A Demonstration of the OtterTune Automatic Database Management System Tuning Service

Bohan Zhang, Dana Van Aken, Justin Wang
Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng
Andrew Pavlo, Geoffrey J. Gordon
Carnegie Mellon University
{ bohanz2, dvanaken, jcwang1 }@cs.cmu.edu
{ tdai1, shulij, jackyl, ssheng }@andrew.cmu.edu
{ pavlo, ggordon }@cs.cmu.edu

ABSTRACT

Database management systems (DBMSs) have a plethora of tunable knobs that control almost everything in the system. The performance of a DBMS is highly dependent on these configuration knobs, however, getting this tuning right is hard. Many organizations resort to hiring experts to configure these knobs, but this is prohibitively expensive. As databases grow in both size and complexity, optimizing a DBMS has surpassed the abilities of even the best human experts. We recently introduced OtterTune, a tuning service that is able to automatically find good settings for a DBMS's configuration knobs. OtterTune leverages data collected from previous tuning efforts to train machine learning models, and recommends new configurations that are as good as or better than ones generated by existing tools or a human expert. In this demonstration, we showcase OtterTune's ability to automatically select a configuration that improves a DBMS's performance.

PVLDB Reference Format:
Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. A Demonstration of the OtterTune Automatic Database Management System Tuning Service. *PVLDB*, 11 (12): 1910 - 1913, 2018.
DOI: https://doi.org/10.14778/3229863.3236222

1. INTRODUCTION

DBMSs are an important component of data-intensive applications. But achieving good performance in DBMSs is non-trivial. Tuning a DBMS to perform well has many challenges. First, modern DBMSs are notorious for having many configuration knobs [7], such as the amount of memory to use for caches and how often data is written to storage. The number of DBMS knobs is always increasing as new versions and features are released. Another challenge is that many of these knobs are not independent, which means that changing one knob may affect the optimal setting for another one. It is hard enough for humans to understand the impact of one knob let alone the interactions between multiple knobs. Lastly, one often cannot reuse the same configuration from one application to the next. This is because the optimal configuration depends heavily on the application's workload and the database server's underlying hardware. The best configuration for one application may not be the best for another.

What is needed is a tuning tool that automatically selects the best DBMS configuration for a particular application. Previous efforts to create tools for this problem have made progress [3, 5], but they have shortcomings that limit their usefulness in practice. Foremost is that many of them only support a single DBMS. This is often because they rely on some internal mechanism of the target DBMS to reason about the workload (e.g., the optimizer's cost model). A small number of tuning tools do support multiple DBMSs tuning, but they still require manual steps, such as having the DBA (1) deploy a second copy of the database, (2) map dependencies between knobs, or (3) guide the training process. The second is that none of these tools are able to reuse previous training data. Instead, they must tune each DBMS deployment from scratch without transferring knowledge gained from previous tuning efforts, which takes time and requires additional resources to collect enough information to make the correct decision.

Given these deficiencies in existing tools, we developed the **OtterTune** [1] automatic DBMS tuning service through machine learning. OtterTune differs from other tuning tools because it leverages knowledge gained from tuning previous DBMS deployments to tune new ones. This reduces the amount of time and resources it takes to tune a DBMS for a new application. To do this, OtterTune maintains a repository of tuning data collected from previous tuning sessions. It uses this data to build a combination of supervised and unsupervised machine learning (ML) models, and uses these models to (1) select runtime metrics to characterize workloads, (2) choose the most impactful knobs to tune, (3) map unseen database workloads to previous workloads from which it can transfer experience, and (4) recommend knob settings that improve the target objective (e.g., throughput, latency). Our previous results [7] show that OtterTune produces a DBMS configuration for both OLTP and OLAP workloads that achieves 58–94% lower latency compared to their default settings or configurations generated by other tuning advisors. OtterTune also generates configurations in under 60 min that are within 94% of ones created by expert DBAs.

In this paper, we provide the details of OtterTune's implementation and lessons that we have learned in the process. We begin with an overview of OtterTune's architecture in Section 2. We then describe its machine learning pipeline in Section 3. We conclude in Section 4 with our description of the demonstration.

Proceedings of the VLDB Endowment, Vol. 11, No. 12

DOI: https://doi.org/10.14778/3229863.3236222

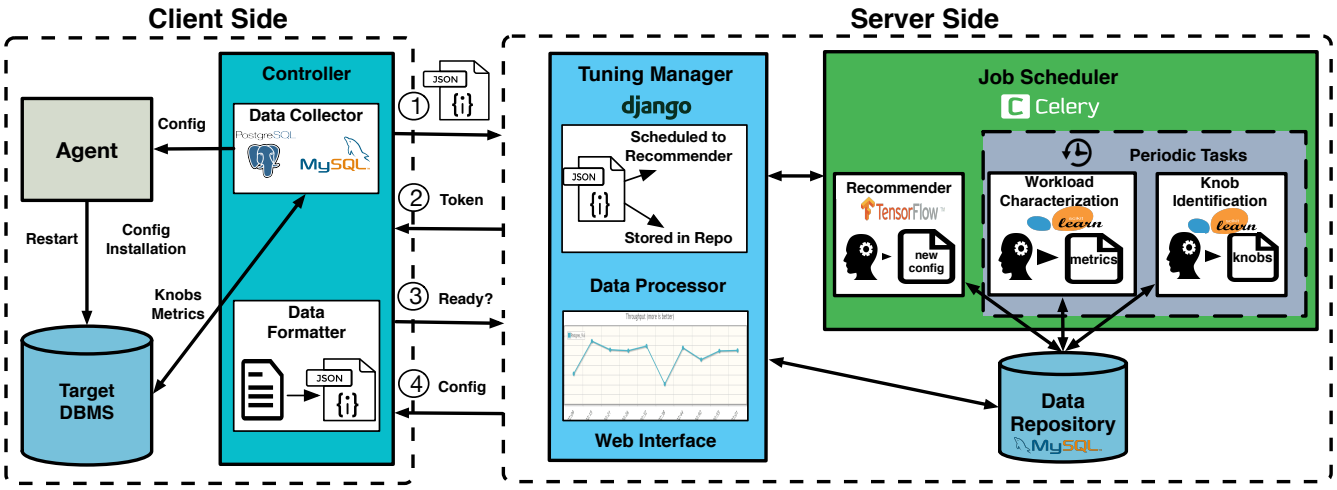

Figure 1: OtterTune Architecture – The controller connects to the target DBMS, collects its knob/metric data, transforms the collected information into the universal JSON schema, and sends it to the server-side tuning manager. The tuning manager stores the information in the data repository and schedules a new task with the job scheduler to compute the next configuration for the target DBMS to try. The controller (1) sends the information to the tuning manager, (2) gets a token from the tuning manager, (3) uses this token to check status of the tuning job, and (4) gets the recommended configuration when the job finishes.

2. ARCHITECTURE

As shown in Figure 1, OtterTune is comprised of two components: (1) **client-side controller** and (2) **server-side tuning manager**. A user deploys the controller in the same administrative domain as the target DBMS. It begins a new tuning session by collecting information from target DBMS (i.e., the system that the user wishes to tune). It then collects this information again after an observation period (five minutes by default) and transmits them to the tuning manager. The tuning manager processes this data and recommends a new knob configuration to the controller. The controller can then install this new configuration, collect more performance data to measure its impact, and repeat this process until the user is satisfied with the target DBMS's performance.

2.1 Client-side Controller

OtterTune's Java-based controller acts as the intermediary between the target DBMS and the tuning manager. Since OtterTune only requires access to the DBMS's runtime metrics through JDBC, the controller is deployed externally. That is, it does not need to "observe" the application's query trace or require the user to install special binaries inside of the DBMS. It also does not require interruption of the application to collect this information.

At the beginning of a new tuning session, the controller connects to the target DBMS and collects its current knob configuration. It also takes a snapshot of the DBMS's internal metrics at the beginning of the session. Although we designed OtterTune to be DBMS-agnostic as much as possible, the commands to retrieve this information are different for each DBMS. The controller contains modules for collecting the knobs and metrics from each DBMS that OtterTune supports[1]. These DBMSs expose this information via their catalogs and other internal tables that are accessible through SQL. Therefore, adding support for a new DBMS requires only basic SQL knowledge. We also found that different versions of the same DBMS differ in what data is available. For example, Postgres v9.4 includes a metric table about its WAL archiver process's activity[2], but this table is not available in earlier versions. Thus, each DBMS collector module also supports different versions of the same DBMS. It remains future work to determine how to automatically combine the metric data from different versions of the same DBMS.

Another challenge in collecting this data from the DBMS is that some systems maintain knobs and metrics that are specific to a subcomponent in the system. Some systems, like MySQL, only report aggregate statistics for the entire DBMS and each knob affects the entire system. Other systems, however, provide separate knobs and metrics for databases, tables, and even sub-elements of a table. For example, MyRocks supports separate knobs and metrics for the column families in a single table[3]. To handle these diverse data sets, the controller transforms them into a universal schema that separates elements into the "global" and "local" groups.

During the observation period, the controller does not interact with the DBMS. This is to ensure that OtterTune has minimal interference with the execution of the application's workload. At the end of the observation period, the controller collects the same internal metrics from the DBMS as it did in the beginning of the period and uploads them to the tuning manager for processing. It receives a unique token from the tuning manager, and uses it to check the task status periodically. Once the task finishes and the tuning manager recommends a configuration successfully, the controller downloads it and employs a separate *agent* to install the new configuration and restart the target DBMS. The agent is a stand-alone script used by the controller for performing tasks on the target DBMS that cannot be done through JDBC, such as those that require administrative privileges (e.g., restarting the system). The controller then starts the next observation period and uploads the collected information in the next loop. This process continues until the user is satisfied with the improvements over the initial configuration.

2.2 Server-side Tuning Manager

The tuning manager is responsible for processing and storing tuning data, scheduling jobs to compute OtterTune's ML models and make configuration recommendations, and visualizing the results from each tuning session in its front-end web interface. The tuning manager is written in Python using the Django web framework. We use the same MySQL v5.7 database for OtterTune's data repository and the Django back-end database. We use Django's ORM APIs to design and create all of the tables, and to query them for data

[1] As of March 2018, OtterTune supports Postgres (v9), MySQL (v5), MyRocks, VectorWise, and Greenplum.

[2] Postgres v9.4 – `pg_stat_archiver`

[3] MyRocks v5.6 – `write_buffer_size, cur_size_active_mem_table`

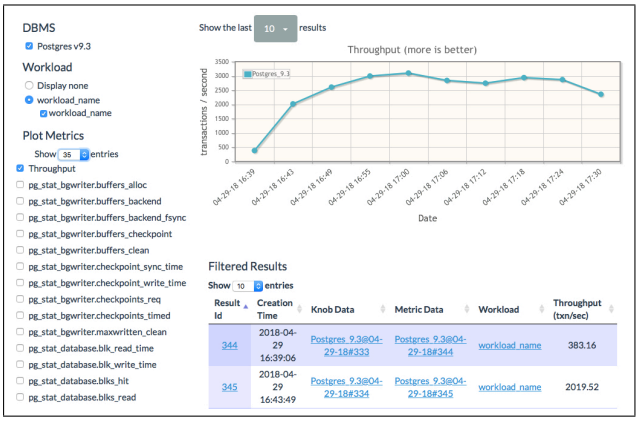

Figure 2: Tuning Session – The user can view the performance of the target DBMS as it tries the configurations recommended throughout the tuning session. Users can also view detailed information about the knob settings and metric values collected during each observation period.

to feed into the ML model training components. We use Celery to schedule and execute tasks for creating OtterTune's ML models and recommending new configurations. Celery is a task queue and scheduler that is easy to integrate with web frameworks like Django. We implemented all of OtterTune's ML models using Python's scikit-learn and Google TensorFlow.

When the tuning manager receives the target DBMS's knob and metric data from the controller via its REST API, it first stores this information in its data repository and then visualizes the results in the front-end. As shown in Figure 2, the user can view graphs of the metrics to see the change in the target DBMS's performance for the different configurations that have been recommended so far in the tuning session. Next, the tuning manager schedules a Celery asynchronous task to incorporate the new knob and metric data into its ML models and then compute the next configuration for the target DBMS to try. As we described in Section 2.1, the tuning manager returns to the controller a unique token for checking the status of the task since it may take several minutes to complete if the task queue is long. When the task finishes, a link is provided to the controller to download the next configuration.

In addition to scheduling asynchronous tasks for making configuration recommendations, the tuning manager also schedules Celery background tasks to recompute the models in OtterTune's ML pipeline to incorporate any new data available in the data repository. These only need to be executed periodically (i.e., every 20 minutes) or whenever a new application begins a new tuning session. We discuss OtterTune's ML pipeline in more detail next in Section 3.

OtterTune cannot train its ML models when no data is available in its data repository. The tuning manager provides two methods for bootstrapping the tuning manager with initial training data. If the user has previous tuning data available externally, then they can use the "batch upload" option in the web interface to upload all of it at once to OtterTune's data repository. Otherwise, the tuning manager generates initial training data by selecting random knob configurations for the target DBMS to try. It starts the tuning session once it collects enough data to train its ML models.

3. TUNING PIPELINE

The knob and metric data collected from past tuning sessions resides in OtterTune's repository. This data is processed in OtterTune's ML pipeline, which consists of three components: (1) Workload Characterization, (2) Knob Identification, and (3) Automatic Tuning. OtterTune first passes the data to the Workload Characterization component. This component identifies a smaller set of DBMS metrics that best capture the variability in performance and the distinguishing characteristics for different workloads. Next, the Knob Identification component generates a ranked list of the knobs that most affect the DBMS's performance. OtterTune then feeds all of this information to the Automatic Tuner. This last component maps the target DBMS's workload to the most similar workload in its data repository, and reuses this workload data to generate better configurations. We now describe these components in more detail.

3.1 Workload Characterization

OtterTune uses the DBMS's internal runtime metrics to characterize how a workload behaves. These metrics provide an accurate representation of a workload because they capture many aspects of its runtime behavior. However, many of the metrics are redundant: some are the same measurement recorded in different units, and others represent independent components of the DBMS whose values are highly correlated. It is important to prune redundant metrics because it reduces the complexity of the ML models that use them. To do this, OtterTune first uses factor analysis (FA) to model each internal runtime metric as linear combinations of a few factors. It then clusters the metrics via k-means, using their factor coefficients as coordinates. Similar metrics are in the same cluster, and it selects one representative metric from each cluster, namely, the one closest to the cluster's center. This set of non-redundant metrics is used in subsequent components in OtterTune's ML pipeline.

3.2 Knob Identification

DBMSs can have hundreds of knobs, but only a subset of them affect the DBMS's performance. OtterTune uses a popular feature-selection technique, called Lasso, to determine which knobs have the most impact the system's overall performance. OtterTune applies this technique to the data in its repository in order to identify the order of importance of the DBMS's knobs.

OtterTune must also decide how many of the knobs to use when making configuration recommendations. Using too many of them significantly increases OtterTune's optimization time, however, using too few could prevent OtterTune from finding the best configuration. To automate this process, OtterTune uses an incremental approach in which it gradually increases the number of knobs used in a tuning session. This approach allows OtterTune to explore and optimize the configuration for a small set of the most important knobs before expanding its scope to consider others.

3.3 Automated Tuning

The Automatic Tuner determines which configuration OtterTune should recommend by performing a two-step analysis after each observation period. First, the system uses the performance data for the metrics identified in the Workload Characterization component to identify the workload from a previous tuning session that best represents the target DBMS's workload. It compares the metrics collected so far in the tuning session with those from previous workloads by calculating the Euclidean distance, and finds the previous workload that is most similar to the target workload, namely, the one with smallest Euclidean distance.

Then, OtterTune chooses another knob configuration to try. It fits a Gaussian Process Regression model to the data that it has collected, along with the data from the most similar workload in its repository. This model lets OtterTune predict how well the DBMS will perform with each possible configuration. OtterTune optimizes the next configuration, trading off exploration (gathering information to improve the model) against exploitation (greedily trying to do well on the target metric).

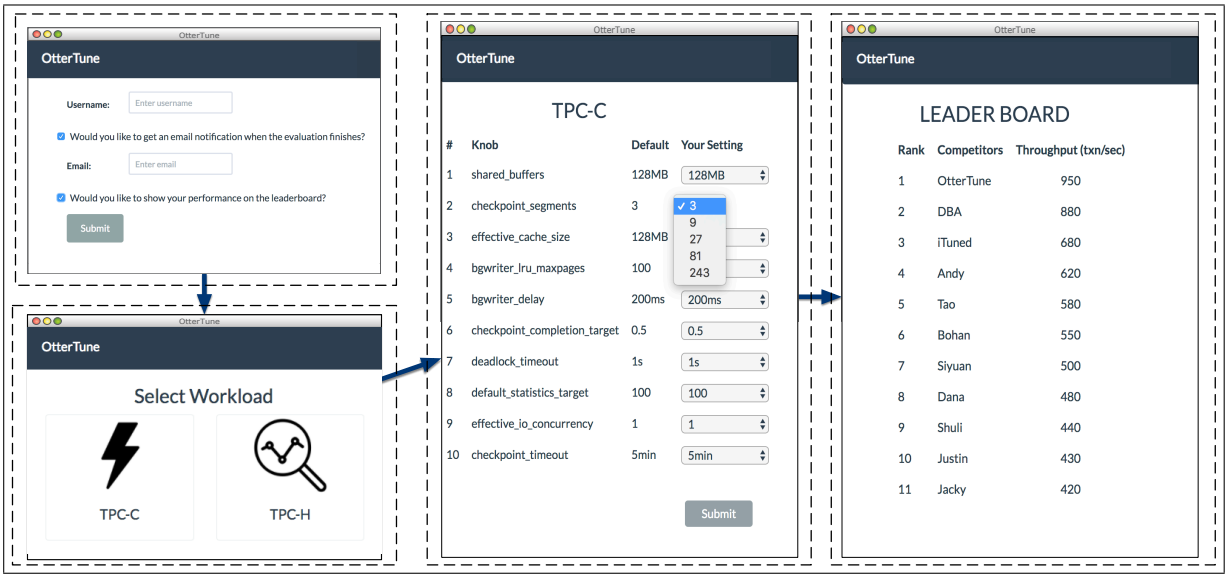

Figure 3: OtterTune Demonstration – The user chooses a workload for the target DBMS and suggests a knob configuration for it to try. We provide users with ten important knobs and five tuning options for each knob. Users can be notified via email with the performance and ranking of their configuration once the observation period ends.The leaderboard shows the rankings of the knob configurations selected by OtterTune, an expert DBA, tuning tools, and other audiences.

4. DEMONSTRATION

Our demonstration of OtterTune is meant to showcase its ability to select DBMS configurations that exceed ones generated by humans. We will challenge participants to tune Postgres (v9.6) to achieve the best performance for OLTP and OLAP workloads (TPC-C and TPC-H, respectively). For each configuration, we will deploy it on the DBMS and run the target workload using the OLTP-Bench benchmark suite [4]. We will also run OtterTune on separate machines and let it learn from these new trials. The system will rank the users' configuration against the OtterTune's best configuration.

As shown in Figure 3, the demonstration is comprised of four steps. Using a web interface, the user first selects which workload they want to tune. The interface then provides the user with ten most important configuration knobs in Postgres v9.6 (as selected by OtterTune's algorithms described in Section 3.2) that they can tune. For each knob, the table includes (1) a short description of what that knob does, (2) the default value, and (3) a list of five different pre-selected possible values. The user can choose a different setting for each knob or leave it set at the default value. After completing their selection, the user then submits the configuration and it is scheduled for execution in the testing cluster. They will then be shown how their selection differs from the OtterTune's best configuration.

We will run the benchmark using OLTP-Bench for five minutes and report the average throughput. OLTP-Bench will then upload this result to a leaderboard to show which user has the best configuration and how it compares with the best configuration from OtterTune. The leaderboard will also include additional results from a human DBA expert and a heuristic-based tuning tool (PGTune [2]).

To ensure that each configuration is fully evaluated, we will run the workload on OLTP-Bench for several minutes. Thus, the result of a user's submission is not known immediately. We will notify them when their result is ready via email. This will take them to a page that OtterTune generates with a breakdown of the performance metrics that OtterTune's controller collected from the DBMS.

Demo Takeways: The goals of this demo are threefold. First, we seek to engage with the audience using a demonstration of the challenges of DBMS tuning. This motivates the need for an automated tuning tool like OtterTune. Second, we will showcase OtterTune's ability to incorporate new information from repeated observation periods in its ML models and improve the efficacy of its recommendations. Lastly, we hope that the game provides users with insight into tuning DBMSs for different workloads.

5. CONCLUSION

We presented the design and implementation of OtterTune service for automatically tuning DBMS configurations. Unlike previous tuning tools, OtterTune reuses training data gathered from previous tuning sessions. Our approach uses a combination of supervised and unsupervised machine learning methods to (1) select the most impactful knobs, (2) map previously unseen database workloads to known workloads, and (3) recommend knob settings. In this paper, we described the architecture of OtterTune and demonstration plans.

Acknowledgments

This research was funded (in part) by the U.S. National Science Foundation (III-1423210), the National Science Foundation's Graduate Research Fellowship Program (DGE-1252522), and AWS Cloud Credits for Research.

This paper is dedicated to the memory of Leon Wrinkles.
May his tortured soul rest in peace.

6. REFERENCES

[1] OtterTune. `https://ottertune.cs.cmu.edu`.
[2] PostgreSQL Configuration Wizard. `http://pgfoundry.org/projects/pgtune/`.
[3] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. *VLDB*, pages 3–14, 2007.
[4] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.
[5] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTuned. *PVLDB*, 2(1):1246–1257, 2009.
[6] A. Sharma, F. M. Schuhknecht, and J. Dittrich. The case for automatic database administration using deep reinforcement learning. arXiv:1801.05643.
[7] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. SIGMOD, pages 1009–1024, 2017.