iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases

Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang,
Honglin Qiao, Yue Shi, Wei Cao, Rui Zhang
Alibaba Group
{j.tan, tieying.zhang, lifeifei, aiao.cj, q.zheng, chiyuan.zp, kenny.qhl,
yue.s, mingsong.cw, jacky.zhang}@alibaba-inc.com

ABSTRACT

Tuning the buffer size appropriately is critical to the performance of a cloud database, since memory is usually the resource bottleneck. For large-scale databases supporting heterogeneous applications, configuring the individual buffer sizes for a significant number of database instances presents a scalability challenge. Manual optimization is neither efficient nor effective, and even not feasible for large cloud clusters, especially when the workload may dynamically change on each instance. The difficulty lies in the fact that each database instance requires a different buffer size that is highly individualized, subject to the constraint of the total buffer memory space. It is imperative to resort to algorithms that automatically orchestrate the buffer pool tuning for the entire database instances.

To this end, we design iBTune that has been deployed for more than 10,000 OLTP cloud database instances in our production system. Specifically, it leverages the information from similar workloads to find out the tolerable miss ratio of each instance. Then, it utilizes the relationship between miss ratios and allocated memory sizes to individually optimize the target buffer pool sizes.

To provide a guaranteed level of service level agreement (SLA), we design a pairwise deep neural network that uses features from measurements on pairs of instances to predict the upper bounds of the request response times. A target buffer pool size can be adjusted only when the predicted response time upper bound is in a safe limit. The successful deployment on a production environment, which safely reduces the memory footprint by more than 17% compared to the original system that relies on manual configurations, demonstrates the effectiveness of our solution.

PVLDB Reference Format:
Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. PVLDB, 12(10): 1221 - 1234, 2019.
DOI: https://doi.org/10.14778/3339490.3339503

1. INTRODUCTION

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 12, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3339490.3339503

Buffer pool is a critical resource for an OLTP database, serving as a data caching space to guarantee desirable system performance. Empirical studies on Alibaba's OLTP database clusters with more than 10,000 instances show that the buffer pool consumes on average 99.27% of the memory space on each instance, as shown in Table 1. Existing buffer pool configurations are almost unanimously based on database administrators (DBAs)' experiences and often take a small and fixed number of recommended values. This manual process is neither efficient nor effective, and even not feasible for large cloud clusters, especially when the workload may dynamically change on individual database instances.

Table 1: Usage of different memory pools

Memory Pool	buffer pool	insert buffer	log buffer	join buffer	key buffer	read buffer	sort buffer
Avg. Size	29609.98M	8.00M	200.00M	0.13M	8.00M	0.13M	1.25M
Percent	**99.27%**	0.03%	0.67%	0.00%	0.03%	0.00%	0.00%

To demonstrate, Figure 1 plots the buffer pool size versus the cache miss ratio and response time for each instance on a subset of 10,000 instances. It uses a box plot with each box covering $25\% - 75\%$ sample values for each buffer pool size on a log-log scale. There are two observations: 1) only a small number of buffer pool size configurations (10 in this case) are used by DBAs in this environment; 2) for the same buffer size configuration, database instances exhibit a wide spectrum of cache miss ratios and response times.

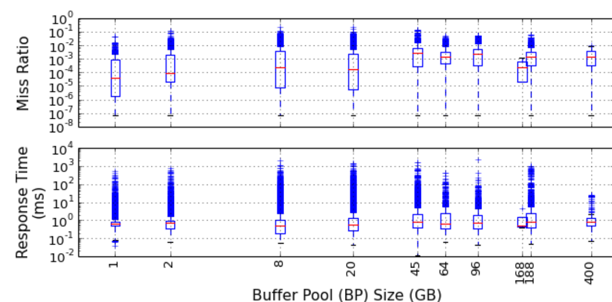

Figure 1: Miss ratio/response time versus buffer pool size

More importantly, Figure 1 reveals that solely relying on DBAs to optimize the buffer sizes is not scalable for a large cloud database cluster that contains tens of thousands of servers, especially when a variety of diverse workload and applications can dynamically change over time. It is imperative to develop a principled and automatic method to optimize individual buffer pool size for a production cloud database cluster that is at large scale.

To that end, we propose iBTune to automatically reduce buffer size for any individual database instance while still maintaining the quality of service for its response time, without relying on DBA to set forth a predefined level. The memory saving is critical for efficiently running a large-scale cloud database cluster. However, this saving should only have minimal negative effects on query response times and throughputs. Since iBTune is deployed online, we have successfully reduced the memory consumption by more than 17% while still satisfying the required quality of service for our diverse business applications.

We emphasize that bluntly reducing all instances' memory footprint by 17% or randomly reducing their sizes can lead to very negative impacts (i.e., query response times will rise abruptly and violate business application constraints or SLA), as each instance's query workload is different and requires careful individual analysis. iBTune learns which instances it can reduce buffer sizes aggressively without negatively impacting the response times, and which instances it should maintain the same buffer sizes to yield stable, targeted query response times. Figure 2 compares the cumulative percentage distributions of the individual buffer pool sizes before and after the iBTune applies the online adjustments, respectively. iBTune does so through learning algorithms

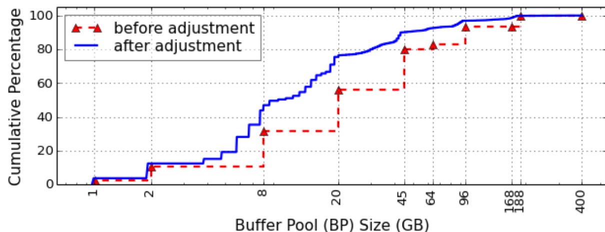

Figure 2: Distribution of the individual buffer pool sizes

without relying on DBAs, since DBAs simply cannot tune 10,000 instances in real-time. The overall memory footprint of the entire buffer pool is reduced yet not a single instance's query response time is negatively impacted to an extent such that the hosted business applications violate their business constraints or specified SLA.

From the system perspective, online adjusting the buffer size of an instance is through switching the roles of a primary instance and a backup. If the adjusted buffer pool size turns out to be bad, a failure recovery mechanism can bring the instance back to its previous state. Given the overhead of switching roles and failure recovery, gradually reducing the buffer sizes of selected instances until failure recoveries occur is not an option. Unnecessary switch and rollbacks can create too much intrusion and overhead to the supported business applications.

Even though automatic database tuning has been extensively studied [42, 29, 31, 27, 39, 9, 25], it still remains as an open problem how to tune individual buffer size for a large cloud database cluster. There are two key challenges: 1) decide the proper buffer pool size for each individual instance and 2) estimate the new response times (RT) due to adjusting buffer pool sizes dynamically.

Determining the proper memory size for a OLTP database buffer pool is different from estimating other resource consumption such as CPU, heap/stack space and disk. The distinction lies in the fact that almost all the allocated memory resource will be completely consumed by the buffer pool that is predominantly running either least recently used (LRU) caching algorithm or its variants [43, 33], e.g, CLOCK [7], LRU-K[30], LRFU [24], EELRU [36]. As a consequence, the measured memory consumption will always be close to the configured buffer pool size. Nevertheless, the impact indirectly affects critical performance indicators, e.g., response times and cache miss ratios, which cannot be directly translated to specific buffer pool size values.

Furthermore, the problem to accurately predict response times for different buffer pool sizes, albeit important, is quite challenging. Using the best algorithm that we have tested so far (a pairwise deep neural network that takes the features from measurements on pairs of instances) still results in a prediction error around 30% (see Table 7). Therefore, we use it as a sanity check, by predicting the upper bounds of the response times. A target buffer pool size can be adjusted only when the predicted response time upper bound is in a safe limit. The safe limit is determined by performance statistics or specified SLA depending on applications.

A common engineering approach in configuring the buffer pool sizes is to over-provision memory in order to guarantee service quality. For example, based on the diagnoses of DBAs, if the miss ratio is too high, or the response time is too large, or the IO read is overwhelming, or too many slow SQL queries negatively impact the business applications, the buffer pool size would be manually reconfigured, e.g., by doubling its size. This commonly used aggressive approach is often by compulsion, not by choice, in order to quickly provide enough resources for a database instance to fulfill business performance goals and to avoid possible risks induced by frequent reconfigurations. However, it can incur significant operational cost and result in wasted memory resource. Note that memory is often the critical bottleneck for high-performance OLTP databases.

Overview and our contributions. To this end, we present iBTune that has been deployed in our production system of more than 10, 000 database instances. Specifically, we utilize the relationship between miss ratios and buffer pool sizes to optimize the memory allocation. Our models leverage the information from similar instances. Meanwhile, we design a novel pairwise deep neural network that uses the features from measurements on pairs of instances to predict the upper bounds of the response times. Specifically,

1) We integrate caching algorithms with learning techniques on measured data. It relies on a model-driven large deviation analysis for LRU caching and a data-driven deep neural network for measurements. Specifically, we infer the tolerable miss ratio $mr_{tolerable}$ for each database instance using its similar instances that report larger but acceptable miss ratios. Then, using the tolerable miss ratio and the current buffer size bp_{cur} of the given instance as two parameters, we compute the target buffer pool size bp_{target} based on a large deviation analysis for LRU caching models.

2) Since $mr_{tolerable}$ cannot be explicitly translated to RT that DBAs care the most, we design a neural network to predict a RT upper bound due to adjusting the buffer pool size from bp_{cur} to bp_{target} by leveraging workload characteristics and configurations. Since overestimation of RT is not as harmful as underestimation, we introduce an asymmetric loss function to measure errors. To improve the predict accuracy, we design a pairwise deep neural network that takes the measurements from a pair of similar instances as inputs.

Paper organization. We present an overview of iBTune and its system design in Section 2, and discuss specific al-

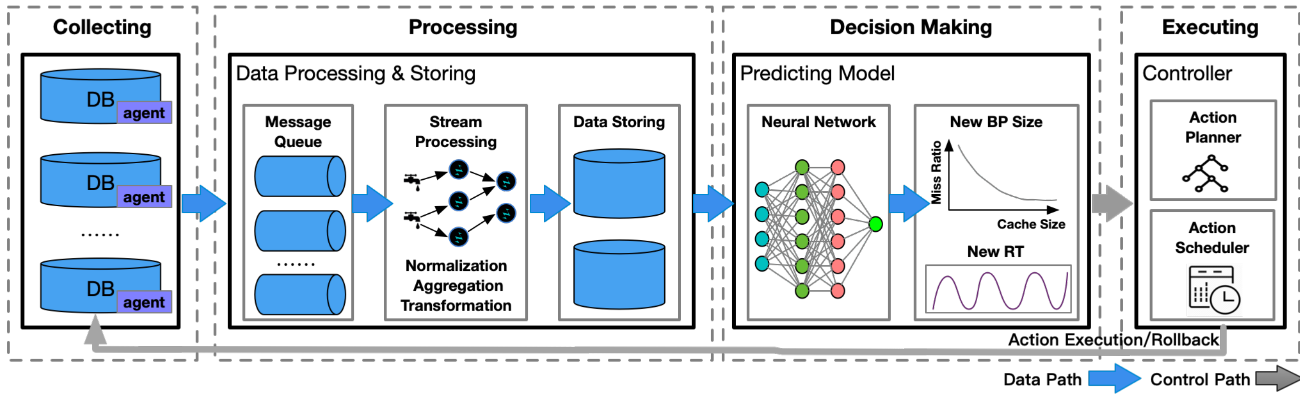

Figure 3: Workflow and overall architecture of iBTune

gorithmic and technical details in Section 3. We evaluate iBTune with extensive experiments using real workloads on a large production environment with more than 10,000 database instances in Section 4. We review related works in Section 5 and conclude in Section 6.

2. OVERVIEW OF IBTUNE

In this section, we present an overview of iBTune and describe the details of the system design. We also explain how it works in our production environment. To this end, we first explain its workflow and key components based on the system architecture. Then we introduce how to guarantee system availability through a rollback mechanism, which is critical for deployment in a production environment.

System implementation. We implement a full-fledged platform to provide monitoring, optimization, parameter configuration, testing, failure recovery and rollback mechanism for our production database cluster that contains tens of thousands of instances. Note that these functionalities are necessary for all large-scale production OLTP database clusters. Therefore, they do not introduce any overhead for deploying our iBTune algorithms. Our algorithm however relies on these system features to provide the service. Specifically, there are three key challenges.

Measurement collection: the database metrics are collected for feature engineering and model training. In our production system, the DBMS forms a complete ecosystem as illustrated in Figure 3. Our database kernel captures and outputs critical runtime performance information with $1,000$ different types of metrics. These metric data are stored and processed in our production platform that contains a distributed storage layer. For the purpose of optimizing buffer pool sizes, we only use relevant metrics including logical reads, io reads, QPS, CPU usage and response times. The details are described in Section 3.3.1.

Online processing: our buffer pool optimization is based on a model trained offline and all results are also computed offline. However, the suggested target buffer pool size bp_{target} needs to be configured and take effect in the online runtime. The executing component is responsible for enforcing this online reconfiguration. The implementation is through switching the roles of a master node and a backup node for the same instance. Note that this DOES NOT introduce any overhead, as each database instance is running a high availability protocol (e.g., Paxos [23]) with at least 3 copies. Specifically, to reconfigure the buffer pool size, the DB kernel suspends the service of the master node for a short period of time. Typically this time period ranges from several seconds to 20 seconds, which depends on the size of the buffer pool. Due to the strong support of high availability by the DB kernel, the leader (master node) can be quickly taken over by one of the followers (backup nodes). The system availability is maintained throughout this transition.

Failure recovery and rollback mechanism: since reconfiguring buffer pool sizes is expensive, this operation is only conducted when the system workload is low in order to reduce the impact to business. The DB kernel provides high availability based on PAXOS [23]. If a leader fails, the DBMS automatically selects a new one from several followers and switches its role without interrupting the running service.

2.1 System architecture and workflow

Figure 3 presents an overview of iBTune's architecture and workflow. There are four key components: *data collection, data processing, decision making, and execution.* The iBTune workflow forms a closed cycle, since data is first collected from DBMS kernel, processed and used for training, and then resulting models are applied to the DBMS again.

Data collection. We use customized agents to collect various database metrics and logs from DBMS. More than 1,000 metrics are collected. The agent sits outside DBMS to avoid unnecessary performance overhead to the DBMS kernel. All metrics and logs are collected in one second granularity and fed into the *data processing* component. The specific metrics that iBTune collects are discussed in Section 3 along with the specific algorithms and models that they feed into.

Data processing. These metrics are first ingested into a message queue system, which decouples data collection and data processing. A stream processing system then reads data from the message queue and performs certain data manipulation/standardization operations such as normalization, aggregation and log transformation. After that, the processed metrics and logs are stored in a distributed data store for analysis and model training.

Decision Making. Although we collect a lot of metrics, iBTune may select to only use part of them. Some key metrics consist of logical reads, io reads, RT, miss ratio, QPS, CPU usage, etc. Logical reads are the read counts including memory reads and IO reads. IO writes are also taken into the consideration, but since the result in most cases is no difference from without it, we did not use IO writes in our

model. We use the method proposed in Section 3 to predict RT and compute the new BP (buffer pool) size. If the predicted RT meets the requirement, the computed new BP size is sent to the *execution* component.

Execution. This component consists of *action planner and action scheduler*. To process a large number of database instances, action planner aims to make a globally efficient and non-conflicting execution plan for tens of thousands of actions. It includes priority settings among different action categories, action merging for the same instance, action conflict detection and resolution, canary executing strategy and so on. This is done through a series of optimizations that are not the main focus of this paper. Its final output consists of several action sequences to action scheduler.

Action scheduler sets up specific execution time for all actions. For example, some actions execute in real time manner, and some execute during off peak hours. Our DBMS has the ability to resize buffer pool online. That means it is not necessary to restart the database instance, but it introduces less than a few to 10-20 seconds un-availability (system halting) during buffer pool online resizing.

With a high performance PAXOS [23] implementation, we avoid halting the system when adjusting BP sizes by leveraging the high availability of the system. Specifically, we have a leader (master node) with at least 2 followers (backup nodes). To adjust the buffer pool size, we first change a follower's buffer pool size, which typically takes several seconds. Then this follower switches its role with the leader. This transition is transparent to the application. Resultantly, iBTune provides high availability when adjusting BP sizes on the fly. Note that iBTune does not depend on PAXOS. Any other protocols that provide high availability can be used seamlessly with our system.

2.2 Rollback mechanism

If the response time slows down after the new BP size is applied, i.e., the number of slow SQLs increases, the DBMS will rollback to the previous BP size. This involves two steps: 1) slow SQL detection; 2) rollback operation.

SSAD service. In iBTune, slow SQL is used to measure how well the DBMS is running. SSAD (Slow SQL Anomaly Detection) service is implemented for this purpose. It detects the slow SQL differences comparing with the past in minute-by-minute statistics. A data collector agent outside the database sends the slow SQL statistics to SSAD periodically. SSAD then analyzes the running status of the database. If multiple consecutive anomaly values are detected, the SSAD service determines the DBMS state is unhealthy and triggers a rollback operation. In practice, we set the time span of multiple consecutive anomaly as 3 minutes.

Rollback. When SSAD determines a DBMS is unhealthy, it rollbacks to the previous BP size. With a PAXOS based high availability protocol, the rollback procedure is the reverse of applying new BP size. Specifically, we first set up a follower with the previous BP size, and then switch this follower and the leader. Similarly, it is transparent and non-interruptive to user applications.

3. ALGORITHM DESIGN IN IBTUNE

The optimization engine either routinely schedules or on demand triggers the invocation, e.g., in the mid-nights. Section 3.1 contains the high-level description. The details are presented in Section 3.2 on adjusting buffer pool sizes by miss ratios, Section 3.3 on finding similar instances, and Section 3.4 on predicting the response times, respectively.

3.1 High-level description

Top level. Each invocation processes multiple rounds of buffer pool size adjustment, as shown by the for-loop in Alg. 1. The instances are partitioned into N (e.g., $N = 10$) groups. Each round only adjusts the instances in a group based on their immediately similar neighbors. These groups can be sequentially adjusted, since each group can leverage the information from other already adjusted groups. In addition, groups avoid adjusting too many instances online at the same time to increase robustness. It may take multiple rounds for the adjustments originated from "source" instances to propagate to others along the neighbors.

Algorithm 1: Top level of the algorithm

1 **Parameter**: a threshold δ for adjustment
2 **Trigger**: scheduled by the optimization engine
3 Divide the instances into N groups $\{g_1, g_2, \cdots, g_N\}$;
4 $\delta \leftarrow threshold$;
5 **while** $\delta \geq threshold$ **do**
6   **for** each group g_i **do**
7   δ_i = a round of adjustment by Alg. 2 for all of the instances in g_i, protected by rollback;
8   **end**
9   $\delta = \max\{\delta_i\}_{1 \leq i \leq N}$
10 **end**

This whole process stops when no improvement, as quantified by a threshold δ (e.g., $threshold = 5\%$) in Algorithm 1, can be further realized. Note that for each round, the rollback mechanism protects the performance by, if necessary, recovering the previous buffer pool sizes.

After each round of adjustment is completed, we keep monitoring the experimented instances by collecting their health information. If their adjusted buffer pool sizes lead to unacceptable performance, the DBPaaS system immediately recovers these instances through the rollback mechanism. However, rollbacks introduce much intrusion to the running applications, which should be avoided if possible.

As shown in Algorithm 2, our algorithm adjusts the buffer pool size for each instance using a reference from its similar instances that can tolerate larger miss ratios and longer response times. Each instance i reports its current miss ratio $mr_{cur,i}$. Then, among its similar instances (denoted by a set $N(i)$), we restrict to the subset with miss ratios larger than $mr_{cur,i}$, and compute a target miss ratio $mr_{target,i}$ for instance i. This set $N(i)$ is constructed from a weighted graph G that connects each instance with its k-nearest neighbors (e.g., $k = 6$). In order to translate this information to buffer pool sizes, we characterize the functional relationship between miss ratios and buffer pool sizes by a large deviation analysis. This analysis addresses the widely used least recently used (LRU) caching algorithm under empirically observed heavy-tailed item popularity distributions, which computes the target buffer pool size $bp_{target,i}$.

Based on the target miss ratio and relevant performance metrics, we estimate an upper bound of the response time (RT) using a deep neural network based on pairwise instance comparisons. Notably, to accurately predict response time for different buffer pool sizes is extremely difficult due to a number of factors, e.g., shared workloads, I/O competition, server hardware differences, changing work dynamics. Af-

Algorithm 2: Compute targeted buffer pool size $bp_{(target,i)}$ by measurements of similar instances

Data: QPS_i, $mr_{(cur,i)}$, $logical_read_i$, RT_i, $bp_{(cur,i)}$ of all instances indexed by i
Result: Compute $bp_{(target,i)}$ for each instance i
1 Construct feature vector $F(i)$ for instance i in Section 3.3;
2 Build a graph G by $\{F(i)\}$ (connect each instance with its k-nearest neighbors) ;
3 **for** each instance i **do**
4 $\quad N(i) \leftarrow$ connected nodes of i on G;
5 $\quad$ **for** each instance j in $N(i)$ **do**
6 $\quad\quad$ **if** $mr_{(cur,j)} > mr_{(cur,i)}$ **then**
7 $\quad\quad\quad w_{ij} = \exp((F(j) - F(i))^2/(2\sigma^2))$;
8 $\quad\quad$ **else**
9 $\quad\quad\quad w_{ij} = 0$;
10 $\quad\quad$ **end**
11 $\quad$ **end**
12 $\quad mr_{(tolerable,i)} \leftarrow \frac{\sum_{j \in N(i)} w_{ij} mr_{(cur,j)}}{\sum_{j \in N(i)} w_{ij}}$;
13 $\quad$ Compute $bp_{(target,i)}$ for i using $mr_{(tolerable,i)}$, $mr_{(cur,i)}$ and $bp_{(cur,i)}$ by equation (3) ;
14 $\quad \delta_i \leftarrow \left(bp_{(cur,i)} - bp_{(target,i)}\right) / bp_{(cur,i)}$;
15 $\quad$ **if** $\delta_i \geq threshold$ **then**
16 $\quad\quad$ Adjust the buffer pool size from $bp_{(cur,i)}$ to $bp_{(target,i)}$;
17 $\quad$ **end**
18 **end**
19 **return** $max\{\delta_i\}$;

ter testing many different algorithms, so far the best result that we can obtain is to use a pairwise deep neural network, with an adjusted mean relative absolute error around 30% (see Table 7). To this end, we only estimate an upper bound of the response time and use it as a sanity check. A buffer pool adjustment is invalid when the predicted response time upper bound is within a safe limit. Such a safe limit is based on the response time distributions for different applications. Therefore, our approach combines domain knowledge on caching algorithms and learning techniques on measured data. As detailed in Sections 3.3 and 3.4, our current deployment collects one week and four weeks of measurements for RT prediction and similarity identification, respectively. Therefore, the adjustment frequency for each instance is on weeks.

3.2 Functional relationship between miss ratios and buffer pool sizes for LRU caching

Our database instances are serving mission critical applications, which require that the hit ratios of interests are in the typical range of 90.0% $\sim$ 99.99%. Thus, the LRU model naturally fits in the large deviation analysis that characterizes events with small probabilities, i.e., missed requests from buffer pools. Thus, from this point onwards, we focus on miss ratios.

There are three reasons for us to choose the miss ratio to bridge the buffer pool size and the desired system performance. First, the configured buffer pool size based on DBA's experiences only take on a small set of values, as illustrated in Figure 1 and analyzed in Section 1. Due to this coarse-grained value set, it is difficult to distinguish different instances in a refined manner. Second, using miss ratios provides a continuous way to interpolate the target buffer pool sizes, so that the suggested buffer pool sizes can take a wide spectrum of values beyond the existing set. Third, we have explained in Section 3.1 that predicting the response time is quite difficult. Thus, we cannot directly reply on predicted response times to determine the target buffer sizes.

For a large class of heavy-tailed requests with popularities following a power law [5, 3, 45, 32, 46], it has been shown that [12, 19, 40, 4, 38], under mild conditions, for instance i, there exist p_i, c_i such that

$$\frac{\log(miss_ratio) - p_i}{\log(buffer_pool_size) - c_i} \approx -\alpha_i, \quad (1)$$

where α_i is the index of the power law request popularity distribution. A number of empirical measurements on real systems have shown power law popularity distributions, e.g., $\alpha_i \in (0.6, 0.86)$ for content distribution systems [8] and $\alpha_i \in (0.2, 1.2)$ for the block I/O traces of storage systems [46]. For example in Figure 4, we vary the buffer size of an instance to obtain the corresponding miss ratios. In the log-log scale, it is very well approximated by a straight line (with an index $\alpha_i = 0.752$), which implies a power law miss ratio distribution. Note that α_i, c_i, p_i depend on the workload characteristics and the configuration of instance i. Thus, these values vary across different instances.

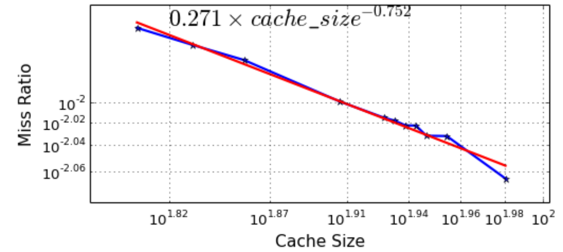

Figure 4: Illustration of power law

Given the measured current miss ratio mr_{cur}, the current buffer pool size bp_{cur}, and the computed target miss ratio mr_{target}, by equation (1), we can derive the following relationship for the target buffer pool size bp_{target}

$$\frac{\log(mr_{target}) - \log(mr_{cur})}{\log\left(bp_{bp_{target}}\right) - \log(bp_{cur})} \approx -\alpha_i. \quad (2)$$

In order to apply equation (2), we need to estimate α_i for instance i. To this purpose, we distinguish two cases. The first case is when the instance i has already adjusted its buffer pool size in the previous rounds. Thus, we have observed multiple (at least 2) previous measurements

$$\{(mr_1, bp_1), (mr_2, bp_2), \cdots\}$$

for instance i. These observed samples can be used to estimate α_i based on regression (e.g., Huber's method [18]). The second degenerate case is when we have only observed a single measurement point (mr_{cur}, bp_{cur}). It prevents us from estimating α_i based on instance i's history information. For this case, we conduct extensive measurements on various applications, and also use the reported statistical characteristics from literature [8, 46]. We observe that $\alpha_i \leq 1.2$ is satisfied by the investigated SQL requests. Thus, for instances lacking enough history information, we choose $\alpha_i = 1.2$ to provide a conservative buffer pool size adjustment.

There is still one issue about applying equation (1), which requires $miss_ratio > 0$. We again distinguish two cases:

normal instances with positive miss ratios and special instances with zero miss ratios. In practice, instead of using *zero* as the threshold to separate the two cases, we empirically choose $\zeta = 0.0002$ so that $mr_{cur} > \zeta$ indicates a normal instance and otherwise a special instance.

3.2.1 Normal instances

To apply equation (1) in our real system, we need to measure the current miss ratio mr_{cur}, which however could change from time to time. In order to reduce the variance, we choose the average miss ratio $\overline{mr}_{cur}$ during a time window, e.g., 24×5 hours during business days. In addition, we also consider certain percentile points that are more robust to noise. For example, we use the 70% percentile point mr_{P70} during the same time window as computing $\overline{mr}_{cur}$. In order to combine the previous two estimates, we empirically choose the current miss ratio to be the mean of $\overline{mr}_{cur}$ and mr_{P70}. Therefore, we obtain an estimation of bp_{target}

$$\frac{bp_{target}}{bp_{cur}} = \left(\frac{\overline{mr}_{cur} + mr_{P70}}{2 \times mr_{target}}\right)^{1/\alpha}. \tag{3}$$

Note that for normal instances, $mr_{cur} > \zeta$ implies that $\overline{mr}_{cur}$, mr_{P70} and mr_{target} are all positive. This formula is used in Algorithm 2 to compute the target buffer pool size based on the target miss ratio obtained from similar instances.

3.2.2 Special instances

Since equation (1) assumes that the observed miss ratios are strictly positive, we need to treat the case when $\overline{mr}_{cur}$, mr_{P70}, mr_{P80} and mr_{target} contain zeros separately. For such an instance i, we use the similarity used in Algorithm 2 to find the nearest instance j that is a normal instance. That normal instance will be used as a reference. Recall that in Section 3.2.1 we have already obtained $bp_{target,j}/bp_{cur,j}$ for all normal instances. Then, we use this ratio to adjust the special instance i, in a way such that

$$bp_{target,i}/bp_{cur,i} = bp_{target,j}/bp_{cur,j}.$$

3.3 Finding similar instances

In order to find the target miss ratios of a studied instance, we need to use the information from its similar instances. Measuring the similarity between different instances is critical to the performance of iBTune. We construct a feature vector $F(i)$ for each instance i, using both the low-level resource consumption measurements and the service quality related metrics, e.g., response time (RT) and query per second (QPS). Section 3.3.1 describes the feature engineering based on the raw database metrics. Section 3.3.2 provides the details on defining $F(i)$ based on direct performance measurements.

3.3.1 Feature engineering

The database metrics selected for RT prediction are logical read, io read, QPS, CPU usage and previously observed RT. The measurement is taken once every minute and summarized once every hour. Figure 5 plots the observed raw metrics on 7 consecutive days for one instance. It shows a clear daily pattern, which should be contrasted with Figure 13 that demonstrates the difference between workdays and holidays.

In Figure 5, the blue dots represent the hourly averages of the corresponding metrics. The hourly average of RT for

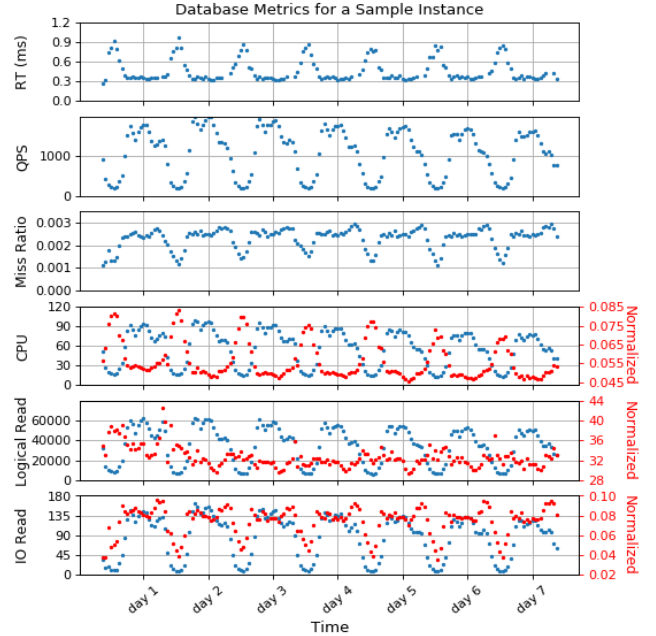

Figure 5: Database metrics over 7 days

this instance shows a daily pattern of a hump lasting for about 9 hours. However, over the same 9 hour period, the hourly mean of all other metrics are in the valley, which contradicts our common belief of the positive correlation between RT and the other metrics. Only after normalizing (dividing) CPU, logical_read and io_read by QPS, a daily pattern matching the one of RT appears in the normalized version. Here is one explanation. While the RT is the averaged response time per query, the raw measurements of QPS, CPU usage, logical read and io read are the summations within one minute. The normalized version gives the per-query value that is important for accurately predicting RT. The effectiveness of such an normalization is confirmed by the observation that Pearson correlation coefficients between RT and CPU, logical read and io read increase from 0.055, 0.127 and 0.017 to 0.093, 0.394 and 0.117, respectively. For the rest of the paper, CPU usage, logical read and io read are all normalized by QPS.

The prominent daily pattern in Figure 5 suggests that the time period within a day is also a factor for predicting RT. Thus we use the one-hot encoding for 6 periods with each period spanning 4 hours. For example, if the measurement is taken from 12 AM to 3:59 AM, the first time-encoding bit is 1 and the rest bits are 0; if from 4 AM to 7:59 AM, the second time-encoding bit is 1 and the rest bits are 0; and so on. These time encodings are used in the pairwise DNN model described in Figure 8.

After examining the time series of the database metrics for various instances, we realize that different instances exhibit distinguishing behaviors. Some manifest daily patterns while others have no patterns at all. Their RT values have complex dependency on other covariate metrics. To characterize the difference (or the similarity) among different instances, we construct a feature vector based on the normalized measurements in Section 3.3.2.

3.3.2 Similarity based on feature vector distance

We have described the 5 critical performance metrics, including CPU usage, logical read, io read, miss ratio and

RT (the first four metrics are divided by QPS, as explained earlier). However, they could have a large variation and possibly change dramatically on weekends compared to normal business days. Therefore, we select a relatively long time window, e.g., 4 weeks, and remove all the measurements on weekends. Since we focus on the system performance during peak hours, we use certain percentiles of the measurements to restrict the attention to important data.

Specifically, we collect the daily performance metrics on business days for each instance and compute 4 statistics (the 90th percentile, 70th percentile, median and mean values) for each of the 5 metrics. Thus, every business day yields a $4 \times 5 = 20$ dimensional data point. To further reduce the variance, we compute the average of each dimension over a time window of 4 weeks, excluding the weekends. To make sure that each of the dimension is equally weighted in computing the distances, the values on each dimension are normalized to be in the range $(0, 1]$ by dividing the maximum value. Last, these values are concatenated to form the feature vector $F(i)$ for each instance i. The parameter σ in Alg. 2 is chosen to be the average of $|F(i) - F(j)|$ for $j \in N(i)$ and all i.

In order to find similar instances, we use the L_2 norm to measure the distance between a pair of distances. Based on the distances, we can rank the similar neighbors. For illustration, Figure 6 plots two pairs of similar instances.

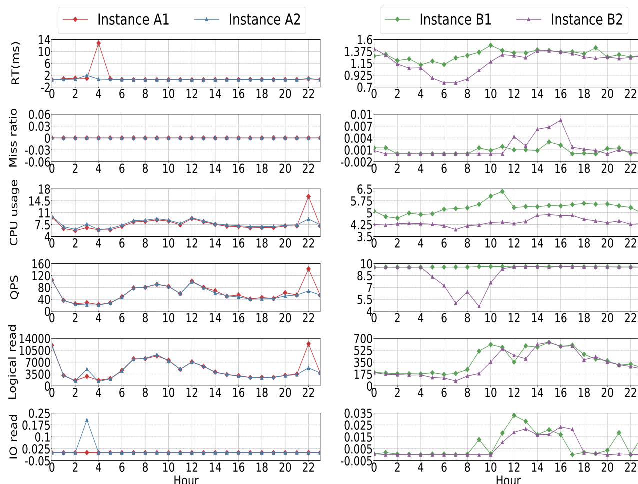

Figure 6: Two pairs of similar instances

Notably, measurements show that the types of server hardwares could significantly impact the response times caused by, e.g., I/O times. Therefore, in our algorithm and real deployment, we only compare two instances that are hosted on the *same type* of bare-metal server.

3.4 Pairwise DNN for RT prediction

As already explained in the introduction, to accurately predict response times after adjusting buffer pool sizes turns out to be very difficult. Therefore, we instead predict the upper bounds of the response times, which are compared with the safe limit. A target buffer pool size can be adjusted only when the predicted response time upper bound is within the safe limit. This mechanism provides a level of protection for the online adjustment. In addition, these predicted response times serve as references for DBAs.

Regarding the safe limit, we group all the instances into different applications, and find the 95% percentile of the response times in each group as the corresponding safe limit for that application. For example, Figure 7 plots the response time empirical distribution for 6 applications in our production cluster.

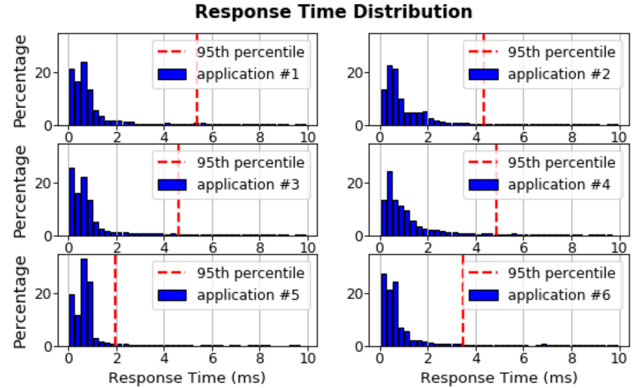

Figure 7: Determine the safe limits of response times

After adjusting the buffer pool size of an instance, it could experience a change of workload characteristics. When updating bp_{cur} to bp_{target}, the miss ratio is expected to change towards $mr_{tolerable}$. However, the instance may never have worked at this level of new miss ratio. It requires the RT prediction model to have a good generalization capability under possibly unseen environments.

The solution we propose is a pairwise DNN model, which is described in Section 3.4.1. When predicting RT for a specific instance, the pairwise model uses not only the measured performance metrics of the investigated instance but also the metrics of other instances that are most similar to this instance. Since we want to estimate the upper bound of the response times, we describe the asymmetric loss function that puts more weights on the large response times in Section 3.4.2. To reduce the generalization error, we design the training data set from pairs of instances based on their similarities in Section 3.4.3. We also compare our proposed model and other popular regression models in Section 3.4.4, which are validated by more concrete experiments in Section 4.2.3.

3.4.1 *Pairwise DNN model*

The proposed model for RT prediction is a DNN shown in Figure 8. Its inputs take the measurements from a pair of instances (left, right) as well as the time encoding that is described in Section 3.3.1.

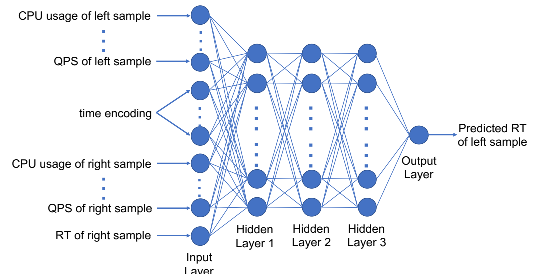

Figure 8: Neural network for predicting RT change

In the training phase, the left sample represents the instance whose response time we want to predict. The right sample represents a similar instance that contains history measurements. Therefore, the right sample has its response time *RT* available. In the testing phase, we apply this mod-

el to predict the RT due to changing the buffer pool size. Both left and right samples are actually collected from the same instance. Thus, the inputs have duplicates. Specifically, we use the performance metrics, including the already observed RT, on the last time period as the right sample, and the identical performance metrics except that the miss ratio mr_{cur} is replaced by $mr_{tolerable}$ as the left sample.

We use the mean values for all the inputs. Actually, we have tested other statistics, such as max, median and 90th percentile, as features in the RT prediction model. We only observe small improvements in the training errors at the cost of slightly larger testing errors. Therefore, we use only mean values for RT prediction.

Regarding the network structure, this pairwise DNN model has three hidden layers with 100, 50 and 50 neurons, respectively. All the layers are fully connected with rectified linear units (ReLU) as the activation function and the ReLU function is again used in the output layer to make sure the predicted RT is non-negative. The output of the DNN is compared with the measured RT to generate the error. The pairwise DNN model is trained with Adam optimization for 20 epochs using the ReL1 loss function introduced in Section 3.4.2.

Using this model, we estimate an upper bound for the RT due to adjusting the buffer pool size, which is used for quality control in Algorithm 1 and provides a level of protection.

3.4.2 Loss function

Due to system stability, underestimating RT has more severe consequences than overestimating. Therefore, we define the following asymmetric loss function, for a given $l(\cdot)$,

$$L : (e, \lambda) \to \begin{cases} l(e)I(e \geq 0) \\ \lambda l(e)I(e < 0) \end{cases} \tag{4}$$

where $I(.)$ is the indicator function, e is the error equal to the actual RT minus the predicted RT and $\lambda \in [0, 1]$ is a tuning parameter to control the impact of underestimation.

Regarding $l(\cdot)$, instead of using the mean square error (L2 norm $l(e) = e^2$) and mean absolute error (L1 norm $l(e) = |e|$), we define the relative error loss ($ReL1$), for $\eta = 0.1$,

$$l(e) = \left| \frac{e}{y + \eta} \right| \tag{5}$$

where y is the actual RT. The error e is normalized by the actual value y, because RT can vary by orders of magnitude across different instances. Minimizing the relative error is more meaningful than the absolute error, since an error of 1ms impacts instances with RT=0.1ms much more than instances with RT=10ms. The factor $\eta = 0.1$ in the denominator is to prevent the optimization result from being dominated by samples with very small RT.

3.4.3 Constructing pairwise training data set

The generalization capability of the proposed model is improved by forming pairwise samples based on the original data set for individual instances.

1. By the feature vector $\{F(i)\}$ computed in Section 3.3.2, we find the k_{pair} nearest neighbors for each instance, including itself, using the distance $||F(i) - F(j)||$ between instances i and j.

2. For each sample in the original data set (called a left sample), we find one data sample (called a right sample) from the same hour on a different date within the instance's k_{pair} nearest neighbors. This precedure expands the original data set to $k_{train}k_{pair}(k_{date} - 1)/2$ pairs of training points, where k_{train} is the sample size of the original data set and k_{date} is the number of different testing dates.

3. For each pair of (left, right) instances, as shown in Figure 8, we concatenate the features of the left sample, including CPU, logical read, io read, miss ratio, QPS and one-hot time encoding, with the features of the right sample, including CPU, logical read, io read, miss ratio, QPS and RT to form the new feature vector as the model input with RT of the left sample being the label.

Compared with a model that only uses a single data point in the original data set, we improve the performance by utilizing all observations in similar environments, as demonstrated by real experiments in Section 4.2.2. Since the workload of the left sample can be different from the right sample, we force the pairwise model to learn the RT difference when the workload or the configurations change.

3.4.4 Comparison with other prediction algorithms

In order to demonstrate the superior performance of our newly designed pairwise neural network, we compare its performance with other commonly used algorithms, including linear regression(LR), XGBoost regressor (XGB) [6], RANSAC regressor (RANSAC) [10], decision tree regressor (DTree) [34], elastic net regressor (ENet) [48], AdaBoost linear regressor (Ada) [35], gradient boosted decision trees (GBDT) [47], K-Neighbors regressor (KNR) [14], bagging Regressor (BR) [20], extremely randomized trees regressor (ETR) [16], random forest (RF) [26], sparse subspace clustering (SSC) [11], as well as a deep neural network with instance-to-vector embedding (I2V-DNN) but without pairwise comparisons. All these algorithms except the last one use the pairwise training technique, since otherwise the performance is much worse.

After a thorough examination of these prediction algorithms, we find that SSC, XGB and I2V-DNN yield the most competitive results among all the comparison candidates. However, these methods are still not as good as our designed pairwise neural network.

The SSC approach assumes that there exists a simple relationship for each database instance

$$\begin{aligned} RT_i = a \times QPS_i + b \times mr_{(cur,i)} \times logical_read_i \\ + c \times logical_read_i + d \times mr_{(cur,i)} + e, \end{aligned} \tag{6}$$

with the covariants coming from queueing delay, execution time and overhead, respectively. These features are carefully chosen from a variety of measurements. For example, we realize that in our case, *pages_flushed*, as an indicator for IO performance, does not really impact the response time. To apply SSC, the SQL workloads on different instances are assumed to form multiple unknown subgroups with different parameters (a, b, c, d, e). Each instance belongs to one subspace, but its membership, i.e., on which subspace it satisfies (6), is unknown. Given the representation in (6), the

objective is to find out all the subspaces as well as the membership of each instance. Then, within each subspace, we can apply all of the afore-mentioned prediction algorithms.

The DNN with instance-to-vector embedding (I2V-DNN) is illustrated in Figure 9. The unique instance name is

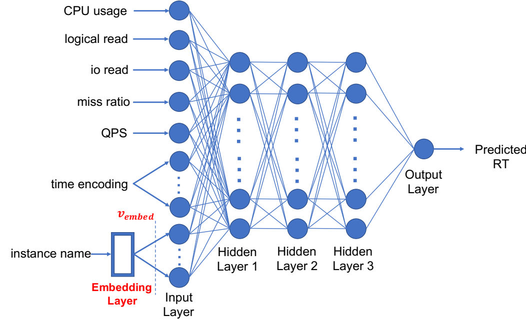

Figure 9: Instance2Vec embedding for I2V-DNN

passed into an embedding layer with a vocabulary size equal to the total number of unique instance names. The embedding size N_{embed} is chosen to be 20 after careful tuning. Then the output of the embedding layer, an $N_{embed} \times 1$ dense vector, is concatenated with other features. The comparison results are presented in Section 4.2.3.

4. EXPERIMENT

This section describes the setting and results of our experiments. The deployed DBMS is called PolarDB-X, a MySQL compatible database based on LSM-Tree storage engine [17]. The experiments are divided into two parts: 1) online testing for buffer pool size adjustment; 2) offline testing for response time prediction. To test the iBTune performance in the online environment, we select instances from different business units and present the experiments and the analysis. To quantitatively evaluate Pairwise DNN for predicting RT, we design a training and testing data set that spans over a holiday season. This evaluation is conducted offline to test the generalization capability of our model.

4.1 Workload and test setting

Online testing. In Algorithm 1, the database instances are divided into groups, which in our setting is selected to be 10 groups with each containing 1,000 instances scattered across different applications. We conduct multiple rounds of online adjustments sequentially. Since we need to collect the feedbacks when adjusting buffer pool sizes, our system currently adjusts 1,000 instances per week in the production environment that has 10,000 instances. As our system is continuously evolving, in the following part, we report the results from one typical week, covering 1,000 instances from more than 4 business units. The statistics of Select, Insert, Update and Delete SQL requests from these business units are provided in Table 2. It demonstrates that our proposed iBTune algorithm supports various workloads.

Table 2: Average QPS from different business units

	Taobao	Tmall	Youku	Fliggy	Others
Select	5245/s	2815/s	1017/s	22930/s	120/s
Insert	4222/s	0/s	1/s	1520/s	10/s
Update	0/s	30/s	315/s	4/s	980/s
Delete	2708/s	0/s	10/s	0/s	0/s

The buffer pool size distribution of these instances is shown in Figure 10, which differs from the distribution of the whole cluster in Figure 2. Among these instances, 286 are from Taobao[1], 212 from Tmall[2], 88 from Youku[3], 32 from Fliggy[4] and 382 from other business units. Each instance is loaded into a container (docker [1]). A container is hosted on a physical machine. We use two types of hosts, as in Table 3.

Table 3: Machine configurations

	Type1	Type2
CPU	Intel E5-2682 V4 × 2	Intel E5-2650 V2 × 2
Memory	32G × 24	16G × 24
NIC	10Gbps × 2	10Gbps × 2
HyperThreading	64	32

Offline testing. In order to quantitatively evaluate the accuracy of the response time prediction algorithm, we offline construct a training and testing data set by choosing a special holiday season. The performance metrics of most instances vary significantly before and during the holidays. We use the measurements before the season as the training data and the measurements during the holidays as the testing data. Judging by how well a model predicts the RT during holidays using data observed before the season, we can evaluate the generalization capability. We use a Tesla P100 GPU machine, with 512G memory and 64 CPU processors, to train our pairwise DNN model.

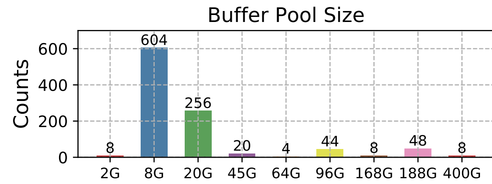

Figure 10: Buffer pool size distribution of 1,000 instances

4.2 Results

We first examine the online buffer pool adjustments in our production environment in Section 4.2.1. We compare the instances' performance before and after the adjustments using extensive measurements, statistics as well as the feedbacks from the business sides. In order to quantify the accuracy of the prediction algorithm, we design a training and testing data set to evaluate the prediction results in Section 4.2.2. Finally, we compare our algorithm with other methods in Section 4.2.3.

4.2.1 *Online adjustment of buffer pool sizes*

We compare the performance before and after adjusting the buffer pool sizes, using the sizes computed by iBTune.

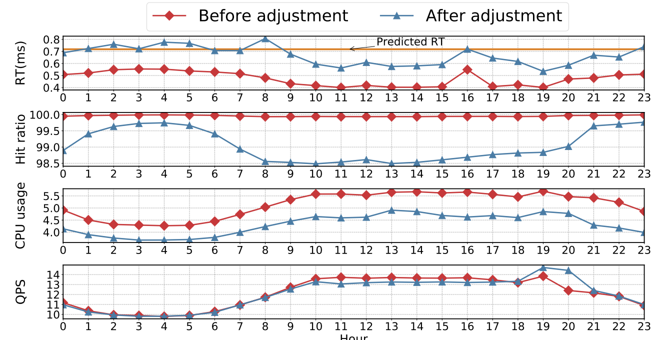

Figure 11: Performance comparison before and after adjustment

[1]Taobao is a customer-to-customer online retail service.
[2]Tmall is a business-to-consumer online retail service.
[3]Youku is a video hosting service.
[4]Fliggy is a travel ticket booking service.

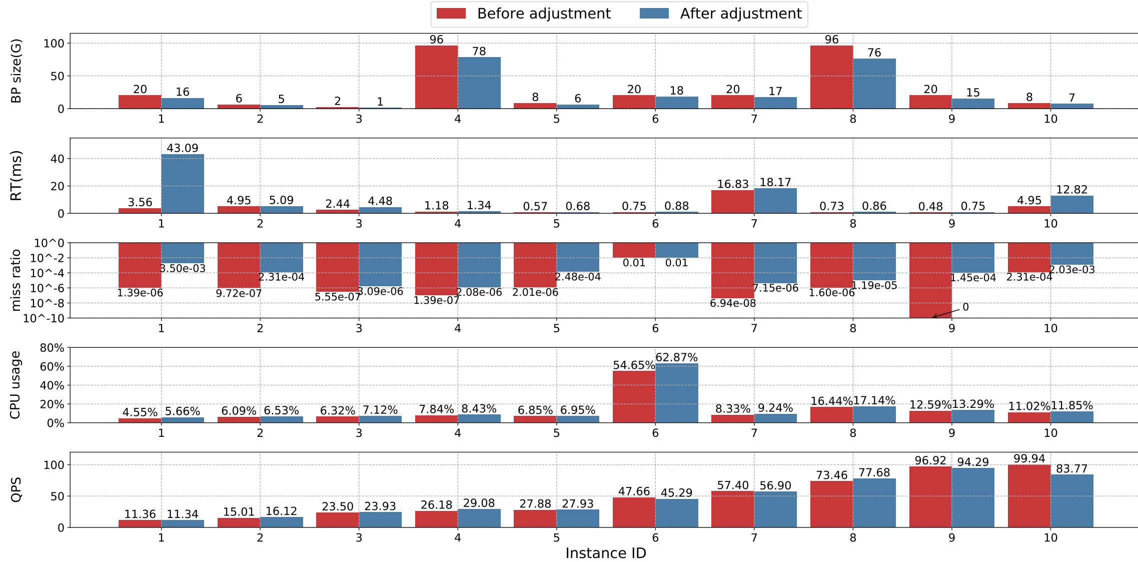

Figure 12: Performance illustration before and after adjustment

Figure 11 shows the result of a typical instance. Our algorithm adjusts the buffer pool size from 96GB to 86GB, about 10% reduction. We see that RT increases around 30% $\sim$ 50%, but the latency still keeps relatively low (under 1ms). Furthermore, most RT after adjustment is under and close to the predicted upper bound of the response time. This indicates that our algorithm predicts the upper bound of RT reasonably well (see the comparison with other different algorithms in Section 4.2.3). Meanwhile, the QPS remains almost the same before and after the adjustment. This is exactly our goal to avoid the negative impact on the throughput when using a smaller buffer size. The hit ratio decreases from 100.0% to around 99.1%, which demonstrates that it is unnecessary to keep the hit ratio as high as 100.0% for many realistic applications. The performance (RT and QPS) still meets the quality of service after we reduce the buffer size.

To further visualize the overall performance, we randomly select 10 representative instances and plot their performance metrics in Figure 12. The memory saving ranges from 50% to 10%, which strongly supports that a single number does not fit all for the individualized buffer pool configuration. After the adjustments, all RT values are still acceptable, although their values do increase as expected. Most of the CPU utilizations still remain low (under 20%) although their average value relatively increases by 10%.

We briefly explain why instance 1 has a large increase in RT after the adjustment. Its QPS is low (around 11 queries/s) before and after the adjustment. We find that there is one query that consumes 99.97% of the total response time. The lookup value in WHERE condition changes for this query. Thus, the logical read, physical read, the rows of examined increase dramatically after the adjustment. Resultantly, the response time increases from 3.56 ms to 43.09 ms. But it does not trigger a rollback, since it still satisfies SLA constraints and no slow SQL anomalies are found.

Last, we use the statistics of the experimental results from the $1,000$ instances to characterize the results after adjusting the buffer pool sizes. We analyze the average percentage of saved memory for each of the 9 sizes in Table 4. The savings vary from 10% to 50%, which verify that we need to use individualized numbers to tune different instances. For example, the instances of sizes 20G and 8G account for the majority (over 80%) of the instances (see Figure 10). Their saving ratios are 15% and 13%, respectively.

Table 4: Average memory saving ratios for different sizes

Before change	400G	188G	168G	96G	64G	45G	20G	8G	2G
After change	259G	167G	152G	76G	48G	33G	17G	7G	1G
Saving ratio	35%	11%	10%	21%	25%	27%	15%	13%	50%

In Table 5, RT_{pred} and RT_{ob} are the RT values predicted by the model and observed online, respectively. Correspondingly, MR_{pred} and MR_{ob} are the miss ratios chosen by the model and observed in the real system, respectively. After shrinking the buffer pool size, both the miss ratio and the RT should increase but RT_{ob} are expected to be smaller than $RT_{predict}$ as the latter is an upper bound. Also, all these predicted upper bounds should be within the safe limit. Our experiments show that indeed all the predicted values are within the safe limit. In addition, most of the observed results (more than 83% for RT and 85% for MR) are consistent with the predictions. A small number of the results (11.7% $\sim$ 16.7% for RT and 11.7% $\sim$ 16.7% for MR) are different from the predictions. These RT/MR values that are higher than expected can potentially trigger the rollbacks, but we did not receive the negative feedback from the business units. Therefore, these instances are acceptable to the applications in our production environment. After the adjustment, the total memory saving is 17.2%.

Table 5: Online performance

Metrics		**Workday**	**Holiday**
RT	$RT_{pred} > RT_{ob}$	83.3%	88.3%
	$RT_{pred} < RT_{ob}$	16.7%	11.7%
MR	$MR_{pred} < MR_{ob}$	86.7%	85.0%
	$MR_{pred} > MR_{ob}$	13.3%	15.0%
BP	Sum (before change)	14458G	
	Sum (after change)	11976G	
	Memory Saving	**17.2%**	

4.2.2 Offline evaluation of RT prediction algorithm

To quantify the accuracy of the pairwise DNN for predicting the response times, we carefully design an offline test by building a benchmark using real measurements. The training data set is the hourly database metrics measured in a time period of 7 days before the holiday season. The testing set is the response times measured on one particular date

during the holiday season. The average of the hourly RT varies by more than 70% on the training and testing data sets due to workload change. It makes the RT prediction difficult. Therefore, it is a representative test for evaluating the generalization capability of our prediction model.

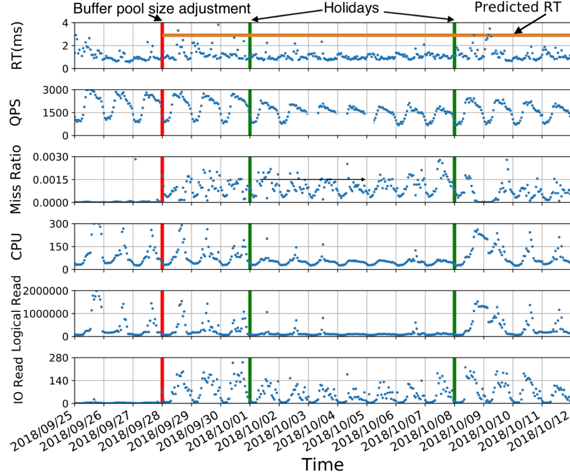

Figure 13: Performance during holidays and workdays

To visualize the variation across the holiday season as well as the relationship between the performance metrics and RT, we select a sample instance, whose buffer size is changed from 45G to 21G. In Figure 13, the plots before and after the red line compare the system performance before and after the buffer pool size is adjusted. The results on holidays are between the first and second green lines in Figure 13. The QPS and logical reads decrease significantly during the holidays compared with workdays. This immediately leads to lower RT and miss ratios, which explains why the results during holidays are better than workdays in Table 5. Each point represents a mean value of the metric within one hour. It is clear that the I/O reads increase notably after changing BP while the logical reads and QPS only experience slight variations. Resultantly, both the miss ratio and RT increase as expected. Note that the RT after changing BP is smaller than our prediction. The mean value of RT is 1.3ms and the predicted RT is 2.9ms, as an upper bound is computed.

We form the pairwise training set as described in Section 3.4.3. Then, we use the metrics (RT excluded) from the testing set as left sample and the metrics from the same instance in the same hour of the last date of the training set as right sample, which form the input to the pairwise DNN model. The output of the model is the predicted RT.

The performance of RT prediction is evaluated by the adjusted mean relative absolute error (AMRAE), mean absolute error (MAE) and underestimation mean absolute error (UMAE), as defined below. Notice that AMRAE and MAE are the overall performance metrics, while UMAE measures the severity of underestimation. For the same reason as using ReL1 as the loss function, AMRAE is the main metric for consideration and MAE and UMAE are supplemental.

$$\text{AMRAE} = \frac{1}{N_t} \sum_{i=1}^{N_t} \left| \frac{y_i - \hat{y}_i}{y_i + 0.1} \right| \times 100\% \tag{7}$$

$$\text{MAE} = \frac{1}{N_t} \sum_{i=1}^{N_t} |y_i - \hat{y}_i| \tag{8}$$

$$\text{UMAE} = \frac{1}{N_t} \sum_{i=1}^{N_t} |y_i - \hat{y}_i| \, I(y_i - \hat{y}_i > 0) \tag{9}$$

where N_t is the sample size of the testing set, y_i is the actual RT and $\hat{y}_i$ is the predicted RT of the i-th sample.

We compare the performance of PW-DNN-1/2/3 (pairwise DNN with k_{pair} =1, 2 and 3, respectively), I2V-DNN (Instance2vec DNN) and DNN (regular DNN) with different loss functions, e.g., ReL1 (defined in Eq. 5), L1 (L1 norm or MAE) and L2 (L2 norm or MSE) and different values of λ in Eq. 4 for asymmetric loss function, in Table 6 and 7. We make the following observations:

- From row 1 and 2 in both tables, we see that reducing λ improves UMAE at the cost of larger AMRAE.
- Results from row 2, 3 and 4 in both tables show that ReL1 loss function is most effective for minimizing AMRAE.
- Row 1, 5 and 6 compare pairwise DNN model with different values of k_{pair} and we can conclude that k_{pair} must be at least 2 for good AMRAE result. Notice that training error naturally gets worse when we increase k_{pair} to add more pairs of different instances to the training set.
- It is shown that embedding improves RT prediction for I2V-DNN by comparing row 7 and 8.
- The training error is small in row 5 and 7, while the corresponding testing error is much larger. But when we add information from nearest neighboring instances, the training and testing errors are much closer (see row 1, 2 and 6). We conclude that the pairwise DNN with pairwise training samples has a stronger generalization capability.
- Row 9 in Table 7 is when we directly use RT of the last day in the training set as the predicted RT for the corresponding instance in the testing set. We can see that the resulting AMRAE is large, indicating large RT variations before and during the holidays.

Table 6: Training set performance (%)

#	Model	Loss	λ	AMRAE	MAE	UMAE
1	PW-DNN-2	ReL1	0.4	31.53	1.51	1.34
2	PW-DNN-2	ReL1	1	28.42	1.64	1.56
3	PW-DNN-2	L2	1	210.15	2.07	0.61
4	PW-DNN-2	L1	1	44.68	1.27	0.96
5	PW-DNN-1	ReL1	0.4	14.38	0.56	0.31
6	PW-DNN-3	ReL1	0.4	38.06	1.93	1.76
7	I2V-DNN	ReL1	0.4	24.10	0.88	0.49
8	DNN	ReL1	0.4	49.00	2.48	2.35

Table 7: Testing set performance (%)

#	Model	Loss	λ	AMRAE	MAE	UMAE
1	PW-DNN-2	ReL1	0.4	34.58	1.08	0.77
2	PW-DNN-2	ReL1	1	31.83	1.18	0.98
3	PW-DNN-2	L2	1	124.96	1.60	0.42
4	PW-DNN-2	L1	1	45.69	1.04	0.58
5	PW-DNN-1	ReL1	0.4	40.28	1.02	0.55
6	PW-DNN-3	ReL1	0.4	34.96	1.24	0.95
7	I2V-DNN	ReL1	0.4	43.00	1.11	0.53
8	DNN	ReL1	0.4	46.65	2.03	1.89
9	Direct guess	-	-	76.05	1.31	0.50

As shown in Table 6, I2V-DNN achieves good performance on the training set, but its errors on the testing set are much larger as shown in Table 7, implying a weak generalization capability. On the other hand, the pairwise DNN model

exhibits good performance on both the training and testing sets, with smaller errors than I2V-DNN on the testing set. One explanation is that pairing similar instances better characterizes the variations to predict the change in RT.

Based on the observations above, the final model we use in our production system is PW-DNN-2 with ReL1 loss function (row 1 and 2), which has the smallest AMRAE value. The tuning factor λ is set to 0.4 for a good tradeoff between minimizing AMRAE and keeping underestimation low.

4.2.3 *RT prediction compared with other algorithms*

We compare our DNN model (PW-DNN-2) with the other algorithms introduced in Section 3.4.4. Figure 14 plots the results of AMRAE, MAE, and UMAE. They are all combined with the pairwise technique. Our model outperforms all the other algorithms on AMRAE and MAE. For UMAE, XGB is the best. However, XGB has bad performance on AMRAE and MAE. The reason why PW-DNN-2 is relatively high for UMAE is because it is conservative in predicting the RT upper bound. For AMRAE, all the afore-mentioned algorithms are significantly worse than our model. Specifically, the best AMRAE among them is over 62%, which is unacceptable in the production environment.

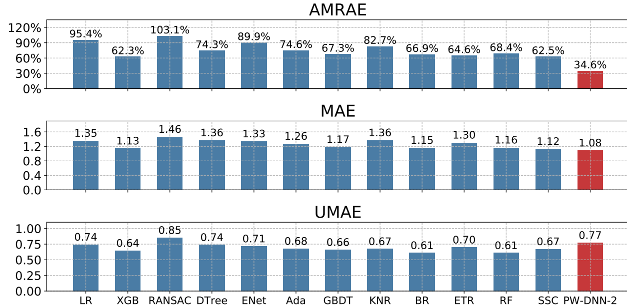

Figure 14: AMRAE, MAE and UMAE of different algorithms

5. RELATED WORK

Database parameter tuning has been an active area in recent years. Andy Pavlo et al. proposed a framework [31] for self-driving DBMS including several key components, like runtime architecture, workload modeling and control framework. They extended the details of this framework to automatically tune DBMS knob configurations, called OtterTune [42]. OtterTune uses a LASSO algorithm to select the most impactful knobs and recommends knob settings based on Gaussian Processes. OtterTune uses many (more than hundreds) metrics of different configurations to train the model. OtterTune's objective is to achieve a good performance of a single DBMS instance by tuning important parameters in the configuration file of a DBMS kernel while our goal is to optimize memory usage by tuning buffer pool sizes of many different database instances.

Others have looked at scaling system by modeling DBMS workload [27, 39, 9]. They use system metrics, like QPS of different queries, to predict the future workload. However, they mainly focused on workload estimation or prediction rather than system tuning.

P-Store [39] scales the DBMS by using a time-series model to predict the load for different applications. It utilizes a dynamic programming algorithm to reconfigure the database. The purpose of P-Store is different from this paper. Our idea is to optimize the buffer pool management of a large number of DBMS instances, by training models to save the memory usage without negatively impacting these database instances' performance.

In a previous study, IBM developed a Self-Tuning Memory Manager (STMM) system for different components of memory consumers in DB2 [37]. The feedback controller requires frequent changes of the buffer pool sizes, ranging from 30 seconds to 10 minutes per change in DB2's STMM. This is prohibited in our setting due to the stability requirement of OLTP DBMS services. Since OLTP services are mission-critical, DBAs are very cautious at each online parameter adjustment. At Alibaba, typically buffer adjustment frequency is on weeks. MT-LRU algorithm [28] was developed for multi-tenants buffer pool memory sharing. Tran et al. proposed a buffer miss ratio equation to tune the buffer pool sizes, and validated with different buffer replacement policies [41]. These methods are based on heuristic that are different from what we proposed in this paper. In our algorithm, we rely on large deviation analysis to characterize missed requests that happen with a small probability. We infer a tolerable buffer miss ratio to satisfy the service level agreement of response times after changing buffer pool sizes. In our system, response time prediction is used as a verification and provides reference information for monitoring.

Query execution time prediction, which is quite related to response time prediction, has been extensively studied in previous research. In [29], a transaction based statistical model is developed to predict the system performance with varying resources. An analytical model for query execution time prediction is proposed in [44]. Machine learning based single query prediction has been studied in [2, 13]. However, they focus on execution times rather than response times. Further more, the query-level prediction is not easy to be used for the response time of an instance. Our system directly predicts instance-level response times.

Lastly, there is an increasing amount of interest in using machine learning techniques to tune and optimize database performance from various aspects, e.g., index building [21], clock synchronization [15], SQL join optimization [22]. iBTune leverages similar approaches to tune system performance at scale.

6. CONCLUSION

In this paper, we propose iBTune to adjust DBMS buffer pool sizes by using a large deviation analysis for LRU caching models and leveraging the similar instances based on performance metrics to find tolerable miss ratios. To provide a level of protection, we build a pairwise deep neural network to predict the upper bounds of the response times. The adjustments are considered to be valid only when the estimations are within the safe limits that are inferred from the collected traces. The deployment on our large-scale production environment shows that this solution can save more than 17% memory resource compared to the original system that only relies on experienced DBAs.

Future work. This paper focuses on shrinking buffer pool sizes to reduce cost, which by far is the most important issue with our production deployment. On a different direction, for certain cases it is also necessary to increase the buffer pool sizes. Currently we rely on DBAs to manually analyze the system expanding requirements before taking important actions. We will explore how to automatically expand the buffer pools in the future.

7. REFERENCES

[1] Docker. https://www.docker.com.

[2] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In Proceedings of the 2012 IEEE 28th International Conference on Data Engineering, ICDE '12, pages 390–401, Washington, DC, USA, 2012. IEEE Computer Society.

[3] M. Arlitt and L. W. C.˙Internet web servers: Workload characterization and performance implications. In IEEE/ACM Transaction on Networking, October 1997.

[4] C. Berthet. Approximation of LRU caches miss rate: Application to power-law popularities. arXiv:1705.10738, 2017.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In Proceedings of the 18th Conference on Information Communications, 1999.

[6] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pages 785–794. ACM, 2016.

[7] F. J. Corbato. A paging experiment with the multics system. MIT Project MAC Report, MAC-M-384, 1968.

[8] G. Dán and N. Carlsson. Power-law revisited: Large scale measurement study of p2p content popularity. In Proceedings of the 9th International Conference on Peer-to-peer Systems, IPTPS'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[9] S. Das, F. Li, V. R. Narasayya, and A. C. König. Automated demand-driven resource scaling in relational database-as-a-service. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pages 1923–1934, New York, NY, USA, 2016. ACM.

[10] K. G. Derpanis. Overview of the ransac algorithm. Image Rochester NY, 4(1):2–3, 2010.

[11] E. Elhamifar and R. Vidal. Sparse subspace clustering: Algorithm, theory, and applications. IEEE transactions on pattern analysis and machine intelligence, 35(11):2765–2781, 2013.

[12] C. Fricker, P. Robert, and J. Roberts. A versatile and accurate approximation for lru cache performance. In Proceedings of the 24th International Teletraffic Congress, page 8. International Teletraffic Congress, 2012.

[13] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In Proceedings of the 2009 IEEE International Conference on Data Engineering, ICDE '09, pages 592–603, Washington, DC, USA, 2009. IEEE Computer Society.

[14] S. Garcia, J. Derrac, J. Cano, and F. Herrera. Prototype selection for nearest neighbor classification: Taxonomy and empirical study. IEEE transactions on pattern analysis and machine intelligence, 34(3):417–435, 2012.

[15] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 81–94, Renton, WA, 2018. USENIX Association.

[16] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. Machine learning, 63(1):3–42, 2006.

[17] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In Proceedings of the 2019 ACM International Conference on Management of Data, SIGMOD '19. ACM, 2019.

[18] P. J. Huber. Robust estimation of a location parameter. Ann. Math. Statist., 35(1):73–101, 03 1964.

[19] P. R. Jelenković. Least-recently-used caching with Zipfs law requests. In The Sixth INFORMS Telecommunications Conference. Boca Raton, Florida, 2002.

[20] A. Kadiyala and A. Kumar. Applications of python to evaluate the performance of bagging methods. Environmental Progress & Sustainable Energy, 37(5):1555–1559, 2018.

[21] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In SIGMOD, pages 489–504, 2018.

[22] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to Optimize Join Queries With Deep Reinforcement Learning. ArXiv e-prints, Aug. 2018.

[23] L. Lamport. The part-time parliament. ACM Transactions on Computer Systems (TOCS), 16(2):133–169, 1998.

[24] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '99, pages 134–143, New York, NY, USA, 1999. ACM.

[25] Z. L. Li, M. C.-J. Liang, W. He, L. Zhu, W. Dai, J. Jiang, and G. Sun. Metis: Robustly tuning tail latencies of cloud systems. In ATC (USENIX Annual Technical Conference). USENIX, July 2018.

[26] A. Liaw, M. Wiener, et al. Classification and regression by randomforest. R news, 2(3):18–22, 2002.

[27] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, pages 631–645, New York, NY, USA, 2018. ACM.

[28] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. PVLDB, 8(7):726–737, 2015.

[29] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting dbms. In 13th IEEE International Symposium on

Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pages 239–248, Sept 2005.

[30] E. J. O'neil, P. E. O'neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. ACM SIGMOD Record, 22(2):297–306, 1993.

[31] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In Proceedings of the 2017 Conference on Innovative Data Systems Research, CIDR '17, 2017.

[32] J. Petrovic. Using Memcached for data distribution in industrial environment. In Proceeding ICONS '08 Proceedings of the Third International Conference on Systems, pages 368–372, April 2008.

[33] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. ACM Computing Surveys (CSUR), 35(4):374–398, Dec. 2003.

[34] L. Rokach and O. Z. Maimon. Data mining with decision trees: theory and applications, volume 69. World scientific, 2008.

[35] D. L. Shrestha and D. P. Solomatine. Experiments with adaboost. rt, an improved boosting scheme for regression. Neural computation, 18(7):1678–1710, 2006.

[36] Y. Smaragdakis, S. Kaplan, and P. Wilson. The eelru adaptive replacement algorithm. Perform. Eval., 53(2):93–123, July 2003.

[37] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in db2. In Proceedings of the 32Nd International Conference on Very Large Data Bases, VLDB '06, pages 1081–1092. VLDB Endowment, 2006.

[38] T. Sugimoto and N. Miyoshi. On the asymptotics of fault probability in least-recently-used caching with Zipf-type request distribution. Random Structures & Algorithms, 29(3):296–323, 2006.

[39] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade. P-store: An elastic database system with predictive provisioning. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, pages 205–219, New York, NY, USA, 2018. ACM.

[40] J. Tan, G. Quan, K. Ji, and N. Shroff. On resource pooling and separation for LRU caching. In Proceedings of the 2018 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science. ACM, 2018.

[41] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung. A new approach to dynamic self-tuning of database buffers. Trans. Storage, 4(1):3:1–3:25, May 2008.

[42] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, pages 1009–1024, New York, NY, USA, 2017. ACM.

[43] J. Wang. A survey of web caching schemes for the internet. SIGCOMM Computer Communication Review, 29(5):36–46, Oct. 1999.

[44] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. PVLDB, 6(10):925–936, 2013.

[45] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Characterizing facebook's memcached workload. IEEE Internet Computing, 18(2):41–49, 2014.

[46] Y. Yang and J. Zhu. Write skew and zipf distribution: Evidence and implications. ACM Trans. Storage, 12(4):21:1–21:19, June 2016.

[47] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In Proceedings of the 18th ACM conference on Information and knowledge management, pages 2061–2064. ACM, 2009.

[48] H. Zou and T. Hastie. Regularization and variable selection via the elastic net. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 67(2):301–320, 2005.