15-400 Final Report : Reusing Performance Data Across System Versions for Automatic Database Configuration Tuning

Shuli Jiang
& CMU Database group:
Andy Pavlo, Dana Van Aken, Bohan Zhang, Justin Wang, Geoffrey J. Gorden
Tao Dai, Jacky Lao, Siyuan Sheng
Carnegie Mellon University School of Computer Science
shulij@andrew.cmu.edu
{pavlo, dvanaken, bohanz2, jcwang1, ggordon}@cs.cmu.edu
{tdai1, jackyl, ssheng}@andrew.cmu.edu

Abstract

Database management systems (DBMSs) have hundreds of tunable knobs that control almost everything in the system. The performance of a DBMS is highly dependent on these configuration knobs. However, tuning the knobs are usually done manually by database experts, which is an expensive and time-consuming process. Therefore, we wish to automate the tuning process and reuse knowledge from past tuning experiences to tuning new configuration knobs to reduce the cost. We introduce OtterTune, a tuning service we collectively built that is able to automatically find good settings for a DBMSs configuration knobs based on both new data and past performance data. We would like to further explore the impact of reusing performance data across different versions of DBMSs.

Welcome to check out the OtterTune project website at http://ottertune.cs.cmu.edu.

1 Introduction

1.1 Problem

In order to achieve good performances of DMBSs, tuning configuration knobs is essential yet non-trivial. The are three major challenges we want to solve for tuning DMBSs are: 1) There are many knobs to tune. 2) Those knobs are interrelated but it is hard to model their relationship. 3) The same configuration cannot be easily reused for a new application. (Van Aken et al. 2017, Zhang et al. 2018)

1.2 Approach

The problem of tuning DBMS configuration knobs could be modeled as a black-box function optimization problem in a classical machine learning setting. The problem setup is as follow: given an objective function, the algorithm can query f(x) for some input x but it does not know the gradient of the function and cannot make any assumptions. The goal is to optimize the function with as few queries as possible. The problem setting is often applied to algorithm hyperparameter tuning. In the case of DBMSs, we can view the performance of DBMS, such as throughput, latency, etc. as the black-box objective function and configuration knobs as the hyperparameters. We want to automatically find a knobs configuration that can either maximize the throughput or minimize the latency of querying a database.

Bayesian optimization provides an elegant solution to the are black-box function optimization problem. Bayesian optimization works by assuming the black-box function is sampled from Gaussian process (GP) and maintains a posterior distribution for this function as observations are made (Snoek et al. 2012). Another key part of the approach is the acquisition function which determines the next point for evaluation (Yogatama et al. 2014). We use GP to model the observation of DBMS performance and use upper confidence bound (UCB), one of the most commonly used acquisition function, as the evaluation criteria for determining the next knob configuration.

In order to make use of previous tuning knowledge in tuning new DBMS knobs, other machine learning is also applied in building the system, which will be described in detail in the following sections. For example, we use k means to prune redundant past information and use GP again for selecting the most relevant past knowledge.

1.3 Related Work

There isn't much literature on database tuning. (Duan et al. 2009) first proposes a DBMS tuning system, iTuned, that automates the task of identifying good settings for database configuration parameters. The iTune system has three features: adaptive sampling of new knobs configuration, an executor that supports online experiments in database environments through a cycle-stealing paradigm and portability across different database systems. However, the largest drawback of (Duan et al. 2009) is that it does not leverage past experiences.

Based on previous work, (Van Aken et al. 2017) provides a prototype of OtterTune. The paper demonstrates how utilizing both past experience and new information can efficiently automate the process of DBMSs tuning and achieve performances that exceed human experts. The paper shows how to utilize both new and previous knowledge by selecting the most impactful knobs (new

information), mapping unseen database workloads to previous workloads (reuse past experience), and recommending knob settings based on both of them.

1.4 Technical Contributions

The key technical contribution of the project is to develop a novel system (OtterTune) that utilize both previous tuning knowledge and new data collected from users and applies machine learning methods to automate the process of tuning DBMS configuration knobs.

2 Project Overall Status

OtterTune is a long term project. The first part of the plan is to implement the idea from (Van Aken et al. 2017) into a real world system, OtterTune. Building the OtterTune system makes it easy for further experiments and exploration, which is necessary for the second part of the plan, exploring how to efficiently reuse performance data across versions of DBMS.

During 15-400, I worked with CMU Database group to collectively build the OtterTune system. By the end of the semester, the first part of the plan is basically done. But the project is yet to be finished. I'm currently heading towards the second part and will continue working on OtterTune after 15-400.

3 The OtterTune System

This section will briefly describe each component within the OtterTune system. We have submitted a demo paper, *A Demonstration of the OtterTune Automatic Database Management System Tuning Service* (Zhang et al. 2018) as a group in which there is a more detailed description. If you are interested, please checkout the demo paper.

OtterTune mainly consists of two parts as shown in figure 1, the client's side and the server's side. A tuning session begins with the client collecting the first batch of database performance data with random knobs configuration as the initialization point for training the GP model. The client then uploads the data to the server's side and the server will use GP-UCB to recommend the next knob configuration. The server sends the new configuration back to the controller at the client's side to install the new configuration. The user then collect the performance data again and repeat the process. The server will find the best knob configuration through the back-and-forth interaction with the client.

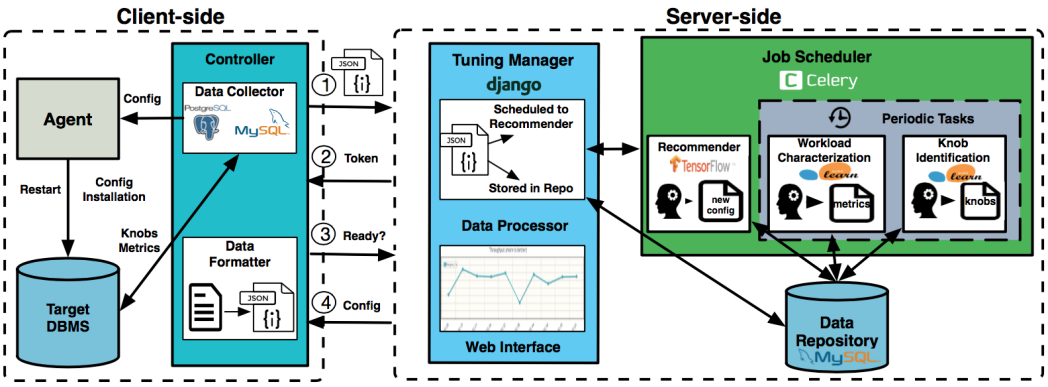

Figure 1: OtterTune Architecture - The controller connects to the target DBMS, collects its knob/metric data, transforms the collected information into the universal JSON schema, and sends it to the server-side tuning manager. The tuning manager stores the information in the data repository and schedules 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. (Picture and Caption source: Zhang et al. 2018)

3.1 The Client's Side

The controller from the client's side benchmarks the database performance with some benchmark framework (usually Oltpbench) though a period of observation (by default 5 minutes), collects the performance metrics and uploads the results to the server. The controller acts as an independent agent outside the database. It does not interact with the server during its observation period to make sure the server has minimal impact when it collects performance metrics given a certain knob configuration. After it uploads the results, it will receive a token from the tuning manager and uses the token to periodically check whether the server has generated the recommended knob configuration. If the next configuration is ready, the controller will automatically install the new configuration and performs experiments on its database to collect metrics again.

3.2 The Server's side

The tuning manager processes and stores the tuning data as well as interacts with the Job Scheduler to train the machine learning models and to generates the next knob configuration. The tuning manager itself can be served as a front-end web application of OtterTune, which visualizes all tuning data and results of the tuning sessions.

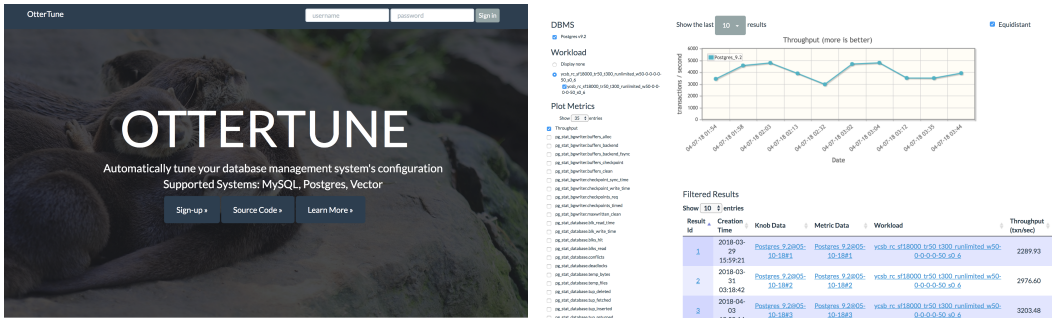

Figure 2: The OtterTune front-end web app. Please checkout http://ottertune.cs.cmu.edu for more information. The user can view the performance of the target DBMS as it tries the configurations recommended throughout the tuning session. Users can also view detailed information about the knob settings and metric values collected during each observation period (Zhang et al. 2018).

After the tuning manager receives tuning knobs and metrics data from the client, it schedules two kind of tasks: 1) an asynchronous task that feeds the processed new training data into the machine learning pipeline to get the next knob configuration. 2) a background task that processes all stored past training data for later usage into training new data. The two tasks are part of the machine learning pipeline described below.

The machine learning pipeline within the Job Scheduler consists of three parts: 1) Work-load Characterization, 2) Knob Identification, and 3) Automatic Tuning. Work-load Characterization runs as the background task. For each unique workload, a Work-load Characterization task prunes redundant metrics and stores only the essential ones that capture how a workload behaves. When a Work-load Characterization task sees new metrics data from the training data, it first uses k means to group all metrics for each workload and picks the one that is closet to the center of each group. It discards the reset of the metrics and stores only the selected ones for each unique workload. When new training data comes into the machine learning pipeline, the new data will first be mapped to the closest workload the pipeline has seen from the past. The GP model is trained by a concatenation of new data and stored metrics of the selected workload. This is how the pipeline reuses past knowledge to train new data.

As a DBMS could have hundreds of knobs but only a few would actually impact the performance, tuning all knobs from the training data could be a waste of time. Thus the machine learning pipeline needs to know which knobs it needs to tune. The Knob Identification task uses Lasso to rank how impactful each knob is in terms of database performance. The machine learning will start tuning the most impactful 10 knobs by default and incrementally adds the number of knobs to tune because tuning too many knobs from the very beginning will greatly increase the optimization time.

Finally, the Automatic Tuning task connects each component in the machine learning pipeline. It maps new data to the closest workload from previous tuning sessions, trains GP-UCB with a concatenation of new and old data, generates next knobs configuration and send it back to the controller.

4 Evaluation

4.1 Data Collection

As mentioned before, in order evaluate the performance of OtterTune, we first need to train the GP-UCB optimization model with pre-collect data from some DBMS using a set of fixed workload and random knobs configuration as the initialization point.

We choose Postgres, one of the databases that OtterTune currently supports, as our target for experiments. We collect data using Postgres on AWS EC2 m3.xlarge machine (4vCPUs and 15 GB virtual memories). The performance metrics of each knob configuration is measured by Yahoo! Cloud Serving Benchmark (YCSB) benchmark and collected through Oltpbench, a database benchmarking framework. We have collected a total of three sets of data from Postgres 9.2, 9.3 and 9.4. Postgres 9.2 and 9.4 data was collected using 6 different workloads while Postgres 9.3 data was collected using 16 different workloads. Each workload has 1500 - 2000 samples.

4.2 Experimental Evaluation

We have evaluated OtterTune's performance with three experiments: 1) using Postgres 9.3 data on a local 16CPU machine 2) using Postgres 9.2 data on AWS EC2 m3.xlarge, 3) using Postgres 9.3 data on AWS EC2 m3.xlarge. We initialize the tuning pipeline with pre-collected data as described in previous section. All database performances in OtterTune tuning evaluation are measured by TPCC, an online transaction processing benchmark. We can visualize all tuning results via the OtterTune website.

Experiment 1: Postgres 9.3, local machine

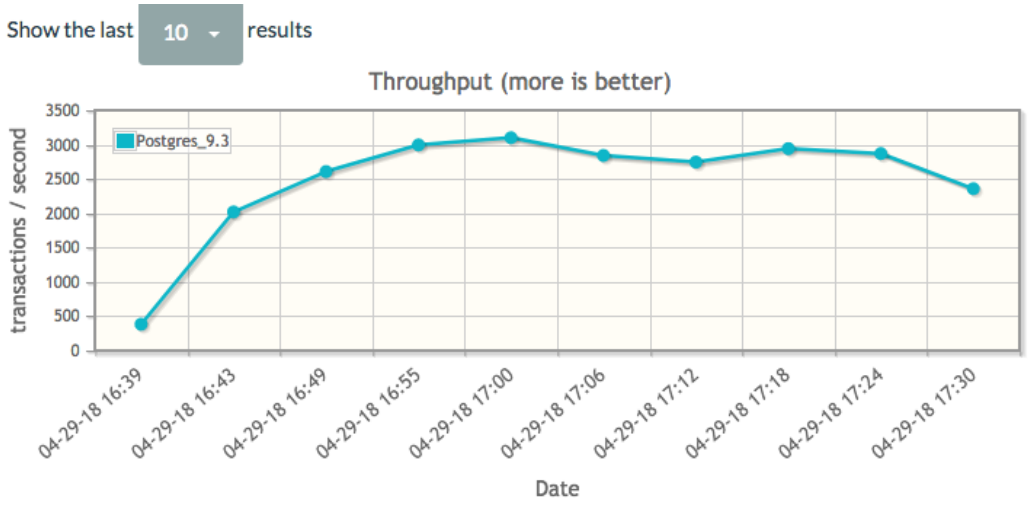

Figure 3: OtterTune performance on Postgres 9.3 data with 16 different workloads on a local machine across 10 configuration recommendations.

We used all 16 workloads for tuning Postgres 9.2 on a local machine. We could see that OtterTune tunes well on Postgres 9.3 on the local machine. The throughput increases significantly, from below 500 to over 3000 transactions per second, within only 5 configuration recommendations. Seeing a success of tuning Postgres 9.3, we moved the experiment setting to tuning Postgres on AWS EC2 m3.xlarge, the machine we used for collecting training data. Due to time limit, we have only tried Postgres 9.2 and 9.3 data to evaluate OtterTune's performance.

Experiment 2: Postgres 9.2, AWS m3.xlarge

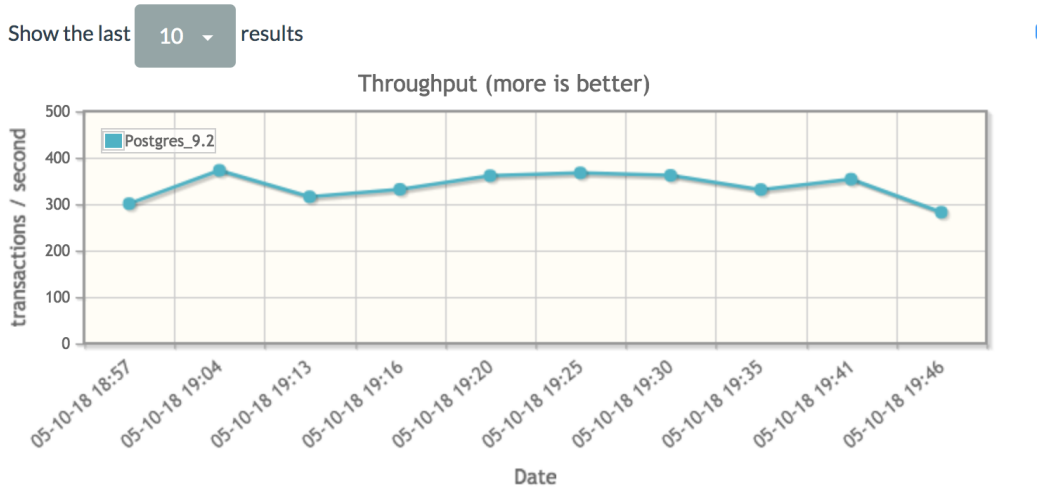

Figure 4: OtterTune performance on Postgres 9.2 data with 6 different workloads on AWS m3.xlarge across 10 configuration recommendations.

This time, OtterTune does not show a significant improvement in terms of throughputs. The throughput fluctuates between 300 and 400 transactions per second, which is almost the same as a default configuration. Not sure what the problem was, we did a second experiment using the same dataset, on the same machine but ran for a longer period. We evaluated OtterTune by having it recommend 20 configurations.

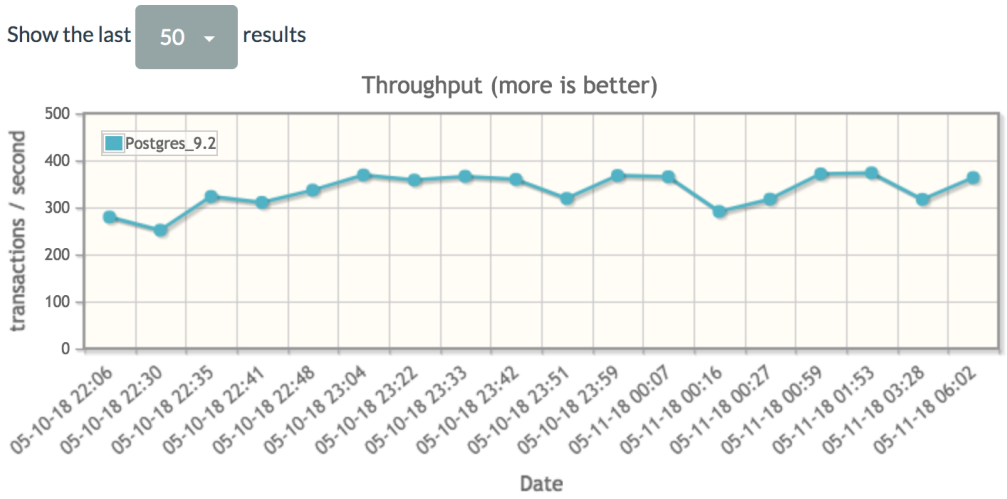

Figure 5: OtterTune performance on Postgres 9.2 data with 6 different workloads on AWS m3.xlarge across 20 configuration recommendations.

OtterTune still does not show a significant improvement in terms of throughputs. The throughput starts below 300 and finally fluctuates between 300 and 400 transactions per second. Another problem found in this experiment is that usually it takes less than 10 minutes for OtterTune to generate the next knob configuration. But from the figure above, it is not hard to see it takes OtterTune more than 1 hour each time to generate the last three configurations. This significantly impairs OtterTune's performance.

Experiment 3: Postgres 9.3, AWS m3.xlarge

When we used full 16 workloads on AWS m3.xlarge, OtterTune was surprisingly irresponsive to running the task of workload mapping and generating knob configuration. This is because running background takes a significantly longer time on AWS m3.xlarge since running a background task could now take around 1200 seconds while running a background task on 6 workloads when we tuned Postgres 9.2 only took around 200 seconds. We decided to test OtterTune on 6 random workloads first.

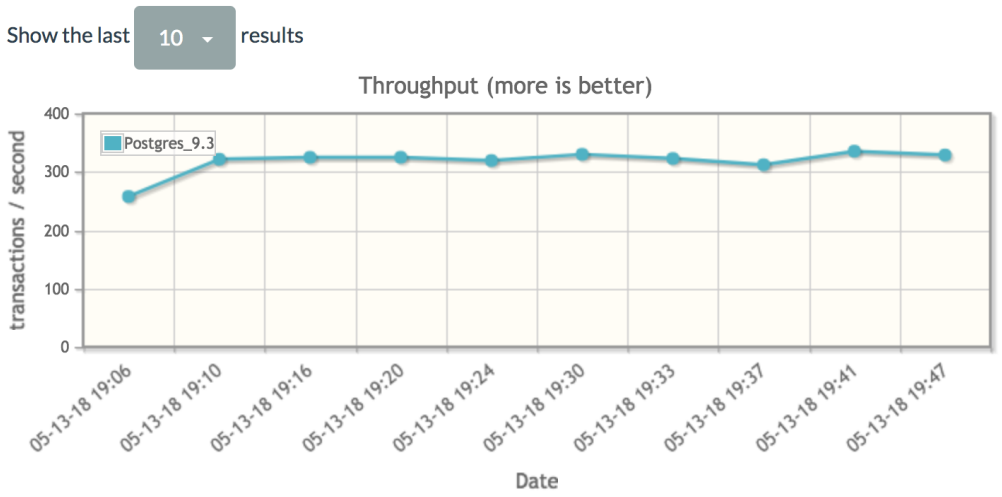

Figure 6: OtterTune performance on Postgres 9.3 data with 6 different workloads at AWS m3.xlarge across 10 configuration recommendations.

The throughput seems to increase at first but finally stabilizes above 300 transactions per second.

4.3 Analysis of Results

OtterTune shows a significant improvement in performance at a local 16CPU machine but does not perform well on both Postgres 9.2 and 9.3 data at AWS m3.xlarge. This might due to machine differences or due to insufficient data for initialization since we only used 6 workloads to tune both Postgres 9.2 and 9.3. We need further investigation into what kind of initialization data is necessary for OtterTune to achieve good performance, e.g. the number of workloads. We also need to figure out a way to deal with significant delay in executing knob recommendation tasks with 16 workloads on AWS m3.xlarge machine.

Another issue observed during experiments with current OtterTune is that the system's performance relies heavily on initialization data. A bad initialization data will not only prevent OtterTune from achieving good performances but also causes other surprising problems like 'segmentation fault'.

5 Conclusions and Future Work

We introduced OtterTune, a machine learning based automatic DBMS knobs tuning system that recommends knob configurations based on both new and past performance data. OtterTune shows a good performance at a local machine in tuning Postgres 9.3 but also exposes several issues when tuning data at AWS machine. The next step is to first investigate more why OtterTune does not perform well on AWS m3.xlarge and fix problems that might be caused due to bad initialization data. After that, we would like to explore the impact of using data from one version of DBMS to

tune another version. Current way of reusing past knowledge by concatenating selected data seen from past workloads to new data is space inefficient since it requires storing the original data. We would like to capture more features of past data and use those features instead of raw data directly when transferring knowledge from tuning sessions across versions. One way of doing this is through transfer GP as described in (Yogatama et al. 2014) where we could reconstruct GP models for recommending new knob configurations based on past GPs.

6 References

[1] Duan, S. Thummala, V. Babu, S. (2009) Tuning Database Configuration Parameters with iTuned, *Proceedings to VLDB '09*.

[2] Van Aken, D. Pavlo, A. Gordon, G.J., Zhang, B. (2017). Automatic Database Management System Tuning Through Large-scale Machine Learning, *Proceedings to SIGMOD'17*.

[3] Zhang, B. Van Aken, D. Wang, J. Dai, T. Jiang, S. Lao, J. Sheng, S. Pavlo, A. Gordon, G.J. (2018) A Demonstration of the OtterTune Automatic Database Management System Tuning Service, *Proceedings to VLDB '18*.

[4] Snoek, J. Larochelle, H. Adams, R.P. (2012) Practical Bayesian Optimization of Machine Learning Algorithms, *Proceedings to NIPS '12*.

[5] Yogatama, D. Mann, G. (2014) E cient Transfer Learning Method for Automatic Hyperparameter Tuning, *Proceedings to AISTATS '14*.