of a repartition transaction, we use the cost model of the data partitioning algorithms, such as [4, 13, 15]. Suppose the cost of an arbitrary normal transaction T_i 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.

To package the repartition operations into transactions, there are two simple options: (1) putting all operations into one transaction and (2) creating one transaction for each operation. The first option will create a very large transaction especially when there are a lot of data to be repartitioned. Such a large transaction will be run for a very long time and will significantly increase resource contention. For example, with a 2PL policy, the repartition transaction has to hold the locks of all the data objects involved in the repartition

plan until it is committed. This will substantially increase the degree of lock contention with the normal transactions. On the other hand, the second option will not suffer from this problem but it will create a lot of transactions each incurring overhead to the transaction manager. It is desirable to find a good trade-off between this two extremes. In principle, we would like to create small transactions and their overhead can be paid off by the benefit that it will bring to the system. As accurately predicting and quantifying the overhead and the degree of lock contentions that will be introduced by a repartition transaction is difficult, we adopt a simple heuristic here. Roughly speaking, we put the repartition operations that repartition all the objects accessed by a normal transaction into a repartition transaction. In this way, we can ensure that there is at least one normal transaction that will benefit from executing the repartition transaction. Provided that the achieved benefit is greater than the overhead of introducing the repartition transaction, we can ensure that the overhead will be paid off when the repartition transaction is executed. Furthermore, even with this heuristic, there are still many possible ways to combine the repartition operations into transactions. We prefer generating transactions that will have higher benefit densities and, as mentioned earlier, schedule them in descending order of their benefit densities.

Algorithm 1 shows the whole process for the scheduler to generate a ranked list of repartition transactions. Given a new partition plan $\mathcal{P}$ generated by a cost-based repartition optimizer, the scheduler will obtain a list of repartition operations OP_{rep} together with the list of normal transactions that will access the data objects modified by each operation. In line 1-8, we construct a map T_{OP} that maps the ID of a normal transaction t_i with frequency f_i to a group of repartition operations that will modify objects accessed by t_i. In other words, the performance of t_i will be affected by this list of repartition operations.

We then calculate the benefit of executing each repartition operation in lines 9-14. After that, we can calculate the total benefits of each group of repartition operations in T_{op} and store them in another map $T_{benefit}$ with value descending order. Finally, we transform each group of repartition operations into a repartition transaction, and make sure each repartition operation only belongs to one repartition transaction. These repartition transactions will be returned by the algorithm as the output. Furthermore, we calculate the benefit density of each repartition transaction and sort them in descending order. Given a repartition transaction r_i and a normal transaction t_i whose performance will be affected by r_i, T_{Rep} maps the ID of r_i to the ID of t_i and the benefit density of r_i. Such auxiliary information will be used in our subsequent scheduling algorithms.

Algorithm 1: Generating and Ranking Repartition Transactions

Data: a list of repartition operations OP_{rep} generated by optimizer, new partition plan $\mathcal{P}$
Result: a list of repartition transactions L_{Rep},a map T_{Rep} mapping repartition transaction id to a affected normal transaction id and the benefit density of the repartition transaction

1 Create HashMap T_{op}, a mapping from normal transaction to the repartition operations that edit the objects visited by it
2 **for** $op_k \in OP_{rep}$ **do**
3 **for** *Normal transaction* t_i *accessing the objects modified by* op_k **do**
4 **if** $C_i(\mathcal{O}) - C_i(\mathcal{P}) > 0$ **then**
5 Insert op_k to T_{op}.get(t_i)
6 **for** $t_i \in T_{op}.keylist$ **do**
7 benefit $\leftarrow f_i \frac{C_i(\mathcal{O}) - C_i(\mathcal{P})}{T_{op}.get(t_i).size()}$
8 **for** $op_k \in T_{op}.get(t_i)$ **do**
9 $op_k.benefit$ += $benefit$
10 Create HashMap $T_{benefit}$, a mapping from repartition operation group ID to the total benefit for system if all the operations within this group are executed
11 **for** (t_i,L_{op}) $\in T_{op}.entrySet$ **do**
12 benefit $\leftarrow$ 0; **for** $op_i \in L_{op}$ **do**
13 benefit += $op_i.benefit$;
14 Insert (t_i,benefit) to $T_{benefit}$;
15 Sort $T_{benefit}$ with value descending order
16 **for** (t_i,*benefit*) $\in T_{benefit}$ **do**
17 ops $\leftarrow T_{op}.get(t_i)$;
18 **for** $op_i \in ops$ **do**
19 **if** $op_i \notin OP_{rep}$ **then**
20 Remove op_i from ops; $T_{benefit}.get(t_i) \leftarrow T_{benefit}.get(t_i) - op_i.benefit$;
21 Remove ops from OP_{rep};
22 Create r_i with ops;
23 $c_i \leftarrow$ Cost(r_i,$\mathcal{O}$);
24 $cpr_i \leftarrow \frac{T_{benefit}.get(t_i)}{c_i}$;
25 Insert ((r_i, t_i),cpr_i) to T_{Rep};
26 Insert r_i to L_{Rep}
27 Sort T_{Rep} with value descending order;

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 the 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 repartition 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 and hence this strategy is equivalent to pausing the processing of normal transactions and performing the repartitioning queries immediately. Depending on the number of repartition transactions, the normal transactions may need to wait for a rather long time, which is usually unacceptable.

After-All. To minimize the interferences to normal transactions, we can use a lazy strategy where repartition transactions will only be scheduled when the system is idle. We can achieve this by giving all the repartition transactions a priority lower than the normal ones. By doing so, the normal transactions will almost not be affected by repartition transactions and the repartitioning could be done transparently. Due to this advantage, a state-of-the-art approach for online repartitioning adopted this strategy [15]. However, there is a downside of this approach: the repartitioning may be per-

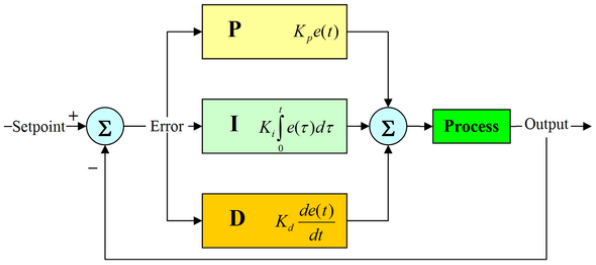


Figure 2: A sample PID controller block diagram

formed 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 those.

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 this, 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 impact over the normal transactions, we can limit the number of high priority repartition transactions.

However, such a seemingly simple idea is rather challenging to realize in practice. Note that the number 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 temporal skewness and fluctuations even if it appears to be uniformly distributed for a long period [13]. 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 make 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). Figure 2 depicts a graphical representation of a 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 desirable 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 take an online heuristic-based tuning method formally known as the Ziegler—Nichols method[19].

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 contend with normal transactions for the locks of database objects and significantly increase the system's workload.

In this section, we propose a piggyback-based approach, which injects repartition operations into the normal transactions that access the same object. As these transactions would acquire the locks of these objects 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.

The algorithm of this approach is illustrated in Algorithm 2. The algorithm make use of the auxiliary information produced in Algorithm 1. It examines the transaction ID t_i of the incoming normal transaction and check if there exist an repartition transaction r_j in T_{Rep} which t_i can benefit from but are not yet executed. If so, r_j will piggyback onto t_i, injecting the repartition operations in r_j to t_i. These

Algorithm	Workload	HighLoad			LowLoad		
		$\alpha = 100\%$	$\alpha = 60\%$	$\alpha = 20\%$	$\alpha = 100\%$	$\alpha = 60\%$	$\alpha = 20\%$
Feedback	Zipf	1.05	1.05	1.1	1.05	1.03	1.015
	Uniform	1.25	1.25	1.25	1.02	1.03	1.02
Hybrid	Zipf	1.05	1.05	1.05	1.05	1.03	1.05
	Uniform	1.05	1.05	1.05	1.03	1.05	1.05

Table 1: SP value for Experiments

Algorithm 2: Piggyback Algorithms for incoming normal transactions $T_i(k)$

Data: a list of repartition transactions L_{Rep}, a map T_{Rep} from repartition transaction id to an affected normal transaction and the benefit density of the repartition transaction, incoming normal transactions $T_i(k)$ in interval k, List of all the repartition operations OP_{Rep}
Result: Piggybacked normal transaction executed in interval k

```
1  Create a map P(k) of all the normal transactions
   piggyback some repartition operations in interval k
2  for t_i ∈ T_i(k) do
3      if r_j, t_i ∈ T_Rep.keylist then
4          ops ← L_Rep.get(r_j)
5          Insert ops to t_i
6          Inert (t_i, (r_j, t_i)) to P(k)
7  Submit T_i(k)
8  Get Result(k) for any finished transaction
9  for (t_i ∈ P(k)) do
10     if t_i ∈ Result(k) then
11         if t_i.success then
12             Remove P(k).get(t_i) from T_Rep
13         else
14             Remove L_Rep.get(P(k).get(t_i).getKey())
               from t_i
15             Resubmit t_i
```

repartition operations will share the locks of objects with the normal transactions. This essentially leads to a repartition-on-demand strategy where data will be repartitioned only when they are accessed. After the piggybacked transaction is successfully committed, we will remove the corresponding repartition transaction in T_{Rep}.

The piggyback method will increase the transaction failure rate as the execution times of normal transactions are increased. If too many repartition operations piggyback onto a normal transaction, then the system throughput will be decreased due to unnecessary aborts caused by the failure of the piggybacked repartition operations. Therefore, we need to limit the maximum number of repartition operations that can piggyback onto each normal transaction. This parameter should be tuned at runtime to adapt to the scenarios of different systems.

3.5 Hybrid Approach

While the piggyback-based approach consumes less resources, it fails to take use of the available system resources to speed up the repartitioning process. This may not work well when the system workload is low and there are few transactions to piggyback on. A more desirable approach is, when the system workload is low, we can take use of the available resources to repartition the data before the actual arrival of transactions that will access them, and when the system is a bit congested, we can take advantage of the opportunities to piggyback the repartition operations on the incoming transactions.

In this section, we present a hybrid approach that is composed by a piggyback module and the feedback module. The piggyback module will piggyback the repartition operations on the incoming normal transactions. Then for each interval, the feedback module will measure the actual PV value by counting in both the repartition transactions and those repartition operations piggybacked on the normal transactions. In this way, the feedback module will adapt the number of repartition transactions according to the actual workload of the system. In other words, when there are more incoming normal transactions that more repartition operations can piggyback on, we will reduce the number of repartition transactions and vice versa.

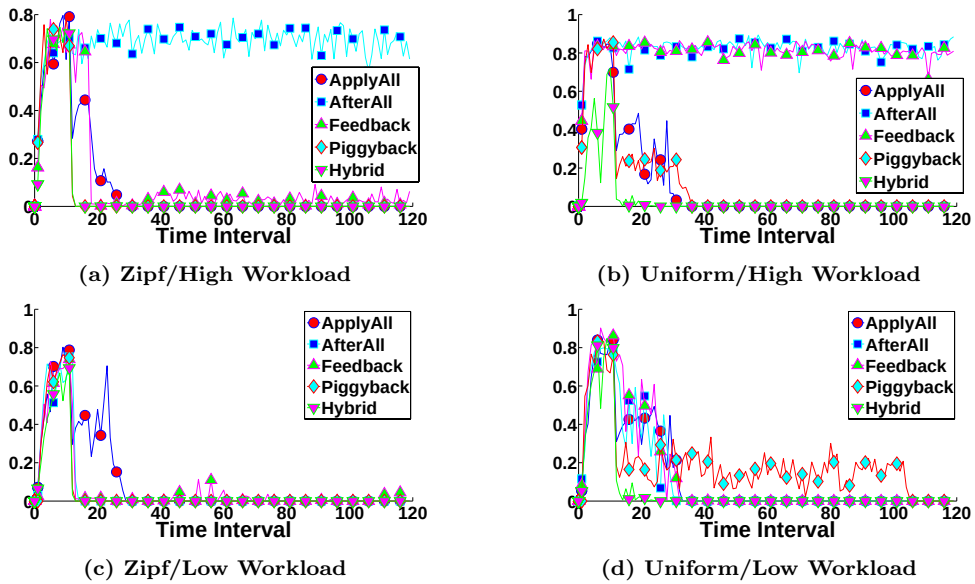
4. EVALUATION

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

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 database objects. We have also implemented a query parser that reads a query and extracts the partition attributes of the target objects, which will be used for query routing and applying our online repartition strategies. For transaction management, we take use of Bitronix[17], an implementation of Java Transaction API 1.1 version adopting the XAResource interface to interact with the DBMS resource managers running each the individual data node and using 2-Phase Commit protocol for distributed transaction commits.

Our evaluation platform is deployed on a Amazon EC2 cluster consisting of 5 data nodes corresponding to 5 data 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 has a limitation of 100 simultaneous connections. Note that higher isolation level will decrease the system concurrency and hence lower the system's capacity. But it will not affect the performance of our algorithms. 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 running 64-bit Ubuntu



(a) Zipf/High Workload

(b) Uniform/High Workload

(c) Zipf/Low Workload

(d) Uniform/Low Workload

Figure 3: Experiment Results for Transaction Failure Rate

13.04. The query router is run on another EC2 instance with the same setting.

Workloads and Datasets. We create a table containing 500,000 tuples, each tuple has a global unique key field and an integer content field. The size of each tuple is 8 bytes. We generate two types of workload distribution to simulate different sceneries of transaction popularity: a uniform distribution with 30,000 distinct transactions and a Zipf distribution with 23,457 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 contains 5 queries. Each query access one unique tuple and is either a read-only or a write query with equal possibility.

We use a Poisson distribution to determine how many normal transactions are submitted to the system during 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 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 vary the percentage of tuples α we need to repartition, which varies from 100% to 20%. After the repartitioning, α percent of the normal transactions would be tranformed from distributed transactions to non-distributed transactions.

Algorithm Settings. We compare all the algorithms we discussed in the previous sections. We use two 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, and the processing latency, which is the time between a transaction is submitted and the time its processing is finished. In order to examine the lock contentions incurred by the various scheduling algorithms, we also collect the failure rate of transactions, which is defined as the number of aborted transactions compared to the total number of transaction submitted to transaction manager.

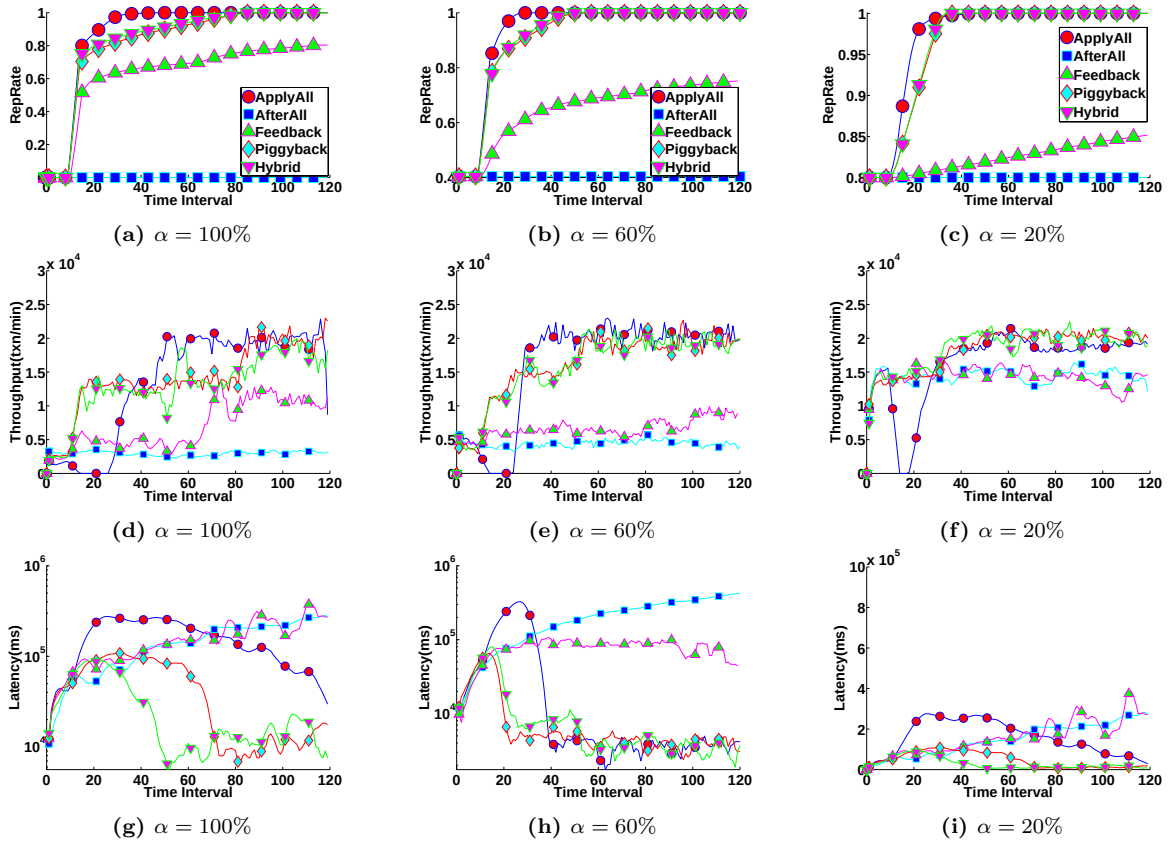
In line with the workload generation, we divide the time into 20 seconds of intervals and run the system for 10 intervals to warm it up before we start the repartitioning. Furthermore, the feedback-based approach uses 20 seconds as the monitoring interval. For the feedback model parameter used in each of the experiment, we have the different SP values which are listed in Table 1. All the experiments will have the 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 soon.

Zipf workload. The experimental results are presented in Figure 4. 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 Figures 4a, 4b and 4c. However as one can see from Figures 4d, 4e and 4f and Figures 4g, 4h and 4i, using this approach will experience a period that system has a very low throughput and very high processing latency caused by the stalling of the normal transactions.

As shown in Figure 4i, the impact on processing latency can actually last much longer than the time needed to perform the repartitioning. This is because a long waiting queue



(a) $\alpha = 100\%$ (b) $\alpha = 60\%$ (c) $\alpha = 20\%$

(d) $\alpha = 100\%$ (e) $\alpha = 60\%$ (f) $\alpha = 20\%$

(g) $\alpha = 100\%$ (h) $\alpha = 60\%$ (i) $\alpha = 20\%$

Figure 4: Experiment Results for Zipf High Workload

will be built up during the repartitioning process and hence it needs a longer time to clear the queue. (Note that we have set the initial ratio of workload to system capacity be roughly equal for all α values and we actually submit more normal transactions in the case with a lower α value. Hence the waiting queue will be longer in this case.)

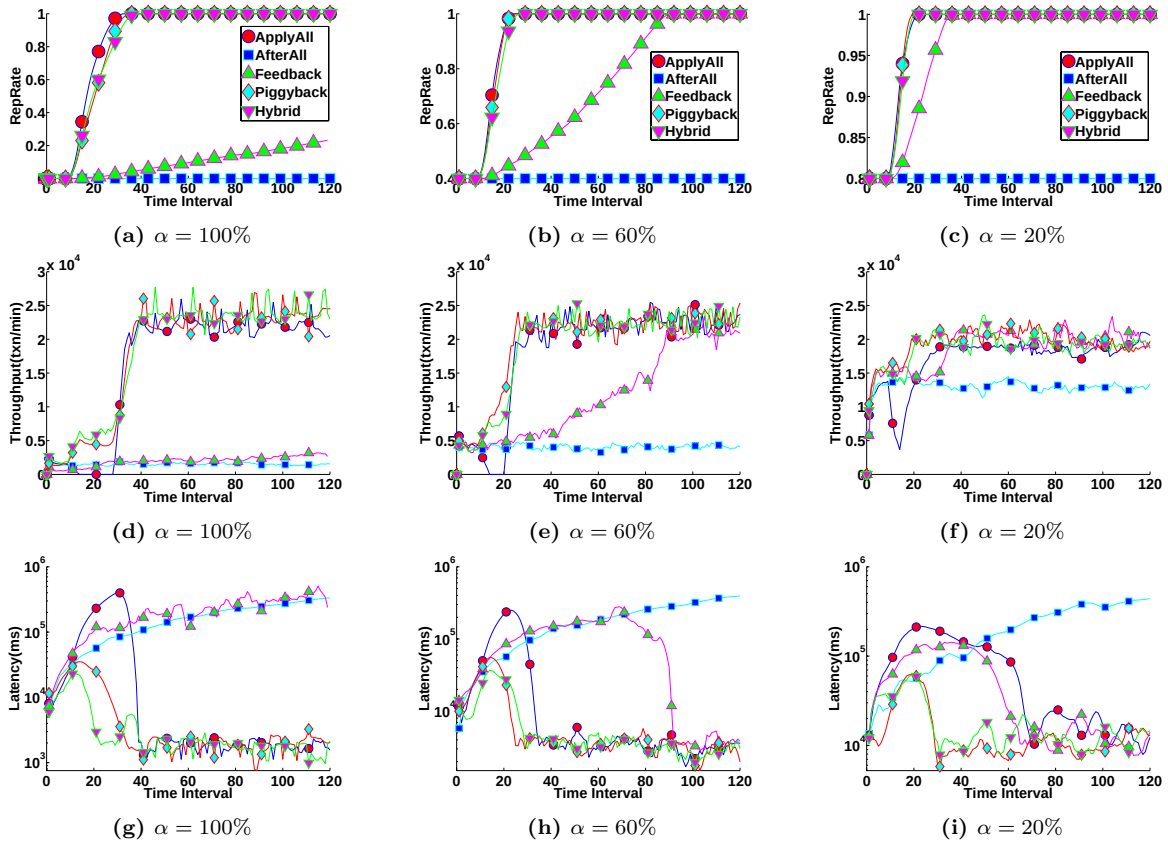
On the contrary, AfterAll basically cannot execute any repartition transactions due to the lack of system idle time, hence it cannot take advantage of the new data partition plan. The Feedback approach will enforce the scheduling of some repartition transactions, hence can make some progress in deploying the new partition plan (Figures 4a, 4b and 4c). Accordingly the system's throughput and processing latency will improve gradually. (Again, we submit more normal transactions in the case with a lower α value. So it takes a longer time to deploy the repartition plan.)

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 4 verify our analysis. In comparing to ApplyAll, both Piggyback and Hybrid do not incur any sudden dropping of system performance while is able to quickly execute the repartition plan. It even outperforms ApplyAll at almost all time intervals in terms of both throughput and latency with a lower α value, i.e. fewer tuples to be repartitioned.

Uniform workload. We also perform the experiments with a workload under a uniform distribution. The results are presented in Figure 5. 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, while ApplyAll finish the repartition process in 20,12 and 4 intervals, which is proportional to the number of repartition transactions that need to be executed.

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. Under $\alpha = 100\%$, we cannot apply enough repartition transactions to stop the queue size from increasing. So even the repartition rate increases a bit in figure 5a, the throughput and latency does not get improved. But under the cases with $\alpha = 60\%$ and $\alpha = 20\%$, since the number of repartition transactions we need to execute is smaller, the system finally finish the repartition in time and make the system able to process all the incoming normal 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.



(a) $\alpha = 100\%$ (b) $\alpha = 60\%$ (c) $\alpha = 20\%$

(d) $\alpha = 100\%$ (e) $\alpha = 60\%$ (f) $\alpha = 20\%$

(g) $\alpha = 100\%$ (h) $\alpha = 60\%$ (i) $\alpha = 20\%$

Figure 5: Experiment Results for Uniform High Workload

Similar to the Zipf workload, both Piggyback and Hybrid performed the best in all cases. They can achieve a speed that are almost identical to the *ApplyAll* approach.

Transaction failure rate. To further investigate the effect of the algorithms, we also collect the failure rates of transactions. Here we only report the results with $\alpha = 100\%$ since we could experience highest lock contention in these scenarios. The results are shown in Figure 3. We can see from Figure 3a, AfterAll has a very high failure rate during the whole period simply because the system's workload is very high and it fails to apply the repartition plan to quickly improve the system's performance. Furthermore, both the piggyback and hybrid method has a very low failure rate through the whole period, which clearly show the piggyback-based method's advantage of lowering lock contentions. On the other hand, the feedback-based method experiences some failure caused by the extra transactions scheduled by the feedback-based method.

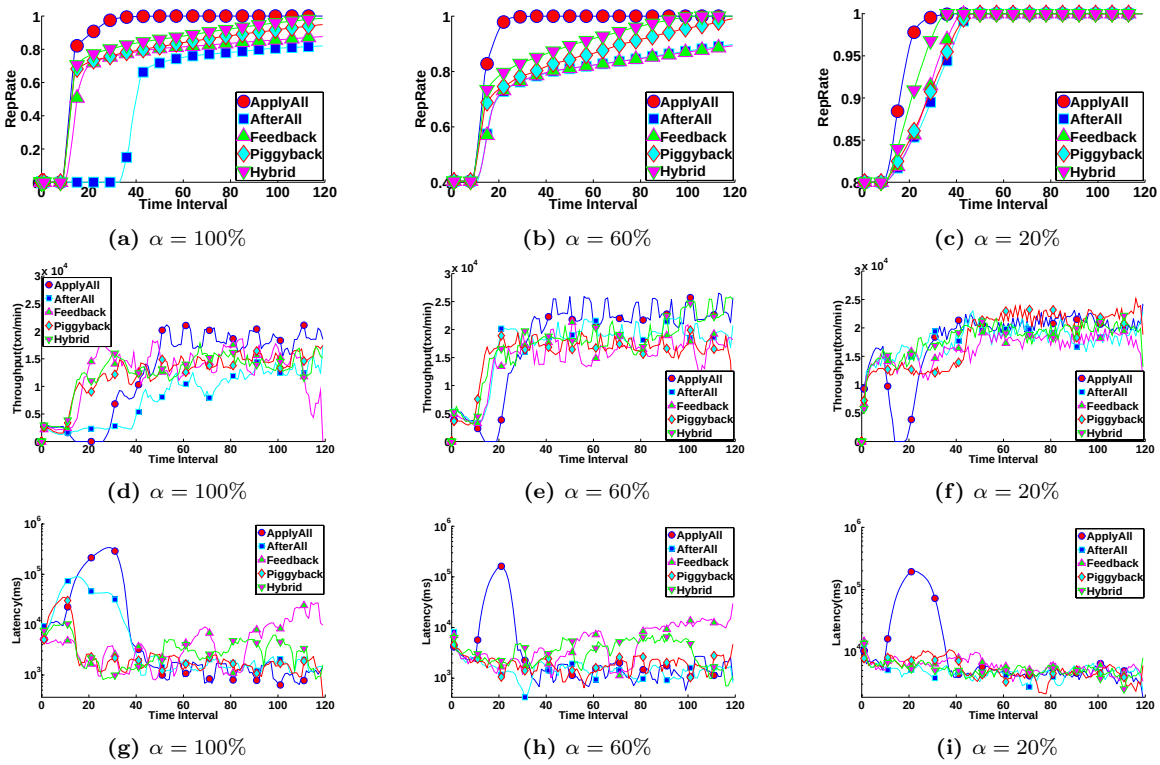
Figure 3b shows the results with the uniform workload. The general trend is similar. But it is interesting to see that the extra failure rate caused by the piggyback strategy lasts longer than the case with Zipf workload. This is because there is not any very hot transaction in this scenario, so we cannot deploy the repartition plan as quickly as in the case with Zipf workload. The mechanism of executing more high-priority repartition transactions with Hybrid causes a higher initial failure rate but it drops more quickly than the piggyback approach. This shows that Hybrid has an edge when there are less transactions to piggyback on.

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 make use of those available resources. For the Zipf workload, since there exist some transactions that have very high frequencies, the resource contention under the same load level will be higher than the Uniform workload. The results are shown in Figure 6 and 7.

Zipf 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 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. In Figure 6a, we can see that it takes some time for the system to have an idle period. It happens that there are three intervals that have a very high workload generated by our Poisson load generator. When the tuples accessed by the high frequency normal transactions are not repartitioned yet, their average execution time will be much higher than normal because of resources contention. We could see that AfterAll does not have this problem in Figures 4b and 4c. AfterAll has the minimum interference to the normal transactions when the system is able to handle the normal transaction load like



(a) $\alpha = 100\%$ (b) $\alpha = 60\%$ (c) $\alpha = 20\%$

(d) $\alpha = 100\%$ (e) $\alpha = 60\%$ (f) $\alpha = 20\%$

(g) $\alpha = 100\%$ (h) $\alpha = 60\%$ (i) $\alpha = 20\%$

Figure 6: Experiment Results for Zipf Low Workload

the situations in Figure 6h and 6i and hence it could be the algorithm that can achieve the lowest average latency.

The Feedback method will add more repartition transactions to the system besides filling the idle period. So we can see in Figure 6g that it partitions the tuples accessed by some high frequency transactions and render the system load decreasing more quickly than AfterAll. Adding the extra repartition transactions will increase processing latency of the normal transactions. This extra latency is a trade-off against the repartitioning speed. In Figure 5d, we can see that Feedback has a higher throughput than AfterAll but with a higher latency before it finishes all the repartition transactions.

The overhead of the piggyback method is proportional to the number of piggybacked transactions. When the workload is low, like the condition in Figure 6f, it may not be able to repartition the database as fast as the other methods. But the overhead on latency is also lower when the workload is low, which is similar to AfterAll.

Hybrid performs very well even under low workload. It always finishes the repartitioning work with a speed that is only slower than ApplyAll and in the meantime keeps the latency overhead less than the Feedback method. Since the repartitioning speed of Hybrid is faster than Feedback, the time period that normal transactions will experienced some extra latency are much shorter.

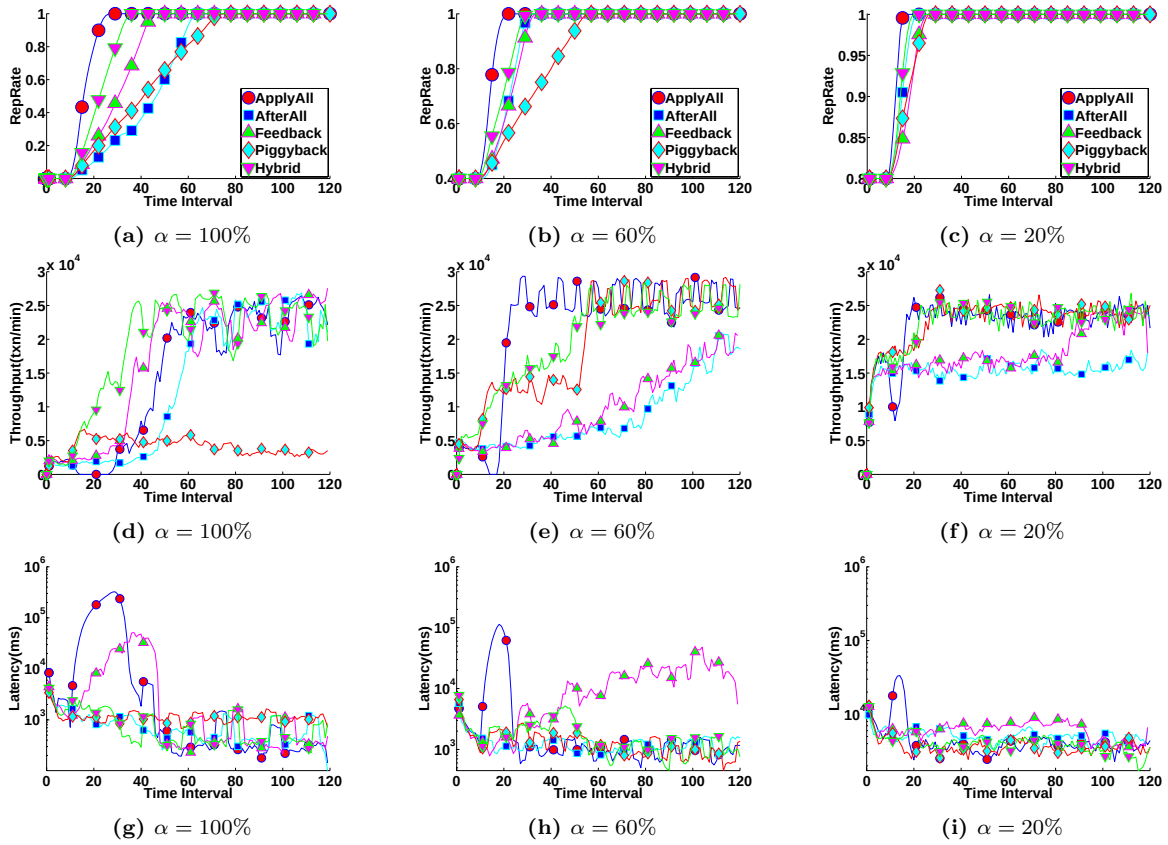
With regard to the failure rate, the trend shown in Figure 3c is very similar to the case with high workload, except that AfterAll has a much lower failure rate in this case. This is because the system's workload is not that high and AfterAll has the opportunity apply the repartition plan to further improve the system's performance.

Uniform workload. The results are reported in Figure 7. In this case, the degree of resource contention among normal transactions is lower than that with the Zipf workload. AfterAll could finish the repartitioning more quickly than with the Zipf load. Since the frequency of each normal transaction is low, the effect of repartition may take some time to get into effect. An interesting phenomenon is that Piggyback in this case works worse than the previous cases. With the uniform and low load situation, there are relatively few transactions to be piggybacked on and hence it take much longer for the piggyback approach to finish the repartitioning.

Furthermore, from Figure 3d, we can find that before the repartitioning is finished, the piggyback approach incurs a higher failure rate. This is because, even though Piggyback does not incur additional transactions, the piggybacked transactions will run longer than normal and this may still increase lock contentions to a certain degree. Given the longer repartitioning period in this case, its overhead on failure rate is more prominent here. On the contrary, Hybrid is able to make use of the available system resources to speed up the repartition and hence do not suffer this problem.

5. RELATED WORKS

Data partitioning for distributed database system is about designing a data placement strategy minimizing the transaction overheads and balancing the workloads. Besides basic algorithms using some static functions, such as range-based or hash-based, to partition the data, researchers have recently been focusing on workload-aware partitioning al-



(a) $\alpha = 100\%$ (b) $\alpha = 60\%$ (c) $\alpha = 20\%$

(d) $\alpha = 100\%$ (e) $\alpha = 60\%$ (f) $\alpha = 20\%$

(g) $\alpha = 100\%$ (h) $\alpha = 60\%$ (i) $\alpha = 20\%$

Figure 7: Experiment Results for Uniform Low Workload

gorithms that take the transaction statistics as input [4, 13]. These solutions utilize various techniques to model the historical workload information and search for an optimal partition scheme according to a specific objective function. Schism [4] is an automatic partitioning tool trying to minimize distributed transactions. Horticulture [13] further improve this approach by considering temporary skewness of workloads and using a local search algorithm to optimize partition schemes. This work provides a cost model for processing a transaction that considers both the number of partitions the transaction need to access and the overall skewness of data access.

Alekh etc.[8] present an online partitioning method that will partition the data in checkpoint time intervals. They generate partition schemes based on historical query execution logs and automatically update the partition plan when the workload changes. Besides the static partitioning methods, there are also some studies on incremental partitioning. For example, Sword [15] adopts a similar model as Schism [4] and introduces an incremental repartitioning algorithm that calculates the contribution of each repartition operation and the cost of executing it. They also propose a threshold on the number of repartition queries that will be generated for each repartition step and a constrained number of repartition queries will be executed when the system is at a lean period. This approach is similar to our baseline solution *AfterAll*. Some commercial database systems [12, 10] support online partitioning. But all of them require the partitioned data would not be untouched by normal transactions during the partitioning period. Hence this is similar to our *ApplyAll* solution. In our former short paper[1], we briefly presented the basic ideas about the feedback and piggyback algorithms. In this paper, we provide a more thorough analysis of the problem, consider the drawbacks of the piggyback solution and provide more experiment results to illustrate the trade-offs in the piggyback solution. We also propose the hybrid approach that combines the feedback and piggyback algorithm to benefit from the advantages of both algorithms while avoiding the problem of high failure rate in the piggyback solution and the problem of high interference with normal transactions in the feedback solution.

On the other hand, some researches on quality of service (QoS) of OLTP systems, such as [11], have considered the different resources' influence on transaction performance and have attempted to find the bottleneck resource for OLTP transactions and shows an arresting performance improvement. [6] makes use of machine learning methods to predict multiple performance metrics of analytical queries. The solution relies on SQL text extracted from the query execution plans. In [18], the authors addressed predictions of query execution time using the query optimizer's cost model and the generated query plan, which can be used to estimate the transaction execution time in OLTP systems. For distributed systems, the extra cost of network communication will be the new bottleneck, which limits the OLTP transactions performance [3]. It is an important part of execution

cost if we want to optimize the transaction performance in a distributed environment.

Live migration of databases [5] focuses on migrating a whole database from one system to another while providing non-stop services and has not considered the scenario of migrating data from multiple nodes to multiple destinations, which however is the common case in our online data partitioning problem and may encounter distributed transactions that update the data at both the source and the destination nodes simultaneously. The authors of [5] provide a solution of combining on-demand pull and asynchronous push to migrate a tenant with minimal service interruption. Their solution is somehow similar to our piggyback approach, where data that needs to be moved will be migrated when the incoming transactions visit it.

Transaction scheduling is an important topic studied in various areas such as Web services and database systems, and there are several works, such as [7, 2]), that tried to find an optimal schedule by considering query execution time, transaction deadline and system workload. Given the transactions' execution time and hard execution deadlines, most of the scheduling problems are NP-Complete [16]. The most common solution is cost-based algorithms [9]. The quality of a schedule highly depends on the cost estimation and how the execution cost each transaction is modeled. [14] is a scheduling and admission control method using a priority token bank in computer networks. They classify jobs into N classes and jobs within each class are treated equally (by using FIFO). This approach is much simpler than cost-based scheduling (CBS).

6. 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.

7. REFERENCES

[1] Kaiji Chen, Yongluan Zhou, and Yu Cao. Scheduling online repartitioning in oltp systems. In *Proceedings of the Middleware Industry Track*, Industry papers, pages 4:1–4:6, 2014.

[2] Yun Chi, Hakan Hacígümüş, Wang-Pin Hsiung, and Jeffrey F Naughton. Distribution-based query scheduling. *Proceedings of the VLDB Endowment*, 6(9):673–684, 2013.

[3] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. 41:98–109, 2011.

[4] 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.

[5] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 301–312, 2011.

[6] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael I Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. pages 592–603, 2009.

[7] Shenoda Guirguis, Mohamed A Sharaf, Panos K Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Adaptive scheduling of web transactions. pages 357–368, 2009.

[8] Alekh Jindal and Jens Dittrich. Relax and let the database do the partitioning online. In *BIRTE*, pages 65–80, 2011.

[9] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[10] MSDN Library. Partitioned tables and indexes.

[11] David T McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority mechanisms for oltp and transactional web applications. pages 535–546, 2004.

[12] Oracle. Using partitioning in an online transaction processing environment.

[13] 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.

[14] Jon M Peha. Scheduling and admission control for integrated-services networks: the priority token bank. *Computer Networks*, 31(23):2559–2576, 1999.

[15] 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, 2013.

[16] Jeffrey D. Ullman. *NP*-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

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

[18] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, H Hacigumus, and Jeffrey F Naughton. Predicting query execution time: Are optimizer cost models really unusable? pages 1081–1092, 2013.

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