An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems

Dana Van Aken[a], Dongsheng Yang[b], Sebastien Brillard[c], Ari Fiorino[a]
Bohan Zhang[d], Christian Bilien, Andrew Pavlo[a]
[a]Carnegie Mellon University, [b]Princeton University, [c]Société Générale, [d]OtterTune
dvanaken@cs.cmu.edu

ABSTRACT

Modern database management systems (DBMS) expose dozens of configurable knobs that control their runtime behavior. Setting these knobs correctly for an application's workload can improve the performance and efficiency of the DBMS. But because of their complexity, tuning a DBMS often requires considerable effort from experienced database administrators (DBAs). Recent work on automated tuning methods using machine learning (ML) have shown to achieve better performance compared with expert DBAs. These ML-based methods, however, were evaluated on synthetic workloads with limited tuning opportunities, and thus it is unknown whether they provide the same benefit in a production environment.

To better understand ML-based tuning, we conducted a thorough evaluation of ML-based DBMS knob tuning methods on an enterprise database application. We use the OtterTune tuning service to compare three state-of-the-art ML algorithms on an Oracle installation with a real workload trace. Our results with OtterTune show that these algorithms generate knob configurations that improve performance by 45% over enterprise-grade configurations. We also identify deployment and measurement issues that were overlooked by previous research in automated DBMS tuning services.

PVLDB Reference Format:
Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien,Andrew Pavlo. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. PVLDB, 14(7): 1241-1253, 2021.
doi:10.14778/3450980.3450992

1 INTRODUCTION

Since the 1970s, there have been several efforts to automate the tuning of database management systems (DBMSs). The first were "self-adaptive" DBMSs that used recommendation tools to help with physical database design (e.g., indexes [20], partitioning [21]). In the early 2000s, "self-tuning" systems expanded the scope of the problem to include automatic knob configuration. These knobs are tunable options that control nearly all aspects of the DBMS's runtime behavior. Researchers began to explore this problem because DBMSs have hundreds of knobs, and a database administrator (DBA) cannot reason about how to tune all of them for each application.

Unlike physical database design tools [11], knob configuration tools cannot use the built-in cost models of the DBMS's query optimizers. There are two ways that these tools automatically tune knobs. The first is to use heuristics (i.e., static rules) that the tool developers manually create [1, 5, 14, 27, 42]. Previous evaluations with open-source DBMSs (Postgres, MySQL) showed that these tools improve their throughput by 2–5× over their default configuration for OLTP workloads [38]. These are welcomed improvements, but there are additional optimizations that the tools failed to achieve. This is partly because they only target the 10–15 knobs that are thought to have the most impact. It is also because the rules do not capture the nuances of each workload that are difficult to codify.

The second way to tune a DBMS's knobs is to use machine learning (ML) methods that devise strategies to configure knobs without using hardcoded rules [17, 28, 38, 40]. In the same evaluation for OLTP workloads from above [38], an ML-based tool achieves 15–35% better throughput than static tools for Postgres and MySQL. The reason for this improvement is twofold: first, the ML algorithms consider more knobs during a tuning session than the rule-based tools. Second, they also handle the dependencies between knobs that are challenging for humans to reason about because their non-linear relationships vary with the DBMS's workload and hardware.

Recent results from ML-based approaches have demonstrated that they achieve better performance compared to human DBAs and other tuning tools on a variety of workloads and hardware configurations [28, 40]. But these evaluations are limited to (1) open-source DBMSs with limited tuning potential (e.g., Postgres, MySQL, MongoDB) and (2) synthetic benchmarks with uniform workload patterns. Additionally, although these evaluations used virtualized environments for the target DBMSs, to the best of our knowledge, they all used dedicated local storage (i.e., SSDs attached to the VM). Many real-world DBMS deployments, however, use non-local, shared-disk storage, such as on-premise SANs and cloud-based block stores. These non-local storage devices have higher read/write latencies and incur more variance in their performance than local storage. It is unclear how these differences affect the efficacy of ML-based tuning algorithms. Lastly, previous studies are vague about how much of the tuning process was truly automated. For example, they do not specify how they select the bounds of the knobs they are tuning. This means that the quality of the configurations may still depend on a human initializing it with the right parameters.

Given these issues, this paper presents a field study of automatic knob configuration tuning on a commercial DBMS with a real-world

Proceedings of the VLDB Endowment, Vol. 14, No. 7 ISSN 2150-8097.
doi:10.14778/3450980.3450992

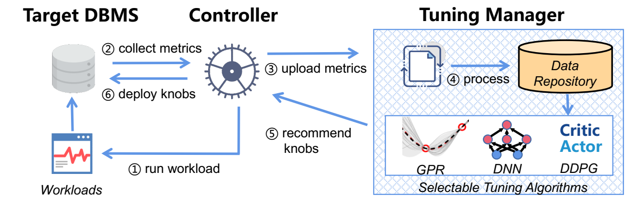


Figure 1: OtterTune Architecture – The controller runs the workload on the target DBMS, collects its knobs and metrics, and uploads them to the tuning manager. The tuning manager processes the uploaded data and uses the selected algorithm to generate the next knob configuration. Finally, the controller installs the configuration on the target DBMS.

workload in a production environment. We provide an evaluation of state-of-the-art ML-based methods for tuning an enterprise Oracle DBMS (v12) instance running on virtualized computing infrastructure with non-local storage. We extended the **OtterTune** [3] tuning service to support three ML tuning algorithms: (1) Gaussian Process Regression (GPR) from OtterTune [38], (2) Deep Neural Networks (DNN) [4, 39], and (3) Deep Deterministic Policy Gradient (DDPG) from CDBTune [40]. We also present optimizations for OtterTune and its ML algorithms that were needed to support this study.

Our results show that ML-based tools generate knob configurations that achieve better performance than enterprise-grade configurations. We also found that the quality of the configurations from the ML algorithms (GPR, DDPG, DNN) are about the same on higher knob counts, but vary in their convergence times.

2 BACKGROUND

We first provide an overview of how an automated tuning service works using OtterTune as an example [3]. We then discuss the limitations of previous evaluations of such services and why a more robust assessment is needed to understand their capabilities.

2.1 OtterTune Overview

OtterTune is a tuning service that finds good settings for a DBMS's knob configuration [38]. It maintains a repository of data collected from previous tuning sessions and uses it to build models of how the DBMS responds to different knob configurations. It uses these models to guide experimentation and recommend new settings. Each recommendation provides the service with more data in a feedback loop for refining and improving the accuracy of its models.

OtterTune is made up of a *controller* and a *tuning manager*. The controller acts as an intermediary between the target DBMS and the tuning manager. It collects runtime data from the target DBMS and installs configurations recommended by the tuning manager. The tuning manager updates its repository and internal ML models with the information provided by the controller and then recommends a new configuration for the user to try.

At the start of a tuning session, the user specifies which metric should be the *target objective* for the service to optimize. The service retrieves these metrics either from (1) the DBMS itself via its query API, (2) a third-party monitoring service (e.g., Prometheus, Druid), or (3) a benchmarking framework [16].

The service then begins the first tuning iteration. As shown in Figure 1, the controller ① executes the target workload on the DBMS. After the workload finishes, the controller ② collects the runtime metrics and configuration knobs from the DBMS and ③ uploads them to the tuning manager. OtterTune's tuning manager ④ receives the latest result from the controller and stores it in its repository. Next, the tuning manager ⑤ uses its ML models to generate the next knob configuration and returns it to the controller. The controller ⑥ applies the knob configuration to the DBMS and starts the next tuning iteration. This loop continues until the user is satisfied with the improvements over the original configuration.

2.2 Motivation

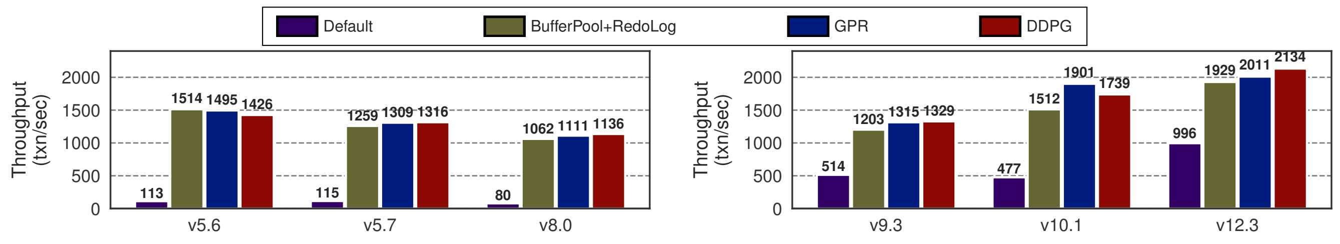
Recent studies on automated DBMS tuning services show that they can generate configurations that are equivalent to or exceed those created by expert DBAs [38, 40]. Despite these measurable benefits, we observe a mismatch between aspects of the evaluations of previous research on ML-based tuning approaches versus what we see in real-world DBMS deployments [28, 38, 40]. The three facets of this discrepancy are the (1) workload, (2) DBMS, and (3) operating environment. We now discuss them in further detail.

Workload Complexity: Gaining access to production workloads to evaluate new research ideas is non-trivial due to privacy constraints and other restrictions. Prior studies evaluate their techniques using synthetic benchmarks; the most complex benchmark used to evaluate ML-based tuning techniques to date is the TPC-C OLTP benchmark from the early 1990s. But previous studies have found that some characteristics of TPC-C are not representative of real-world database applications [23, 25]. Many of the unrealistic aspects of TPC-C are due to its simplistic database schema and query complexity. Another notable difference is the existence of temporary and large objects in production databases. Some DBMSs provide knobs for tuning these objects (e.g., Postgres, Oracle), which have not been considered in prior work.

System Complexity: The simplistic nature of workloads like TPC-C means that there are fewer tuning opportunities in some DBMSs, especially for the two most common DBMSs evaluated in previous studies (i.e., MySQL, Postgres). For these two DBMSs, one can achieve a substantial portion of the performance gain from configurations generated by ML-based tuning algorithms by setting *two knobs* according to the DBMS's documentation. These two knobs control the amount of RAM for the buffer pool cache and the size of the redo log file on disk.[1, 2]

To illustrate this issue, we ran a series of experiments on multiple versions of MySQL (v5.6, v5.7, v8.0) and Postgres (v9.3, v10.1, v12.3) using the TPC-C workload. We deployed the DBMSs on a machine running Ubuntu 18.04 with an Intel Core i7-8650U CPU (8 cores @ 1.90GHz, 2× HT) and 32 GB RAM. For each DBMS version, we measure the system's throughput under four knob configurations: (1) the OS's default configuration, (2) the recommended settings from the DBMS's documentation for the two knobs that control the buffer pool and redo log file sizes, (3) the configuration generated by OtterTune using GPR, and (4) the configuration generated by OtterTune using DDPG. We allowed OtterTune to tune 10 knobs for both DBMSs as selected by the tuning manager's ranking algorithm. We discuss the details of these algorithms in Section 4.

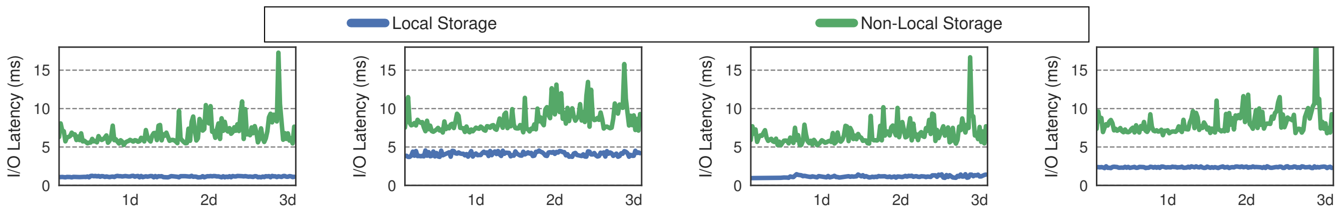
[1] **Postgres Knobs** – `SHARED_BUFFERS, MAX_WAL_SIZE`
[2] **MySQL Knobs** – `INNODB_BUFFER_POOL_SIZE, INNODB_LOG_FILE_SIZE`



(a) MySQL

(b) Postgres

Figure 2: DBMS Tuning Comparison – Throughput measurements for the TPC-C benchmark running on three versions of MySQL (v5.6, v5.7, v8.0) and Postgres (v9.3, v10.1, v12.3) using the (1) default configuration, (2) buffer pool & redo log configuration, (3) GPR configuration, and (4) DDPG configuration.



(a) Sequential Reads

(b) Sequential Writes

(c) Random Reads

(d) Random Writes

Figure 3: Operating Environment – I/O latency of local versus non-local storage for four different I/O workloads over a three-day period.

Figure 2 shows that the two-knob configuration and OtterTune-generated configurations improve the performance for TPC-C over the DBMS's default settings. This is expected since the default configurations for MySQL and Postgres are based on their minimal hardware requirements. More importantly, however, the configurations generated by ML algorithms achieve only 5–25% higher throughput than the two-knob configuration across the different versions of MySQL and Postgres. That is, one can achieve 75–95% of the performance obtained by ML-generated configurations by tuning only two knobs for the TPC-C benchmark.

Operating Environment: Disk speed is often the most important factor in a DBMS's performance. Although the previous studies used virtualized environments to evaluate their methods, to our knowledge, they deploy the DBMS on ephemeral storage that is physically attached to the host machine. But many real-world DBMS deployments use durable, non-local storage for data and logs, such as on-premise SANs and cloud-based block/object stores. The problem with these non-local storage devices is that their performance can vary substantially in a multi-tenant cloud environment [32].

To demonstrate this point, we measured the I/O latency on both local and non-local storage devices every 30 minutes over three days using `Fio` [2]. We conducted the local storage experiments on a machine with a Samsung 960EVO M.2 SSD. We ran the non-local storage experiments on a VM with virtual storage deployed on an enterprise private cloud. The results in Figure 3 show that the read/write latencies for the local storage are stable across all workloads. In contrast, the read/write latencies for the non-local storage are higher and more variable. The spike on the third day also demonstrates the unpredictable nature of non-local storage.

3 AUTOMATED TUNING FIELD STUDY

The above issues highlight the limitations in recent evaluations of configuration tuning approaches. These examples argue the need for a more rigorous analysis to understand whether real-world DBMS deployments can benefit from automated tuning frameworks. If automated tuning proves to be viable in these deployments, we seek to identify the trade-offs of ML-based algorithms and the extent to which human-guidance makes a difference.

We conducted an evaluation of the OtterTune framework at the **Société Générale** (SG) multi-national bank in 2020 [6]. SG runs most of their database applications on Oracle on private cloud infrastructure. They provide self-service provisioning for DBMS deployments that use a pre-tuned configuration based on the expected workload (e.g., OLTP vs. OLAP). These Oracle deployments are managed by a team of skilled DBAs with experience in knob tuning. Thus, the goal of our field study is to see whether automated tuning could improve a DBMS's performance beyond what their DBAs achieve through manual tuning.

In this section, we provide the details of our deployment of OtterTune at SG. We begin with a description of the target database workload and how it differs from synthetic benchmarks. We then describe SG's operating environment and the challenges we had to overcome with running an automated tuning service.

3.1 Target Database Application

The data and workload trace that we use in our study came from an internal issue tracking application (TicketTracker) for SG's IT infrastructure. The core functionality of TicketTracker is similar to other widely used project management software, such as Atlassian Jira and Mozilla Bugzilla. This application keeps track of work tickets submitted across the entire organization. SG has ∼140,000 employees spread across the globe [6], and thus TicketTracker's workload patterns and query arrival rate are mostly uniform 24-hours a day during the work week. SG currently runs TicketTracker on Oracle v12.1. We developed custom reporting tools to summarize the contents of the database and query trace. We now provide a high-level description of TicketTracker from this analysis.

Operator Type	% of Queries
`TABLE ACCESS BY INDEX ROWID`	31%
`INDEX RANGE SCAN`	23%
`INDEX UNIQUE SCAN`	16%
`SORT ORDER BY`	8%
`TABLE ACCESS FULL`	5%
All Others	17%

Table 1: Query Plan Operators – The percentage of queries in the TicketTracker workload that contain each operator type.

Database: We created a snapshot of the TicketTracker database from its production server using the Oracle Recovery Manager tool. The total uncompressed size of the database on disk is ~1.1 TB, of which 27% is table data, 19% is table indexes, and 54% is large objects (LOBs). This LOB data is notable because Oracle exposes knobs that control how it manages LOBs, and previous work has not explored this aspect of DBMS tuning.

The TicketTracker database contains 1226 tables, but 773 of them are empty tables from previous staging and testing efforts. We exclude them from our analysis here as no query accesses them. For the remaining 453 tables with data, the database contains 1647 indexes based on them. The charts in Figure 4 provide breakdowns of the number of tuples, columns, and indexes per table. Figure 4b shows that most of the tables have 20 or fewer columns. There is also a large percentage of tables that only have a single index; these are mostly tables with a small number of tuples (i.e., <10k).

Workload: We collected the TicketTracker workload trace using Oracle's Real Application Testing (RAT) tool. RAT captures the queries that the application executes on the production DBMS instance starting at the snapshot. It then supports replaying those queries multiple times on a test database with the exact timing, concurrency, and transaction characteristics of the original workload [19]. Our trace is from a two-hour period during regular business hours and contains over 3.6m query invocations.

The majority of the queries (90.7%) that TicketTracker executes are read-only `SELECT` statements. They are short queries that access a small number of tuples. Figure 5a shows that the average execution time of `SELECT` queries with SG's default configuration is 25 ms. The application executes some longer-running queries, but these are rare. The 99th-tile latency for `SELECT` queries is only 370 ms.

We also counted the number of times that a `SELECT` query accesses each table. Only 2% of the queries perform a join between two or more tables; the remaining 98% only access a single table. The histogram in Figure 5b shows the top 10 most accessed tables in the workload. The remaining tables are accessed by 1% or less of the queries. These results indicate that there is no single table that queries touch significantly more than others.

Since the workload trace includes query plans, we extracted the operators for each `SELECT` query to characterize their behavior. This analysis helped us understand whether the configurations selected by the algorithms in our experiments would even affect the queries. Table 1 provides a ranked list of the five most common operators. We see that almost all the queries perform index look-ups and scans. The most common operator (`TABLE ACCESS BY INDEX ROWID`) is when the query uses a non-covering index to get a pointer to the tuple. Only 5% of the queries execute a sequential scan on a table.

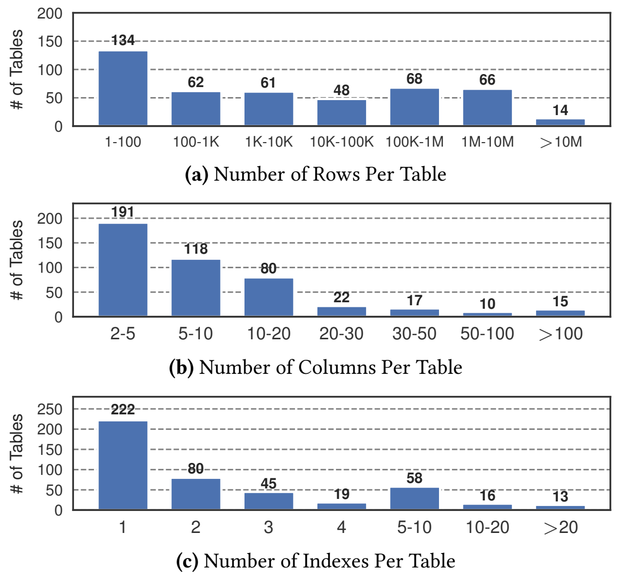


Figure 4: Database Contents Analysis – The number of tuples, columns, and indexes per table for the TicketTracker database.

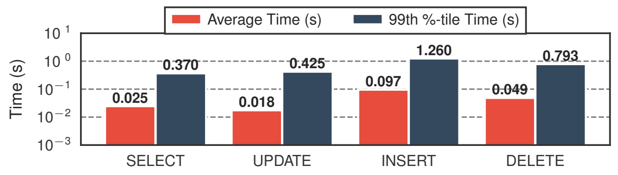
The rest of the TicketTracker workload contains `UPDATE` (5.2%), `INSERT` (3.4%), and `DELETE` (0.7%) queries. The average execution times of these queries are 18 ms, 97 ms, and 49 ms, respectively. For `INSERTs`, Figure 5a shows that some queries take 1260 ms to run. Our analysis also shows that a large portion of the modification queries are on tables with over 100k tuples. Some of the largest tables (i.e., >10m tuples) are never used in `SELECT` queries.

There are important differences in the TicketTracker application compared to the TPC-C benchmark used in previous ML tuning evaluations. Foremost is that the TicketTracker database has hundreds of tables and the TPC-C database only has nine. TPC-C also has a much higher write ratio for queries (46%) than the TicketTracker workload (10%). This finding is consistent with previous work that has compared TPC-C with real-world workloads [23, 25]. Prior to our study, it was unknown whether these differences affect the efficacy of ML-based tuning algorithms.

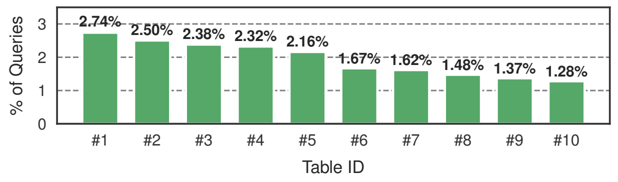
3.2 Deployment

We deployed five copies of the TicketTracker database and workload on separate Oracle v12.2 installations in SG's private cloud. We used the same hardware configuration as the production instance. Each DBMS instance runs on a VM with 12 vCPUs (Intel Xeon CPU E5-2697v4 at 2.30 GHz) and 64 GB RAM. We configured the VMs to write to a NAS shared-disk running in the same data center. As shown in our previous experiment in Figure 3, the average read and write latencies for this storage are ~6.7 ms and ~8.3 ms, respectively.

The initial knob configuration for each Oracle instance is selected from a set of pre-tuned configurations that SG uses for their entire fleet. The SG IT team provides their employees with a self-service web interface for provisioning new DBMSs. In addition to selecting the hardware configuration of a new DBMS (e.g., CPU cores, memory), a user must also specify the expected workload that the DBMS will support (e.g., OLTP, OLAP, HTAP). The provisioning system installs the knob configuration that has been pre-tuned by the SG



(a) Execution Time of Query Types (Log Scale)



(b) Top 10 Tables Accessed by Queries (%)

Figure 5: TicketTracker Workload Analysis – Execution information for the TicketTracker queries extracted from the workload trace.

administrators for the selected workload type. Although these configurations outperform Oracle's default settings, they only modify 4–6 knobs and are still not tailored to the individual applications' workloads. As such, for the TicketTracker workload, the DBA further customized some of the knobs in the pre-tuned configuration, including one that improves the performance of LOBs.[3]

We set up multiple of OtterTune's tuning managers and controllers in the same data center as the Oracle DBMSs. We ran each component in a Docker container with eight vCPUs and 16 GB RAM. Each DBMS instance has a dedicated OtterTune tuning manager assigned to it. This separation prevents one session from using training data collected in another session, which will affect the convergence rate and efficacy of the algorithms.

3.3 Tuning

At the beginning of each iteration in a tuning session, the controller first restarts its target DBMS instance. Restarting ensures that the knob changes that OtterTune made to the DBMS in the last iteration take effect. Some knobs in Oracle do not require restarting the DBMS, but changing them is not instantaneous and requires additional monitoring to determine when their updated values have been fully applied. To avoid issues with incomplete or inconsistent configurations, we restart the DBMS each time.

Another issue is that Oracle could refuse to start if one of its knobs has an invalid setting. For example, if one sets the knob that controls the buffer pool size[4] to be larger than the amount of physical memory on the underlying machine, then Oracle will not start and prints an error message in the log. If the controller detects this failure, it halts the tuning iteration, reports the failure to the tuning manager, and then starts a new iteration with the next configuration. This failure is still useful for the tuning algorithms; we discuss how to handle this and other failure scenarios in Section 6.

Once the DBMS is online and accepting connections, the controller resets the database back to what it was at the beginning of the workload trace (i.e., any tuple modified during the workload replay is reverted to its original state). Although the TicketTracker database is over 1 TB in size, this step takes on average five minutes per iteration because Oracle's snapshot tool only resets the pages modified since in the last iteration.

After resetting the DBMS, the controller executes a `Fio` [2] microbenchmark on the DBMS's VM to collect the current performance measurements for its shared disk. This step is not necessary for tuning, and none of the algorithms use this data in their models. Instead, we use these metrics to explain the DBMSs' performance in noisy cloud environments (see Section 6).

Now the controller begins the execution step on the target DBMS using the current configuration. It first retrieves the current values for DBMS's metrics through Oracle-specific SQL commands. Oracle generates over 3900 metrics that are a mix of counters and aggregates. We only collect global metrics from the DBMS (i.e., there are no table- or index-specific metrics). We set the tuning algorithm's target objective function to *DB Time* [15, 18]. This is an Oracle-specific metric that measures the total time spent by the database in processing user requests. A key feature of DB Time is that it provides a "common currency" to measure the impact of any component in the system. It is the SG DBAs' preferred metric because it allows them to reason about the interactions between DBMS components to diagnose problems.

OtterTune's controller executes TicketTracker's workload trace using Oracle RAT. We use RAT's automatic setup option to determine the number of client threads that it needs to replicate the same concurrency as the original application. We configure RAT to execute a 10-minute segment (230k queries) from the original trace. We limit the replay time for two reasons. First, the segment's timespan is based on the wall clock of when the trace was collected on the production DBMS. This means that when the trace executes on a DBMS with a sub-optimal configuration (which is often the case at the beginning of a tuning session), the 10-minute segment could take several hours to complete. We halt replays that run longer than 45 minutes. The second reason is specific to Oracle: RAT is unstable on large traces for our DBMS version. Oracle's engineers did provide SG with a fix, but only several months after we started our study, and therefore it was too late to restart our experiments.

After the workload execution completes, the controller collects the DBMS's metrics again, computes the delta for the counters from the start of the iteration, and then sends the results to the tuning manager. The controller then polls the tuning manager for the next configuration to install and repeats the above steps.

4 TUNING ALGORITHMS

Our goal is to understand how the DBMS configuration tuning algorithms proposed in recent years behave in a real-world setting and under what conditions one performs better than others. To this end, we extended OtterTune to support multiple algorithms in its tuning manager. This allows us to deploy a single platform without making major changes to the tuning pipeline.

We now describe the three algorithms that we evaluated: (1) Gaussian Process Regression (GPR), (2) Deep Neural Network (DNN), and (3) Deep Deterministic Policy Gradient (DDPG). Although there are other algorithms that use query data to guide the search process [28], they are not usable at SG because of privacy concerns since the queries contain user-identifiable data. Methods to anonymize this data are outside the scope of this paper.

[3]**Oracle Knob** – `DB_32K_CACHE_SIZE`
[4]**Oracle Knob** – `DB_CACHE_SIZE`

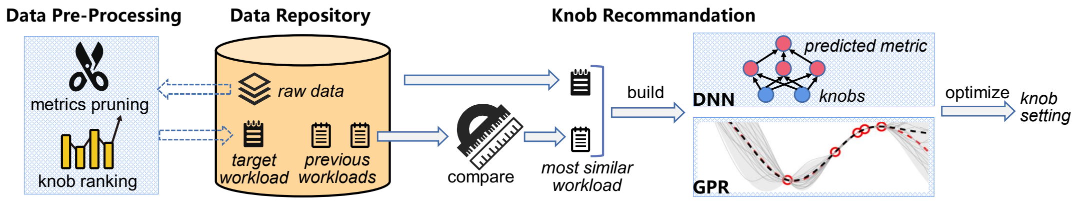


Figure 6: GPR/DNN Tuning Pipeline – The raw data for each previous workload is aggregated and compared with the target workload. Data from the most similar previous workload is then merged with the target workload data to build a machine learning model (DNN or GPR). Finally, the algorithm recommends the next configuration to run by optimizing the model.

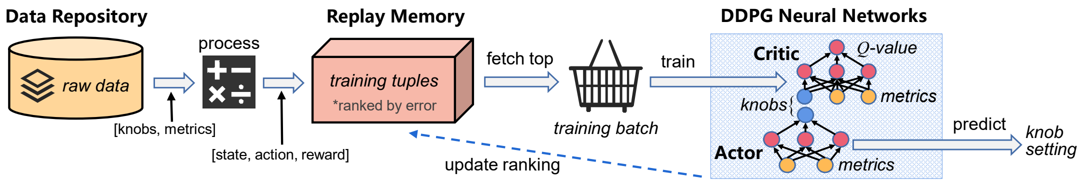


Figure 7: DDPG Tuning Pipeline – The raw data is converted to states, actions, and rewards and then inserted into the replay memory. The tuples in the replay memory are ranked by the error of the predicted Q-value. In the training process, the critic and actor are updated with a batch of the top tuples. After training, the prediction error in the replay memory is updated, and the actor recommends the next configuration to run.

4.1 GPR — OtterTune (2017)

Our implementation of GPR is based on the original algorithm supported by OtterTune [3]. It employs a Gaussian process as a prior over functions to calculate the distance between the test point and all the training points [31]. The algorithm uses the kernel functions to predict the value of the test point and the uncertainty. Figure 6 shows that OtterTune's GPR pipeline is comprised of two stages. The first is the *Data Pre-Processing* stage that prepares the knob and metric data in OtterTune's data repository. The second is the *Knob Recommendation* stage that selects values for the knobs.

Data Pre-Processing: This stage aims to reduce the dimensionality of the metrics and determine the most important knobs to tune. The service uses the output of this stage to generate knob configurations for the target DBMS. OtterTune runs this stage periodically in the background. Each invocation takes up to an hour, depending on the number of samples and DBMS metrics.

The Data Pre-Processing stage first identifies a subset of DBMS metrics that best capture the variability in performance and the distinguishing characteristics of a given workload. The algorithm uses a dimensionality reduction technique, *factor analysis*, to reduce the metrics to a smaller set of factors that capture the correlation patterns of the original variables. Each factor is a linear combination of the original variables, and their coefficients can be interpreted in the same way as the coefficients in linear regression. This means that one can order the factors by how much of the variability in the original data they explain. The algorithm then groups the factors with similar correlation patterns using *k-means* clustering. Lastly, the algorithm selects one representative metric from each group.

The stage then computes a ranked list of the knobs that have the greatest impact on the target objective function. It uses a feature selection method called *Lasso* [37], where the knob data is the input X, and the output y is the target objective data combined with the pruned metrics. Lasso identifies the most important knobs during the regression between X and y. To do this, it starts with a high penalty setting where all weights are zero, and thus no features are selected in the regression model. It then decreases the penalty in small increments, recomputes the regression, and tracks what features are added back to the model at each step. The order in which the knobs first appear in the regression determines how much impact they have on the target metric.

Knob Recommendation: This stage is responsible for generating a new configuration recommendation at the end of each iteration in the tuning session. The first step is to determine which of the workloads that OtterTune tuned in the past is the most similar to the current workload. It uses this previous data to "bootstrap" the new session. To do this, the algorithm uses the output data from the first stage to predict the metric values of the target DBMS's workload given the ranked listing of knobs.

The service then builds a GPR model with the data from both the target workload and the most similar workload. For the given array of knobs (x) as its input, the model outputs the pair (y,u) of the target objective value (y) and the uncertainty value (u). The algorithm calculates the *upper confidence bounds* (UCB) as the sum of y and u. It then performs gradient ascent on the UCB to find the knob settings expected to lead to a good objective value. It searches from random knob settings as starting points, performs gradient descent to find the local optimum from each starting point, and recommends the highest one among those local optima as the recommended knob configuration for the target DBMS.

An important issue in this process is how the algorithm manages the trade-off between exploration (i.e., collecting new information to improve the model) and exploitation (i.e., greedily trying to do well on the objective). OtterTune adjusts the weight of the uncertainty in UCB to control exploration and exploitation.

4.2 DNN — OtterTune (2019)

Previous research has argued that Gaussian process models do not perform well on larger data sets and high-dimensional feature vectors [26]. Given this, we modified OtterTune's original GPR-based algorithm described above to use a deep neural network (DNN) instead of the Gaussian models. As shown in Figure 6, OtterTune's DNN algorithm follows the same ML pipeline as GPR.

DNN relies on a deep learning algorithm that applies linear combinations and non-linear activations to the input. The network structure of the DNN model has two hidden layers with 64 neurons each. All of the layers are fully connected with *rectified linear units* (ReLU) as the activation function. We implemented a popular technique called *dropout regularization* to avoid overfitting the models and improve their generalization [35]. It uses a dropout layer between the two hidden layers with a dropout rate of 0.5. DNN also adds Gaussian noise to the parameters of the neural network during the knob recommendation step [30] to control the amount of exploration versus exploitation. OtterTune increases exploitation throughout the tuning session by reducing the scale of the noise.

4.3 DDPG — CDBTune (2019)

This method was first proposed by CDBTune [40]. DDPG is a deep reinforcement learning algorithm that searches for the optimal policy in a continuous action space environment. The ability to work on a continuous action space means that DDPG can set a knob to any value within a range, whereas other reinforcement learning algorithms, such as Deep-Q learning, are limited to setting a knob from a finite set of predefined values. We first describe CDBTune's DDPG, and then we present an extension to it that we developed to improve its convergence rate.

As shown in Figure 7, DDPG consists of three components: (1) *actor*, (2) *critic*, and (3) *replay memory*. The actor is a neural network that chooses an action (i.e., what value to use for a knob) based on the given states. The critic is a second neural network that evaluates the selected action based on the states. In other words, the actor decides how to set a knob, and then the critic provides feedback on this choice to guide the actor. In CDBTune, the critic takes the previous metrics and the recommended knobs as the input and outputs a Q-value, which is an accumulation of the future rewards. The actor takes the previous metrics as its input and outputs the recommended knobs. The replay memory stores the training data tuples ranked by the prediction error in descending order.

Upon receiving a new data point, CDBTune first calculates the reward by comparing the current, previous, and initial target objective values. For each knob k, DDPG constructs a tuple that contains (1) the array of previous metrics m_{prev}, (2) the array of current metrics m, and (3) the current reward value. The algorithm stores this tuple in its replay memory. It next fetches a mini-batch of the top-ranked tuples from the memory and updates the actor and critic weights via backpropagation. Lastly, it feeds the current metrics m into the actor to get the recommendation of the knobs k_{next}, and adds noise to k_{next} to encourage exploration.

We identified a few optimizations to CDBTune's DDPG algorithm that reduce the amount of training data needed to learn the representation of the Q-value. We call this enhanced version DDPG++. There are three core differences between these algorithms. First, DDPG++ uses the immediate reward instead of the accumulated future reward as the Q-value. The assumption is that each knob setting is only responsible for the DBMS's performance in the current tuning iteration and has no relationship to the performance in future iterations. Second, DDPG++ uses a simpler reward function that does not consider the previous or base target objective values. Thus, each reward is independent of the previous one. Lastly, upon getting a new result, DDPG++ fetches multiple mini-batches from the replay memory to train the networks to converge faster.

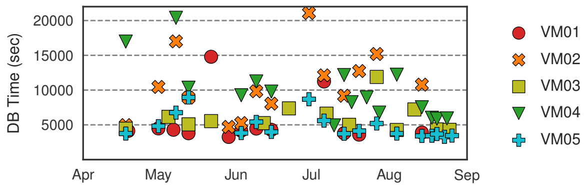


Figure 8: Performance Variability – Performance for the TicketTracker workload using the default configuration on multiple VMs over six months.

5 EVALUATION

We now present the results from our comparison of the above tuning algorithms for SG's Oracle installation on TicketTracker.

Random sampling methods serve as competitive baselines for judging optimization algorithms because they are simple yet surprisingly effective [9]. In our evaluation, we use a random sampling method called Latin Hypercube Sampling (LHS) [22] as a baseline. LHS is a space-filling technique that attempts to distribute sample points evenly across all possible values. Such techniques are generally more effective than naïve random sampling in high-dimensional spaces, especially when collecting a small number of samples relative to the total number of possible values.

We begin with an initial evaluation of the variability in the performance measurements for SG's environment. This discussion is necessary to explain how we conduct our experiments and analyze their results in the subsequent sections.

5.1 Performance Variability

Because each tuning session in our experiments takes multiple days to complete, we deployed the Oracle DBMS on multiple VMs to run the sessions in parallel. Our VMs run on the same physical machines during this time, but the other tenants on these machines or in the same rack may change. As discussed in Section 2.2, running a DBMS in virtualized environments with shared storage can lead to unexplained changes in the system's performance across instances with the same hardware allocations and even on the same instance.

To better understand the extent of this variability in SG's data center, we measured the performance of our VMs once a week over six months. We run the 10-minute segment of the TicketTracker workload using SG's default configuration. The results in Figure 8 show the DB Time metric for each VM instance over time. The first observation from this data is that the DBMS's performance on the same VM can fluctuate by as much as 4× even though the DBMS's configuration and workload are the same. For example, VM02's DB Time in July is higher than what we measured in the previous month. The next observation is that the relative performance of VMs can vary as well, even within a short time window.

We believe that these inconsistent results are due to latency spikes in the shared-disk storage. Figure 9 shows the DBMS's performance for one VM during a tuning session, along with its CPU busy time and I/O latency. These results show a correlation between spikes in the I/O latency (three highlighted regions) and degradation in the DBMS's performance. In this example, the algorithm had converged at this point of the tuning session, so the

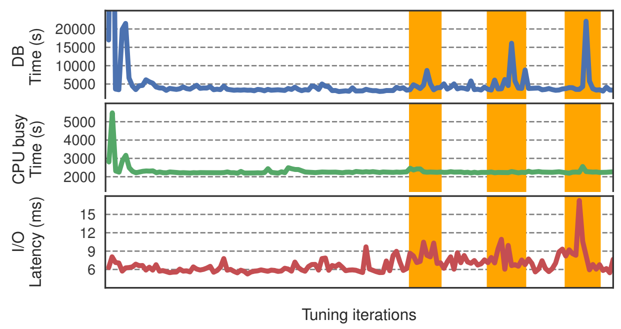


Figure 9: Effect of I/O Latency Spikes – Runtime measurements of DBMS performance with CPU utilization and I/O latency.

configuration was stable. Thus, it is likely that these latency spikes are due to external causes outside of the DBMS's control.

These fluctuations make our evaluation challenging since we cannot reliably compare tuning sessions that run on different VMs, or even the same VM but at different times. Given this, we made a substantial effort to conduct our experiments in such a way that we can provide meaningful analysis. We use the same procedure in all of our experiments in this paper. Each tuning session is comprised of 150 iterations. Every iteration can take up to one hour depending on the quality of the DBMS's configuration. As such, each session took three to five days to complete.

For a given experiment, we run three tuning sessions per algorithm under each condition being evaluated. We then collect the optimized configurations from all the sessions, along with the SG default configuration, and run them consecutively, three times each, on three different VMs. That is, we run each configuration a total of nine times – thrice per VM. Running the configurations sequentially in the same time period is necessary since a VM's performance varies over time. It also lets us use the same DB Time measurement for the SG default configuration to calculate their relative improvements. Running the configurations on three different VMs guards against one VM being especially noisy.

We select the performance of each configuration on a given VM as the median of the three runs. The overall performance of each configuration is the average across the three VMs. We report the minimum and maximum performance measurements from the three optimized configurations for each algorithm.

5.2 Tuning Knobs Selected by DBA

This first experiment evaluates the quality of the configurations that the tuning algorithms generate when increasing the number of knobs that they tune. Although Oracle exposes over 400 knobs, we limit the maximum number of knobs tuned to 40 for two reasons. First, we want to evaluate how much better the ML algorithms are at ranking the importance of knobs versus a DBA-selected ranking. Asking a human to select more than 40 knobs to tune is unrealistic and will produce random results. The second reason is to reduce the time that the algorithms need to converge because the more knobs there are, the harder it is to tune. Since each iteration of the TicketTracker workload takes up to 45 minutes, it would potentially take weeks for the models to converge. Hence, we consider a maximum of 40 knobs that the DBA selected and ordered based on their expected impact on the DBMS's performance.

Knob Name	Default	Best Observed
`DB_CACHE_SIZE`	4 GB	20–30 GB
`DB_32K_CACHE_SIZE`	10 GB	15 GB
`OPTIMIZER_FEATURES_ENABLE`	v11.2.0.4	v12.2.0.1

Table 2: Most Important Knobs – The three most important knobs for the TicketTracker workload with their default and best observed values.

For these experiments, the ML-based algorithms do not reuse data from previous tuning sessions. We instead bootstrap their models by executing 10 configurations generated by LHS.

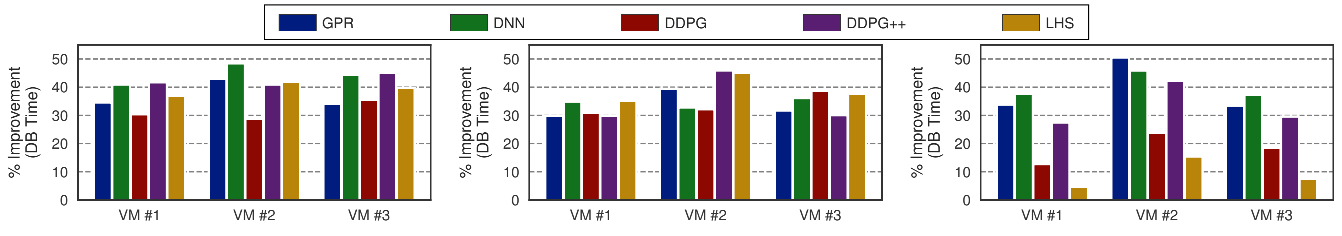
Figure 10 shows the improvement in DB Time over the SG default configuration achieved by the best (i.e., highest-performing) of three configurations generated per algorithm when optimizing 10, 20, and 40 knobs by VM. Although the absolute measurements vary, the algorithms' relative performance rankings are consistent across the VMs. Figure 11 shows the average performance improvement over the three VMs for the optimized configurations generated by the algorithms. The dark and light portions of each bar represent minimum and maximum performance per algorithm, respectively.

To understand why the configurations perform differently, we manually examined each configuration and identified three Oracle knobs that have the most impact when the algorithms fail to set them correctly. Table 2 shows the knobs' value in the SG default configuration and their best-observed value(s) from our experiments. The first two control the size of the DBMSs' main buffer caches. One of these caches is for the DBMS's 8 KB buffers for regular table data, and the other is for 32 KB buffers that the DBMS uses for LOB data. The third knob enables optimizer features based on an Oracle release; this is a categorical variable with seven possible values.

Figure 11 shows that the configurations recommended by DNN and DDPG++ that tune 10 knobs improve the DB Time by 45% and 43% over the default settings, respectively. Although LHS, GPR, and DDPG achieve over 35% better DB Time, they do not perform as well as DNN and DDPG++ because they select a sub-optimal version of the optimizer features to enable.

For the 20-knob configurations, Figure 11 shows that all the algorithms improve the DBMS's performance by 33–40% over the default configuration. Each algorithm, however, sets at least one of the important knobs in Table 2 incorrectly. This is because the tuning complexity increases with the number of knobs. We also see that DNN has the largest gap between its minimum and maximum optimized configurations. This is generally due to the randomness in the exploration of the algorithms and the amount of noise on the VM during a given tuning session.

As shown in Figure 11, the configurations from DNN and GPR achieve 40% better DB Time than the default configuration. DDPG and DDPG++ only achieve 18% and 32% improvement, respectively. The reason is that neither of them can fully optimize the 40 knobs within 150 iterations. DDPG++ outperforms DDPG because of the optimizations that help it converge more quickly (see Section 4.3). With more iterations, DDPG would likely achieve similar performance to the other ML-based algorithms. But due to computing costs and labor time, it was not practical to run a session for more than 150 iterations in our evaluation. The LHS configuration performs the worst of all, achieving only 10% improvement over the



(a) 10 Knobs **(b)** 20 Knobs **(c)** 40 Knobs

Figure 10: Tuning Knobs Selected by DBA (Per VM) – The performance improvement of the best configuration per algorithm running on separate VMs relative to the performance of the SG default configuration measured at the beginning of the tuning session.

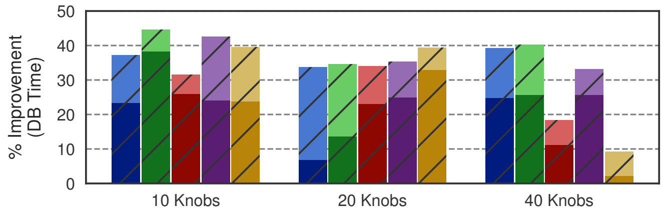


Figure 11: Tuning Knobs Selected by DBA – Performance measurements for 10, 20, and 40 knob configurations for the TicketTracker workload. The shading on each bar indicates the minimum and maximum performance of the optimized configurations from three tuning sessions.

default. This shows how sampling techniques like LHS can be inefficient for high-dimensional spaces.

In summary, we find that the configurations generated by all of the algorithms that tune 10, 20, and 40 knobs can improve the DBMS's performance over the default configuration. GPR always converges quickly, even when optimizing 40 knobs. But GPR is prone to getting stuck in local minima, and once it converges, it stops exploring and thus does not continue to improve after that point. The performance of GPR, therefore, depends on whether it explores the best-observed ranges of the impactful knobs from Table 2. We also observe that its performance is influenced by the initial samples executed at the start of the tuning session. This is consistent with findings from previous studies [26]. In contrast, DNN, DDPG, and DDPG++ require more data to converge and carry out more exploration. The configurations that tune 10 knobs perform the best overall. This is because the lower complexity of the configuration space enables DNN and DDPG++ to find good settings for the impactful knobs in Table 2.

5.3 Tuning Knobs Ranked by OtterTune

Our comparison in the previous experiment used DBA-selected knobs. We next measure the quality of the configurations when we remove the human entirely from the tuning process and use OtterTune's Lasso algorithm described in Section 4.1 to select the knobs to tune for all the algorithms. This arrangement is pertinent because, in real-world deployments, a DBA may not be available to choose what knobs to tune or may not be able to rank them correctly. To generate this list of knobs, we train Lasso on the data collected from the experiments in Section 5.2. We then use Lasso to rank the knobs based on their estimated influence on the target objective function [38] and split this list into two sets of 10 and 20 for the algorithms to tune. We again initialize the ML models by executing 10 configurations generated by LHS.

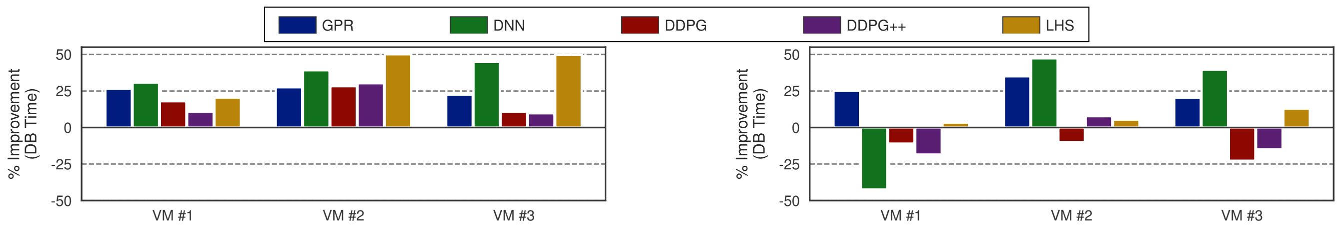
When comparing the knob rankings selected by OtterTune and the DBA, we find that five of the top 10 knobs selected by OtterTune also appear in the DBA's top 10 knobs. For the top 20 OtterTune-selected knobs, 11 of them overlap with the ones chosen by the DBA. Crucially, OtterTune's top 10 knobs include the three most important knobs from Table 2.

Figure 12 shows the performance improvement of the best configuration over the SG default for each algorithm by VM. For 10 knobs, the results show that the DB Time measurements from VM #1 are lower than the other VMs, but that the performance trends of the algorithms are similar. Figure 12b shows that for the 20-knob configurations, the improvements achieved by the algorithms are mostly stable across the three VMs. The exception is DNN, which performs the best on VMs #2 and #3 but then the worst on VM #1.

Figure 13 shows the average performance improvement for 10 and 20 knob configurations. The 10-knob configurations from LHS and DNN perform the best, achieving ~40% better DB Time over the default configuration. GPR, DDPG, and DDPG++ have improvements of 26%, 19%, and 17% for 10 knobs, respectively. Only LHS and DNN generate configurations with the ideal settings for the three most important knobs in Table 2, whereas the other algorithms have incorrect settings for at least one of them. We could not identify any knob in LHS's configuration that explains the 5% improvement over the best configuration in Figure 11.

For 20 knobs, the results in Figure 13 show that the optimized configuration for GPR achieves 27% better DB Time than the SG default configuration. DNN performs the next best, improving the DB Time by 15%. The configurations generated by DDPG++ and LHS improve the performance by less than 10%. For DDPG, none of its optimized configurations that tune 20 knobs improved the DB Time over the default settings. Likewise, none of the worst-performing 20-knob configurations outperformed the default. We believe the overall poor performance of the 20-knob configurations is partly due to more shared storage noise at the beginning of August 2020 when we ran these experiments. The variability in the performance measurements at that time supports this explanation (see Figure 8). All the ML-based algorithms take longer to converge when the performance of the VM is unstable. This especially impacts DDPG and DDPG++ since they take longer to converge in general.

The improvements when tuning the top 10 knobs ranked by OtterTune are comparable to the DBA-ranked knobs shown in Figure 11. This is partly because the Lasso algorithm correctly identified the importance of the three knobs in Table 2. Operating in a cloud environment makes it difficult to determine which set is superior since smaller improvements likely due to noise.



(a) 10 Knobs **(b)** 20 Knobs

Figure 12: Tuning Knobs Ranked by OtterTune (Per VM) – The performance improvement of the best configuration per algorithm running on separate VMs relative to the performance of the SG default configuration measured at the beginning of the tuning session.

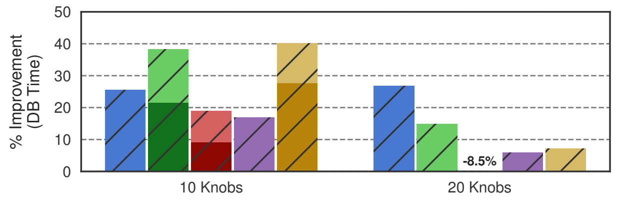


Figure 13: Tuning Knobs Ranked by OtterTune – Performance measurements for the ML algorithm configurations using 10 and 20 knobs selected by OtterTune's Lasso ranking algorithm. The shading on each bar indicates the minimum and maximum performance of the optimized configurations from three tuning sessions.

5.4 Adaptability to Different Workload

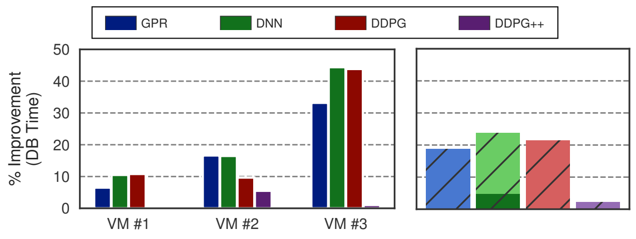
We next analyze the quality of ML-generated configurations when we train their models on one workload and then use them to tune another workload. The ability to reuse training data across workloads potentially reduces the number of iterations that the algorithms need for their models to converge.

We first train models for each algorithm using a TPC-C workload executed by OLTP-Bench [16]. We configured the benchmark to use 200 warehouses (~20 GB) with 50 terminals. We then ran the workload for 10 minutes and captured the queries using Oracle's RAT tool. Next, we tune the top 20 knobs selected by the DBA and train each TPC-C model for 150 iterations. We then use the TPC-C model to tune the TicketTracker workload for 20 iterations.

Figure 14a shows the performance improvement over the SG default configuration achieved by the algorithms per VM. Although the algorithms' rankings based on performance are similar for the three VMs, the performance gains on VM #3 are much larger than the other VMs. The reason is that we calculate the improvement of each algorithm relative to the performance of the SG default configuration, which is particularly bad on VM #3.

Figure 14b shows the DBMS's average performance for the best configurations selected by each algorithm. DNN's configuration performs the best, improving the DB Time by 23% over the default configuration. DDPG and GPR perform nearly as well and achieve 21% and 18% better DB Time, respectively. The best configuration generated by DDPG++ only improves the performance by 3%.

We examined the configurations for differences in the best-observed settings for TPC-C and TicketTracker that may explain why the algorithms were unable to achieve performance comparable to the results in Figures 11 and 13. We observed that none of the algorithms changed the sizes of the two buffer caches in Table 2 from their SG default settings. This is expected for the LOB buffer cache since none of TPC-C's data is stored in the 32 KB tablespace.



(a) Performance per VM **(b)** Performance Summary

Figure 14: Adaptability to Different Workloads – Comparison when applying the model trained on TPC-C data to the TicketTracker workload.

For the main buffer cache, the benefit of increasing its size may be negligible since TPC-C's working set size is small.

We also found contrary settings for the knob that specifies file I/O operations.[5] Its best-observed setting for TicketTracker enables direct I/O, whereas for TPC-C, it disables it. One possibility is that the OS page cache is more efficient than the DBMS's buffer cache for TPC-C because it consists of mostly small writes. This would also explain why the size of the buffer cache was small.

5.5 Execution Time Breakdown

In this section, we evaluate the execution time details to better understand where time is being spent during a tuning iteration. OtterTune's controller and tuning manager record execution times from the service's components for each tuning session. We group these measurements into seven categories:

1. **Restore Database:** Reset the database to its initial state.
2. **Collect Storage Metrics:** Run the `Fio` microbenchmarks on the DBMS's underlying storage device.
3. **Prepare Workload:** Run the Oracle RAT procedures to initialize and prepare for the next workload replay.
4. **Execute Workload:** Replay the workload trace.
5. **Collect Data:** Retrieve knob/metric data and generate the summary reports provided by Oracle.
6. **Download Configuration:** Upload the new result to the OtterTune service and download the next configuration to try.
7. **Update Configuration:** Install the next configuration.

Table 3 shows the breakdown of the median time spent in each category during a tuning iteration. As expected, most of the time is spent executing the workload trace. Although we replay a 10-minute segment of the trace, the actual time it takes to execute it

[5]**Oracle Knob** – `FILESYSTEM_IO`

Category	Time (sec)	% of Total Time
Restore Database	318	18.8%
Run `Fio`	84	5.0%
Prepare Workload	57	3.4%
Execute Workload	959	56.9%
Collect Data	167	9.9%
Download Configuration	19	1.1%
Update Configuration	82	4.9%
Total Time	1777	100%

Table 3: Execution Time Breakdown – The median amount of time spent in different parts of the system during a tuning iteration.

	GPR	DNN	DDPG	DDPG++	LHS
Execute (sec)	762	1006	1021	1274	1311
% Canceled	1.8%	8.7%	12.9%	26.8%	32.4%

Table 4: Workload Replay Time per Algorithm – The median workload execution time and the percentage of replays canceled for the algorithms.

can be longer if the DBMS's configuration has bad settings external factors are affecting the VM's performance (e.g., high I/O latency due to resource contention). The median time it takes to replay the workload is ~16 minutes, but it can take up to 45 minutes. Long-running replays lasting more than 45 minutes are automatically canceled (see Section 6). The next highest percentage of time is spent restoring the database to its original state after the workload replay. This process takes approximately five minutes since the system only needs to restore the modified pages.

OtterTune's controller spends ~10% of its time each iteration collecting data from the DBMS. Although the portion of time spent on this task is relatively low, spending nearly three minutes on data collection might seem questionably high. But only 15 seconds of that time is spent collecting the knob and metric data; the remaining time is spent collecting summary reports provided by Oracle. These reports were useful for debugging the issues we encountered during this study, and thus we believe the overhead is worthwhile.

Of the categories shown in Table 3, the only two that vary per algorithm are *Execute Workload* and *Download Config*. Table 4 shows the median workload execution time and the percentage of replays canceled for each algorithm. Both the execution time and the replay cancel rate are related to how quickly the algorithm converges. As an algorithm learns more, it is less likely to select poor configurations. Thus, the number of long-running replays decreases as the algorithm nears convergence. Table 4 shows that GPR has the fewest canceled replays. DDPG++ has fewer canceled replays than DDPG due to its improved convergence rate (see Section 4.3). LHS has the highest workload execution time and percentage of canceled replays because it is a sampling technique and never converges.

6 LESSONS LEARNED

During the process of setting up and deploying OtterTune at SG for this study, several issues arose that we did not anticipate. Some of these were specific to SG's operating environment and cloud infrastructure. Several issues, however, are related to the broad field of automated DBMS tuning. We now discuss these problems and our solutions for dealing with them.

(1) Handling Long-running Configurations: As discussed in Section 2.2, prior studies on ML-based tuning relied on synthetic benchmarks in their evaluations. Benchmarks like TPC-C are fixed workloads that can be executed for a specific amount of time. Bad knob configurations and other performance factors do not affect the execution time. Conversely, the TicketTracker workload's execution time depends on how long it takes to replay the queries in that trace. Thus, the DBMS's performance affects how long this will take. We found in our experiments that a poor knob configuration could increase the execution of SG's 10-minute trace to several hours. We also observed that the trace took longer when the VMs were experiencing higher I/O latencies.

Given this, the controller needs to support an early abort mechanism that stops long-running workload replays. Setting the early abort threshold to lower values is beneficial because it reduces the total tuning time. This threshold, however, must be set high enough to account for variability in cloud environments. We found that the 45-minute cut-off worked well, but further investigation is needed on more robust methods. For early aborted configurations, the DBMS's metrics, especially Oracle's DB Time, are incorrectly smaller because the workload is cut off. Thus, to correct this data, the controller calculates a *completion ratio* as the number of finished transactions divided by the total transactions in the workload. It then uses this ratio to scale all counter metrics to approximate what they would have been if the DBMS executed the full workload trace.

(2) Handling Failed Configurations: If the ML algorithms do not have prior training data, they will inevitably select poor configurations to install on the DBMS at the beginning of a tuning session. There are two kinds of configurations that cause failures, and each one must be handled by the tuning service differently. The first of these prevents the DBMS from even starting. The most common case is when a knob's value exceeds its allowable bounds. For example, some knobs related to memory cannot be set to a value higher than the available RAM. But this problem also occurs when an implicit dependency that exists between knobs is violated. For Oracle, such a dependency exists between two knobs that configure the DBMS's "shared" pool for SQL statements. One of these knobs controls the total size of the pool and the other specifies how much of the pool to reserve for large objects, which cannot be set to a value larger than half the total size of the pool.[6]

The second kind of bad configuration is when the DBMS successfully starts but then crashes at some point during workload replay. In the case of Oracle, this occurs when the buffer cache size is set too large. The DBMS allocates the memory for this buffer incrementally, and thus it is not caught in start-up checks.

The first issue with a bad configuration is how to identify that it caused a failure. Configurations that cause the DBMS to fail to start or crash require access to its host machine to determine the nature of the failure. To do this, we modified the controller to retrieve Oracle's debug log from the host machine and then check for specific error messages. We acknowledge that DBMS cloud offerings that do not allow login access to the DBMS's host machine will require different failure detection methods.

The next problem is what to do with data collected for failed configurations. Simply discarding this data and starting the next

[6]**Oracle Knobs** – `SHARED_POOL_SIZE, SHARED_POOL_RESERVED_SIZE`

iteration means that the tuning algorithms would fail to learn that the configuration was bad. But including the metrics from a delayed crash is risky because if they are not scaled correctly, the algorithms could improperly learn that those configurations improve the objective function. Our solution is to set the result for that iteration to be twice the objective function value of the worst configuration ever seen. Because the DBMS is not operational with these failed configurations, it is valid to give them the same "score."

(3) DBMS Maintenance Tasks: Every major DBMS contains components that perform periodic maintenance tasks in the system. Some DBMSs invoke these tasks at scheduled intervals, while other tasks are in response to the workload (e.g., Postgres's autovacuum runs when a table is modified a certain number of times). It is best to be aware of these in advance before starting a tuning session.

We also found it helpful to collect metrics from the DBMS's OS to identify the causes of random performance spikes. While running the experiments for this study, we noticed a degradation in Oracle's performance that occurred at the same time each evening. This reduction was due to Oracle's maintenance task that computes optimizer statistics once a day. It took us longer to discover the source of this problem than we would have liked because we did not initially collect the metrics to help us track it down. Since we were restoring the database to the same state after each iteration, our solution was to disable the maintenance task from running in our experiments. Additional research is needed on how to best handle maintenance tasks that are scheduled during a tuning session.

(4) Unexpected Cost Considerations: Our results showed that ML-based algorithms generate configurations that improved performance by up to 45%. Although these gains are noteworthy, there is a trade-off between the time it took to deploy OtterTune versus the benefit. There are several non-obvious factors that one must consider when determining whether an ML-based tuning solution is worthwhile. First, it depends on the economic significance of the applications that the organization wishes to tune. Such considerations include the DBMS software license and hardware costs, and the applications' monetary and SLA requirements.

The second factor to consider is the administrative effort involved in tuning a database. This effort is the cost of going through the proper stakeholders to get approval. Third, it depends on whether the organization has the tooling and infrastructure to run the tuning sessions. These capabilities include the ability to clone the database and its workload onto hardware similar to the production environment. It is non-trivial to estimate these intangible costs relative to the benefit of deploying an ML-based tuning service – they are just factors that an organization must consider to make that decision.

7 RELATED WORK

Much of the previous work on automatic database tuning has focused on optimizing the physical design of the database [12], such as selecting indexes [7, 20], partitioning schemes [8, 13, 29], or materialized views [7]. There have been efforts to automatically tune a DBMSs' configuration knobs since the early 2000s. We classify the previous work on database configuration tuning into two categories: (1) rule-based methods and (2) ML-based methods.

Rule-based methods select DBMSs' knob settings based on a predefined set of rules or heuristics. In most cases, the heuristics only apply to a particular DBMS and target a specific group of knobs [1, 5, 14, 27]; thus, rule-based tools are often limited in scope. The IBM DB2 Performance Wizard Tool asks the DBA questions about their application and provides DBMS knob settings based on the answers [27]. Oracle provides a diagnostic tool that can identify the performance bottlenecks caused by misconfigurations [15]. BestConfig [42] divides the high-dimensional parameter space into subspaces and uses principles derived from the given resource limits of the knobs to search for the optimal configuration.

ML-based methods employ black-box techniques to automatically learn the optimal settings for DBMSs' configuration knobs [41]. Bayesian Optimization (BO) is a popular ML-based approach that has been successfully applied to system configuration and hyperparameter tuning problems [33, 34]. BO is used by both iTuned [17] and our previous work on OtterTune [38], which model the DBMS tuning problem as a Gaussian Process.

Reinforcement learning is another ML-based method that has been adapted for database tuning, notably in CDBTune [40] and QTune [28]. Both of these use the DDPG algorithm described in Section 4.3, but QTune extends it to include information that it extracts from the queries in its models (e.g., query type, accessed tables). Although QTune supports more fine-grained tuning, privacy constraints may prevent some organizations from sharing this query information since the queries contain sensitive user data.

There are also methods that focus on tuning specific knobs to optimize database performance. iBTune only tunes the buffer pool size for individual database instances [36]. It employs a pairwise DNN that uses features from pairs of cloud database instances to predict request response times. RelM is a multi-level tuning method that optimizes memory allocations in data analytics systems (Hadoop, Spark) [26]; it uses Guided Bayesian Optimization that incorporates metrics derived from the application to speed up the optimization. Other studies focused on reducing the number of knobs that a tuning service considers. Our original OtterTune implementation uses Lasso [37] to rank knobs by importance. Kanellis et al. use classification and regression trees (CART) [10] to determine the most important knobs to tune to achieve good performance [24].

8 CONCLUSION

In this study, we conducted a thorough evaluation of machine learning-based DBMS knob tuning methods with a real workload on an Oracle installation in an enterprise environment. We implemented three state-of-the-art ML algorithms in the OtterTune tuning service in order to make a head-to-head comparison. Our results showed that these algorithms could generate knob configurations that improved performance by up to 45% over ones generated by a human expert, but the performance was influenced by the number of tuning knobs and the assistance of human experts in knob selection. We also solved several deployment and measurement issues that were overlooked by previous studies.

ACKNOWLEDGMENTS

This work was supported (in part) by the National Science Foundation (IIS-1846158, III-1423210), the National Science Foundation's Graduate Research Fellowship (DGE-1252522), Google Research Grants, and the Alfred P. Sloan Research Fellowship program. **TKBM**.

REFERENCES

[1] 2011. MySQL Tuning Primer Script. https://launchpad.net/mysql-tuning-primer.

[2] 2021. FIO: Flexible I/O Tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[3] 2021. OtterTune. https://ottertune.cs.cmu.edu.

[4] 2021. OtterTune - Automated Database Tuning Service. https://ottertune.com.

[5] 2021. PostgreSQL Configuration Wizard. https://pgtune.leopard.in.ua.

[6] 2021. Société Générale. https://www.societegenerale.com.

[7] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*. 496–505.

[8] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 359–370.

[9] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research* 13, 10 (2012), 281–305.

[10] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. 1984. *Classification and Regression Trees*. CRC press.

[11] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "What-If" Index Analysis Utility. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. 367–378.

[12] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. 3–14.

[13] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 48–57.

[14] Benoît Dageville and Mohamed Zait. 2002. SQL Memory Management in Oracle9I. In *Proceedings of the 28th International Conference on Very Large Data Bases*. 962–973.

[15] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle. In *CIDR*. 84–94.

[16] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.

[17] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[18] Kurt Engeleiter, John Beresniewicz, and Cecilia Gervasio. 2010. *Maximizing Database Performance: Performance Tuning with DB Time*. Retrieved December 29, 2020 from https://www.oracle.com/technetwork/oem/db-mgmt/s317294-db-perf-tuning-with-db-time-181631.pdf

[19] Leonidas Galanis, Supiti Buranawatanachoke, Romain Colle, Benoît Dageville, Karl Dias, Jonathan Klein, Stratos Papadomanolakis, Leng Leng Tan, Venkateshwaran Venkataramani, Yujun Wang, and Graham Wood. 2008. Oracle Database Replay. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. 1159–1170.

[20] Michael Hammer and Arvola Chan. 1976. Index Selection in a Self-Adaptive Data Base Management System. In *Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data*. 1–8.

[21] Michael Hammer and Bahram Niamir. 1979. A Heuristic Approach to Attribute Partitioning. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. 93–101.

[22] Charles R. Hicks and Kenneth V. Turner. 1997. *Fundamental Concepts in the Design of Experiments* (5 ed.). Oxford University Press.

[23] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. 2001. Characteristics of Production Database Workloads and the TPC Benchmarks. *IBM Systems Journal* 40, 3 (2001), 781–802.

[24] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[25] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *Proceedings of the VLDB Endowment* 5, 1 (2011), 61–72.

[26] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-based Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1667–1683.

[27] Eva Kwan, Sam Lightstone, Adam Storm, and Leanne Wu. 2002. *Automatic Configuration for IBM DB2 Universal Database*. Technical Report. IBM.

[28] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[29] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 61–72.

[30] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. 2018. Parameter Space Noise for Exploration. In *6th International Conference on Learning Representations (ICLR 2018)*.

[31] Carl Edward Rasmussen. 2003. Gaussian Processes in Machine Learning. In *Summer School on Machine Learning*. Springer, 63–71.

[32] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 460–471.

[33] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando De Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (2016), 148–175.

[34] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*. 2951–2959.

[35] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958.

[36] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. iBTune: Individualized Buffer Tuning for Large-Scale Cloud Databases. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1221–1234.

[37] Robert Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[38] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1009–1024.

[39] Bohan Zhang. 2021. https://github.com/bohanjason/ottertune.

[40] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.

[41] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database Meets Artificial Intelligence: A Survey. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[42] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*. 338–350.