On Automatic Database Management System Tuning Using Machine Learning

Dana Van Aken
June 14, 2020

Thesis Proposal

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Andrew Pavlo, Chair
David G. Andersen
Michael Cafarella, University of Michigan
Geoffrey J. Gordon

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Copyright © 2020 Dana Van Aken
Abstract

Database management systems (DBMSs) are an essential component of any data-intensive application. But tuning a DBMS to perform well is a notoriously difficult task because they have hundreds of configuration knobs that control aspects of their runtime behavior, such as cache sizes and how frequently to flush data to disk. Getting the right configuration for these knobs is hard because they are not standardized (i.e., sets of knobs for different DBMSs vary), not independent (i.e., changing one knob may alter the effects of others), and not uniform (i.e., the optimal configuration depends on the target workload and hardware). Furthermore, as databases grow in both size and complexity, optimizing a DBMS to meet the needs of new applications has surpassed the abilities of even the best human experts. Recent studies using machine learning techniques to automatically configure a DBMS’s knobs have shown that such techniques can produce high-quality configurations; however, they need a large amount of training data to achieve good results. Collecting this data is costly and time-consuming.

In this thesis, we seek to address the challenge of developing effective yet practical techniques for the automatic configuration of DBMSs using machine learning. We show that leveraging knowledge gained from previous tuning efforts to assist in the tuning of others can significantly reduce the amount of time and resources needed to tune a DBMS for a new application.
# Contents

1 Introduction ................................................................. 1

2 Tuning Database Configuration Knobs with Machine Learning .............. 4
   2.1 System Overview ...................................................... 5
   2.2 Machine Learning Pipeline .......................................... 5
       2.2.1 Workload Characterization .................................. 6
       2.2.2 Knob Identification ............................................ 6
       2.2.3 Automatic Tuning ............................................... 7
   2.3 Evaluation ............................................................. 7
       2.3.1 Workloads ....................................................... 8
       2.3.2 Training Data Collection ...................................... 8
       2.3.3 Tuning Evaluation .............................................. 9
       2.3.4 Efficacy Comparison ........................................... 10

3 Tuning in the Real World .................................................. 14
   3.1 Motivation ............................................................. 15
       3.1.1 Workload ....................................................... 15
       3.1.2 DBMS ........................................................... 16
       3.1.3 Operating Environment ....................................... 17
   3.2 Algorithms ........................................................... 17
       3.2.1 GPR ............................................................. 17
       3.2.2 DNN ............................................................ 18
       3.2.3 DDPG .......................................................... 18
       3.2.4 LHS ............................................................. 18
   3.3 Evaluation ............................................................. 19
4 Proposed Work

4.1 Micro Improvements ......................................................... 21
4.2 Macro Improvements ...................................................... 22
4.3 Contextual Bandits ......................................................... 23

5 Timeline .......................................................................... 24

Bibliography ...................................................................... 25
Chapter 1

Introduction

Database management systems (DBMSs) are an essential component of any data-intensive application. But tuning a DBMS to perform well is a notoriously difficult task because they have hundreds of configuration knobs that control aspects of their runtime behavior, such as cache sizes and how frequently to flush data to disk. Getting the right configuration for these knobs is hard because they are not standardized (i.e., sets of knobs for different DBMSs vary), not independent (i.e., changing one knob may alter the effects of others), and not uniform (i.e., the optimal configuration depends on the target workload and hardware). Tuning these knobs is such a critical task for database applications because the difference in performance between good and bad settings is substantial. This means that organizations often hire expert database administrators (DBAs) to manually tune their DBMSs, but they are prohibitively expensive for many. Furthermore, as databases grow in both size and complexity, optimizing a DBMS to meet the needs of new applications has surpassed the abilities of even the best human experts.

Given this, there is strong interest in automatic techniques for tuning a DBMS. Most of the knob configuration tools available today were created by vendors, and thus they only support that particular company’s DBMS [16, 20, 8]. A small number of external tools support multiple DBMSs using black-box methods [11, 21, 9, 24, 25, 17]. There are two general approaches that these tools use to tune knobs automatically. The first is to use heuristics (i.e., static rules) based on the expertise and experience of human DBAs that are manually created and maintained by the tool developers [8, 16, 2, 4, 26]. These tools, however, are unable to fully optimize a DBMS’s knobs. This is partly because they only target 10 – 15 knobs believed to have the largest impact on performance. It is also because the rules are unable to capture the nuances of the workload (e.g., read/write mixtures, cyclic access patterns).

The second approach is to use machine learning (ML) techniques that automatically learn how to configure knobs for a given application based on real observations of a DBMS’s performance [24, 11, 25, 17, 15]. ML-based tools achieve better performance than rule-based tools because they are able to optimize more knobs and account for the inherent dependencies between them. The downside is that ML-based tools need a large amount of training data to achieve good results, and collecting this data is costly and time-consuming. The DBA must first prepare a copy of the application’s database and derive a representative workload sample. The tuning tool then runs trials with this workload on a separate test system so that it does not interfere with the produc-
tion DBMS. Depending on the duration of the workload sample, it could take days or even weeks to collect sufficient training data. Optimizing each DBMS independently in this manner is horribly inefficient and infeasible for deployments with hundreds or thousands of databases.

This thesis seeks to address the challenge of developing effective yet practical techniques for the automatic configuration of DBMSs using machine learning. In particular, we aim to improve the data efficiency of the machine learning models to reduce the amount of time and resources needed to generate a near-optimal configuration for a new DBMS deployment. Although the focus of this thesis is on DBMS configuration tuning, many of our solutions can be applied to other optimization problems with expensive black-box functions.

This thesis provides evidence to support the following statement:

**Thesis Statement:** Leveraging runtime data collected from previous tuning efforts can enable an ML-based automatic tuning service to optimize a DBMS’s knobs for a new application in less time and with fewer resources.

We have completed the following contributions for this thesis:

**C1.** A DBMS configuration tuning service called OtterTune that automates the task of finding good settings for a DBMS’s configuration knobs. The latest version of OtterTune supports three DBMSs (MySQL, Postgres, Oracle) and several techniques for optimizing their knob configurations. OtterTune is available as an open-source, and its extensible architecture makes it easy to support new DBMSs and tuning techniques. (Chapter 2, Section 2.1)

**C2.** A technique that reuses training data gathered from previous tuning sessions to tune new DBMS deployments. Instead of starting each new session with no prior knowledge, the algorithm determines which of the workloads tuned in the past are similar to the target workload and then reuses this previous data to “bootstrap” the new tuning session. Reusing previous data reduces the amount of time and resources needed to tune a DBMS for a new application. (Chapter 2, Section 2.2)

**C3.** A field study and evaluation of the efficacy of three state-of-the-art ML-based tuning tools on a 1 TB production database executing a real-world workload trace. Based on our experiences and findings, we provide guidelines and optimizations to help mitigate them (when possible). (Chapter 3)

C1 – C3 reduce the time needed by automatic ML-based DBMS configuration tools to tune new database workloads and also show that such tools can effectively optimize a real-world workload. Through our field study, however, we learned that tuning every database is not always practical. Therefore, our proposed work seeks opportunities to adapt our technique from C2 into an advisory tool that recommends knob configurations using information observed passively from the target database without actually tuning it.

We propose to explore the following ideas to complete this thesis:

**P1.** Micro Improvements: An advisory tool must correctly identify the most similar database workload(s) tuned in the past with minimal information from the target database. This is because the existing configurations for these similar workloads are good candidates for potentially improving the performance of the target database. Thus, we seek to refine the individual
components that collectively form our existing technique since a bad output in one component impacts all subsequent ones. (Chapter 4, Section 4.1)

**P2.** Macro Improvements: P1 identifies which of the optimized configurations from past tuning efforts are most likely to improve the target database’s performance. An equally important task is deciding whether to recommend any of them. That is, the advisory tool must estimate the expected improvement in performance from applying the configurations selected by P1 over the target database’s current configuration. We will approach this problem with a holistic view of our technique since factors from all components must be considered. (Chapter 4, Section 4.2)

**P3.** Contextual Bandits: Our current workload mapping technique is a pipelined approach. A disadvantage of such approaches is that the final output may be inaccurate if any of the assumptions made by the models in the pipeline are not satisfied. As such, we will explore an alternative technique called contextual bandits that would provide an end-to-end (i.e., non-pipelined) solution. (Chapter 4, Section 4.3)
Chapter 2

Tuning Database Configuration Knobs with Machine Learning

In this chapter, we present OtterTune, an automatic DBMS tuning service that reuses training data gathered from previous sessions to tune new DBMS deployments. To do this, OtterTune trains ML models from historical performance data, and then use the models to (1) select the most important knobs, (2) map previously unseen database workloads to known workloads so that we can transfer previous experience, and (3) recommend knob settings that improve a target objective (e.g., latency, throughput). Reusing past experiences reduces the amount of time and resources needed to tune a DBMS for a new application.

We begin this chapter with an overview of OtterTune’s architecture in Section 2.1. We then discuss OtterTune’s machine learning pipeline in Section 2.2. Finally, we evaluate the OtterTune tuning service in Section 2.3.
2.1 System Overview

OtterTune is a tuning service that automates the task of finding good settings for a DBMS’s knob configuration [24]. 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. For a new session, it uses these models to guide experimentation and recommend optimal settings. Each recommendation provides OtterTune with more information in a feedback loop that allows it to refine its models and improve their accuracy.

As shown in Figure 2.1, OtterTune’s architecture consists of a client-side controller and a server-side tuning manager. The controller acts as an agent between the target DBMS (i.e., the DBMS that the user wishes to tune) and the tuning manager. The controller collects runtime information 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 to try.

To initialize a new tuning session, the user first selects which metric should be the target objective for OtterTune to optimize when selecting configurations. OtterTune retrieves this information either from (1) the DBMS itself via its query API, (2) a third-party monitoring service, or (3) a benchmarking framework [10]. OtterTune also requests other necessary information from the user about the target DBMS at this time, such as the DBMS’s version, workload name, and connection information.

OtterTune is now ready to begin the first observation period, which is a period of time where the controller observes the target DBMS and measures its performance metrics. Once the observation period is over, OtterTune collects the runtime metrics and configuration knobs from the DBMS and delivers it to the tuning manager. The result from the first observation period serves as a baseline since it reflects the DBMS’s performance using its original configuration.

OtterTune’s tuning manager receives the result from the last observation period from the controller and stores it in its repository. Next, the tuning manager must perform the critical task in OtterTune: selecting the next configuration to try on the target DBMS. Tuning manager generates this configuration with the support of OtterTune’s machine learning pipeline and returns it to the controller. This loop between controller and tuning manager continues until the user is satisfied with the improvements over the original configuration.

2.2 Machine Learning 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
Figure 2.2: OtterTune Machine Learning Pipeline – This diagram shows the processing path of data in OtterTune. All previous observations reside in its repository. This data is first then passed into the **Workload Characterization** component that identifies the most distinguishing DBMS metrics. Next, the **Knob Identification** component generates a ranked list of the most important knobs. All of this information then fed into the **Automatic Tuner** component where it maps the target DBMS’s workload to a previously seen workload and generates better configurations.

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.

### 2.2.1 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. Pruning these redundant metrics is important because it reduces the complexity of the ML models that use them. To do this, OtterTune first uses factor analysis (FA) [5] to model each internal runtime metric as linear combinations of a few factors. It then clusters the metrics via k-means [6], 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.

### 2.2.2 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 [23] to determine which knobs have the most impact on the system’s overall performance. OtterTune applies this technique to the data in its repository 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.

### 2.2.3 Automatic 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 the smallest Euclidean distance.

Then, OtterTune chooses another knob configuration to try. It fits a Gaussian Process Regression (GPR) [19] 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).

OtterTune uses gradient descent [12] to find the configuration that optimizes the target metric (i.e., latency, throughput). Since the GPR function may be non-convex, gradient descent cannot guarantee to find the global optimum. Thus, OtterTune uses a randomly-selected group of configurations, called *initialization set*, as starting points for finding the local optimums. OtterTune recommends the configuration with the best predicted performance from these local optimums.

### 2.3 Evaluation

We now present an evaluation of OtterTune’s ability to automatically optimize the configuration of a DBMS. We implemented all of OtterTune’s algorithms using Google TensorFlow and Python’s scikit-learn.

We use MySQL (v5.6) and Postgres (v9.3) in our evaluation. Both were installed using the OS’s package manager. We did not modify any knobs in their default configurations other than to enable incoming connections from a remote IP address.

We conducted all of our deployment experiments on Amazon EC2. Each experiment consists of two instances. The first instance is OtterTune’s controller that we integrated with the OLTP-Bench framework. These clients are deployed on `m4.1large` instances with 4 vCPUs and 16 GB RAM. The second instance is used for the target DBMS deployment. We used `m3.xlarge` instances with 4 vCPUs and 15 GB RAM. We deployed OtterTune’s tuning manager and repository on a local server with 20 cores and 128 GB RAM.
We first describe OLTP-Bench’s workloads that we used in our data collection and evaluation. We then discuss our data collection to populate OtterTune’s repository. The remaining parts of this section are the experiments that showcase OtterTune’s capabilities.

2.3.1 Workloads

For these experiments, we use workloads from the OLTP-Bench testbed that differ in complexity and system demands [3, 10]:

**YCSB:** The Yahoo! Cloud Serving Benchmark (YCSB) [7] is modeled after data management applications with simple workloads and high scalability requirements. It is comprised of six OLTP transaction types that access random tuples based on a Zipfian distribution. The database contains a single table with ten attributes. We use a database with 18m tuples (∼18 GB).

**TPC-C:** This is the current industry standard for evaluating the performance of OLTP systems [22]. It consists of five transactions with nine tables that simulate an order processing application. We use a database of 200 warehouses (∼18 GB) in each experiment.

**Wikipedia:** This OLTP benchmark is derived from the software that runs the popular online encyclopedia. The database contains 11 tables and eight different transaction types. These transactions correspond to the most common operations in Wikipedia for article and “watchlist” management. We configured OLTP-Bench to load a database of 100k articles that is ∼20 GB in total size. Thus, the combination of a complex database schema with large secondary indexes makes this benchmark useful for stress-testing a DBMS.

We configure OtterTune to use five-minute observation periods and assign the target metric to be the 99%-tile latency. We did not find that shorter or longer fixed periods produced statistically significant differences in our evaluation, but applications with greater variations in their workload patterns may need longer periods.

2.3.2 Training Data Collection

As discussed in Section 2.1, OtterTune requires a corpus of previous tuning sessions that explore different knob configurations to work properly. Otherwise, every tuning session would be the first time that it has seen any application, and it would not be able to leverage the knowledge it gains from previous sessions. This means that we have to bootstrap OtterTune’s repository with initial data for training its ML models. Rather than running every workload in the OLTP-Bench suite, we created 15 variations of YCSB with different workload mixtures. All of the training data was collected using the DBMSs’ default isolation level (i.e., *REPEATABLE READ* for MySQL and *READ COMMITTED* for Postgres). We also needed to evaluate different knob configurations. For each workload, we performed a parameter sweep across the knobs using random values.

We executed a total of over 15,000 trials per DBMS using these different workloads and knob configurations. Each of these trials is treated as an observation period in OtterTune; thus, the system collects both the external metrics (i.e., throughput, latency) and internal metrics (e.g., pages read/written) from the DBMS. For each experiment, we reset OtterTune’s repository back to its
2.3.3 Tuning Evaluation

We now demonstrate how learning from previous tuning sessions improves OtterTune’s ability to find a good DBMS knob configuration. To accomplish this, we compare OtterTune with another tuning tool called iTuned [11] that also uses Gaussian Process Regression (GPR) models to search for an optimal DBMS configuration.

Unlike OtterTune, iTuned does not train its GPR models using data collected from previous tuning sessions. It instead uses a stochastic sampling technique (Latin Hypercube Sampling) to generate an initial set of 10 DBMS configurations that are executed at the start of the tuning session. iTuned uses the data from these initial experiments to train GP models that then search for the best configuration. For this comparison, we use both the TPC-C and Wikipedia benchmarks for MySQL and Postgres. OtterTune trains its GP models using the data from the most similar workload

---

Figure 2.3: Tuning Evaluation (TPC-C) – A comparison of the OLTP DBMSs for the TPC-C workload when using configurations generated by OtterTune and iTuned.

Figure 2.4: Tuning Evaluation (Wikipedia) – A comparison of the OLTP DBMSs for the Wikipedia workload when using configurations generated by OtterTune and iTuned.
mixture determined in the last workload mapping stage.

**TPC-C**: The results in Figure 2.3 show that both OtterTune and iTuned find configurations early in the tuning session that improve performance over the default configuration. There are, however, two key differences. First, OtterTune finds this better configuration within the first 30 min for MySQL and 45 min for Postgres, whereas iTuned takes 60–120 min to generate configurations that provide any major improvement for these systems. The second observation is that OtterTune generates a better configuration than iTuned for this workload. In the case of MySQL, Figure 2.3b shows that OtterTune’s best configuration achieves 85% lower latency than iTuned. With Postgres, it is 75% lower. Both approaches choose similar values for some individual knobs, but iTuned is unable to find the proper balance for multiple knobs that OtterTune does. OtterTune does a better job at balancing these knobs because its GP models have a better understanding of the configuration space since they were trained with more data.

**Wikipedia**: We next compare the two tuning approaches on MySQL and Postgres using a more complex workload. Like with TPC-C, the results in Figure 2.4 show that OtterTune has the same reduction in the transaction latency over the default configuration within the first 15 min of the Wikipedia benchmark. Postgres has a similar gradual reduction in the latency over a 100 min period. We found that again iTuned failed to generate a good configuration for the most important knobs at the beginning of its tuning session because it had to populate its initialization set. In total, OtterTune is able to achieve lower latency for both DBMSs.

### 2.3.4 Efficacy Comparison

In our last experiment, we compare the performance of MySQL and Postgres when using the best configuration selected by OtterTune versus ones selected by human DBAs and open-source tuning advisor tools. We also compare OtterTune’s configurations with those created by a cloud database-as-a-service (DBaaS) provider that are customized for MySQL and Postgres running on the same EC2 instance type as the other deployments in these experiments.

Each DBA was provided with the same EC2 setup used in all of our experiments. They were allowed to tune any knobs they wanted but were not allowed to modify things external to the DBMS (e.g., OS kernel parameters). On the client instance, we provided them with a script to execute the workload for the 5 min observation period and a general log full of previously executed queries for that workload. The DBAs were permitted to restart the DBMS and/or the workload as many times as they wanted.

For the DBaaS, we use the configurations generated for Amazon RDS. We use the same instance type and DBMS version as the other deployments in these experiments. We initially executed the workloads on the RDS-managed DBMSs, but found that this did not provide a fair comparison because Amazon does not allow you to disable the replication settings (which causes worse performance). To overcome this, we extracted the DBMS configurations from the RDS instances and evaluated them on the same EC2 setup as our other experiments. We disable the knobs that control the replication settings to be consistent with our other experiments.

**MySQL**: Our first DBA is the premiere MySQL tuning and optimization expert from Lithuania with over 15 years of experience and also works at a well-known Internet company. They finished...
Figure 2.5: **Efficacy Comparison (MySQL)** – Throughput and latency measurements for the TPC-C benchmark using the (1) default configuration, (2) OtterTune configuration, (3) tuning script configuration, (4) Lithuanian DBA configuration, and (5) Amazon RDS configuration.

Figure 2.6: **Efficacy Comparison (Postgres)** – Throughput and latency measurements for the TPC-C benchmark using the (1) default configuration, (2) OtterTune configuration, (3) tuning script configuration, (4) expert DBA configuration, and (5) Amazon RDS configuration.

tuning in under 20 min and modified a total of eight knobs.

The MySQL tuning tool (MySQLTuner [2]) examines the same kind of DBMS metrics that OtterTune collects and uses static heuristics to recommend knob configurations. It uses an iterative approach: we execute the workload and then run the tuning script. The script emits suggestions instead of exact settings (e.g., set the buffer pool size to be at least 2 GB). Thus, we set each knob to its recommended lower bound in the configuration file, restarted the DBMS, and then re-executed the workload. We repeated this until the script stopped recommending settings to further improve the configuration. This process took 45 min (i.e., eight iterations) before it ran out of suggestions, and modified five knobs.

Figure 2.5 shows that MySQL achieves approximately 35% better throughput and 60% better latency when using the best configuration generated by OtterTune versus the one generated by the tuning script for TPC-C. We see that the tuning script’s configuration provides the worst performance of all of the (non-default) configurations. The reason is that the tuning script only modifies
one of the four most impactful knobs, namely, the size of the buffer pool. The other knobs that
the tuning script modifies are the number of independent buffer pools and the query cache settings.
We found, however, that these knobs did not have a measurable effect.

Both the latency and the throughput measurements in Figure 2.5 show that MySQL achieves
\(\sim 22\%\) better throughput and \(\sim 57\%\) better latency when using OtterTune’s configuration compared
to RDS. RDS modified three out of the four most impactful knobs: the size of the buffer pool, the
size of the log file, and the method used to flush data to disk. Still, we see that the performance
of the RDS configuration is only marginally better than that of the tuning script. An interesting
finding is that RDS actually decreases the size of the log file (and other files) to be smaller than
MySQL’s default setting. We expect that these settings were chosen to support instances deployed
on variable-sized EBS storage volumes, but we have not found documentation supporting this.

OtterTune generates a configuration that is almost as good as the DBA. The DBA configured
the same three out of four top-ranking knobs as RDS. We see that OtterTune, the DBA, and RDS
update the knob that determines how data is flushed to disk to be the same option. This knob’s
default setting uses the `fsync` system call to flush all data to disk. But the setting chosen by
OtterTune, the DBA, and RDS is better for this knob because it avoids double buffering when
reading data by bypassing the OS cache. Both the DBA and OtterTune chose similar sizes for the
buffer pool and log file. The DBA modified other settings, like disabling MySQL’s monitoring
tools, but they also modified knobs that affect whether MySQL ensures that all transactions are
fully durable at commit time. OtterTune is forbidden from tuning such knobs that could have
hidden or serious consequences.

**Postgres:** For the next DBMS, our human expert was the lead DBA for a mid-western judicial
court system in the United States. They have over six years of experience and have tuned over
100 complex production database deployments. They completed their tuning task in 20 min and
modified a total of 14 knobs.

The Postgres tuning tool (PGTune [4]) is less sophisticated than the MySQL one in that it only
uses pre-programmed rules that generate knob configurations for the target hardware and does
not consider the DBMS’s metrics. We found, however, that using the Postgres tuning tool was
easier because it was based on the amount of RAM available in the system and some high-level
characteristics about the target workload (e.g., OLTP vs. OLAP). It took 30 seconds to generate
the configuration and we never had to restart the DBMS. It changed a total of eight knobs.

The latency measurements in Figure 2.6b show that the configurations generated by OtterTune,
the tuning tool, the DBA, and RDS all achieve similar improvements for TPC-C over Postgres’
default settings. This is likely because of the overhead of network round-trips between the OLTP-
Bench client and the DBMS. But the throughput measurements in Figure 2.6 show that Postgres
has \(\sim 12\%\) higher performance with OtterTune compared to the DBA and the tuning script, and
\(\sim 32\%\) higher performance compared to RDS.

Unlike our MySQL experiments, there is considerable overlap between the tuning methods
in terms of which knobs they selected and the settings that they chose for them. All of the con-
figurations tune the three knobs that OtterTune finds to have the most impact. The first of these
knobs tunes the size of the buffer pool. All configurations set the value of this knob to be between
2–8 GB. The second knob provides a “hint” to the optimizer about the total amount of memory
available in the OS and Postgres’ buffers but does not actually allocate any memory. The DBA and RDS select conservative settings of 10 GB and 7 GB compared to the settings of 18 GB and 23 GB chosen by OtterTune and the tuning script, respectively. The latter two overprovision the amount of memory available whereas the settings chosen by the DBA and RDS are more accurate.

The last knob controls the maximum number of log files written between checkpoints. Setting this knob too low triggers more checkpoints, leading to a huge performance bottleneck. Increasing the value of this knob improves I/O performance but also increases the recovery time of the DBMS after a crash. The DBA, the tuning script, and AWS set this knob to values between 16 and 64. OtterTune, however, sets this knob to be 540, which is not a practical value since recovery would take too long. The reason that OtterTune chose such a high value compared to the other configurations is a result of it using the latency as its optimization metric. This metric captures the positive impact that minimizing the number of checkpoints has on the latency but not the drawbacks of longer recovery times. We leave this problem as a goal for future work.
Chapter 3

Tuning in the Real World

Recent results from ML-based approaches have demonstrated that they achieve better performance than human DBAs and other state-of-the-art tuning tools on a variety of workloads and hardware configurations [25, 17]. Although these results are promising, up until now, the evaluations have been limited to (1) open-source DBMSs (Postgres, MySQL) with limited tuning potential and (2) synthetic benchmarks (TPC-C, YCSB, Sysbench) with uniform workload patterns. Additionally, even though these evaluations used virtualized environments (i.e., cloud), to the best of our knowledge, they all used dedicated local storage to the DBMS (i.e., SSDs attached to the VM). Many real-world DBMS deployments, however, use non-local shared storage for data and logs. Such non-local storage includes on-premise SANs and cloud-based block stores (e.g., Amazon EBS, Azure Disk Storage). In addition to incurring higher overall read/write latencies than local storage, these non-local storage devices also have more variance in their performance. It is unclear how these differences affect (if at all) the efficacy of ML-based tuning algorithms.

Given this, this chapter presents a field study of automatic knob configuration tuning algorithms on a commercial DBMS with a real-world workload in a production environment. We provide a comprehensive evaluation of state-of-the-art ML-based methods for optimizing the configuration of an Oracle DBMS (v12) installation running on virtualized computing infrastructure with non-local storage. For this work, we extended the open-source OtterTune tuning service (Chapter 2, Section 2.1) to support four tuning algorithms: (1) Gaussian Process Regression from the original version of OtterTune (Chapter 2, Section 2.2), (2) Deep Neural Network, (3) Deep Deterministic Policy Gradient from CDBTune [25], and (4) Latin Hypercube Sampling. As we show later in this chapter, the preliminary results from this effort are promising.

The remainder of this chapter is organized as follows. Section 3.1 begins with a discussion to motivate this field study. We then provide an overview of the state-of-the-art tuning algorithms that we evaluate in Section 3.2. Finally, we evaluate the four algorithms in Section 3.3.
3.1 Motivation

Recent studies have shown that ML-based configuration tuning approaches can achieve better performance compared to human DBAs and other tuning tools on a variety of workloads and hardware configurations [24, 25, 17]. Again, all of these results are promising, but we observe that there is a mismatch between aspects of the evaluations in those previous studies versus what we see in real-world DBMS deployments. In particular, the three facets that are part of this mismatch are the (1) workload, (2) DBMS, and (3) operating environment. We now discuss them in further detail.

3.1.1 Workload

WIP...

TPC-C: This is the current industry standard for evaluating the performance of OLTP systems [22]. It consists of nine tables that simulate an order processing application.

TicketTracker: This is a real-world application used internally at Societe Generale to track the assignment, status, and progress of tickets submitted by employees to their technical support group. Examples of tickets include requests to provision VMs and upgrade software. The production database contains 933 tables and is over a terabyte in total size.

<table>
<thead>
<tr>
<th></th>
<th>TicketTracker</th>
<th>TPC-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Size Breakdown</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td>27%</td>
<td>79%</td>
</tr>
<tr>
<td>Index</td>
<td>19%</td>
<td>21%</td>
</tr>
<tr>
<td>LOB</td>
<td>54%</td>
<td>0%</td>
</tr>
<tr>
<td># Tables</td>
<td>933</td>
<td>9</td>
</tr>
<tr>
<td># Indexes Per Table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Min</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Med</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Max</td>
<td>77</td>
<td>3</td>
</tr>
<tr>
<td># Columns Per Table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>Min</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Med</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Max</td>
<td>599</td>
<td>21</td>
</tr>
<tr>
<td>Distribution of Query Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select</td>
<td>90%</td>
<td>54%</td>
</tr>
<tr>
<td>Update</td>
<td>6%</td>
<td>30%</td>
</tr>
<tr>
<td>Insert</td>
<td>3%</td>
<td>10%</td>
</tr>
<tr>
<td>Delete</td>
<td>1%</td>
<td>6%</td>
</tr>
</tbody>
</table>

Table 3.1: Database Workload Comparison
### 3.1.2 DBMS

![Graph showing throughput measurements for MySQL and Postgres](image)

**Figure 3.1: DBMS Tuning Comparison (TPC-C)** – Throughput measurements for the TPC-C benchmark running on three different versions of MySQL (v5.6, v5.7, v8.0) and Postgres (v9.3, v10, v12) using the (1) default configuration, (2) buffer pool & redo log configuration, (3) GPR configuration, and (4) DDPG configuration.

When tuning MySQL and Postgres for the TPC-C benchmark, 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. To highlight this in further detail, we ran a series of experiments using MySQL (v5.6, v5.7, and v8.0) and Postgres (v9.3, v10, and v12) that execute the TPC-C workload. Each experiment consists of two machines with 8 cores and 32 GB RAM. The first machine is running OtterTune’s controller and tuning manager, and the second is used for the target DBMS deployment. For each DBMS, we compare the throughput measurements of four different knob configurations: (1) the default for the DBMS, (2) the recommended settings from the DBMS’s documentation for the two knobs that control the buffer pool size and the redo log file size, (3) the configuration recommended by OtterTune’s Gaussian Process Regression (GPR) algorithm, and (4) the configuration recommended by CDBTune’s Deep Deterministic Policy Gradient (DDPG) algorithm [25]. We present the details of DDPG in Section 3.2.3.

Figure 3.1 shows that the two-knob configuration and the configurations generated by GPR and DDPG all greatly improve the performance for the TPC-C benchmark over the DBMS’s default settings. This is expected since the default configurations for MySQL and Postgres are set to accommodate the minimal hardware requirements specified for each of their versions. More importantly, for TPC-C, the configurations generated by GPR and DDPG achieve only 5–25% higher throughput than the two-knob configuration across the different versions of MySQL and Postgres. Additional research is needed to determine whether this is due to the simplicity of the benchmark, the DBMSs, or both.
3.1.3 Operating Environment

Although the previous studies used virtualized environments (i.e., cloud) to evaluate their techniques, to our knowledge they all deploy the DBMS on dedicated storage that is physically attached to the host machine. In contrast, many real-world DBMS deployments use non-local, shared storage for data and logs, such as on-premise SANs and cloud-based block/object stores. A common problem with these non-local storage devices is that performance can vary substantially and become highly unpredictable in a multi-tenant environment. To demonstrate this point, we measured the I/O latency on both local and non-local storage devices every 30 minutes over a three-day period for four I/O workloads generated with the FIO tool [1]: (1) sequential reads, (2) sequential writes, (3) random reads, and (4) random writes. For the local storage experiments, we use a machine with a 228 GB SSD. We present the details of the production environment used to conduct the non-local storage experiments in Section 3.3. The results in Figure 3.2 show that the read/write latencies for the non-local storage are generally higher and more variable than on the local storage device. We also see that on the non-local storage, the fluctuations in the read/write latencies increase halfway through the experiment and experience a significant spike at the end of the third day. It is unclear whether such noise in the I/O performance impacts the quality of the configurations recommended by ML-based tuning algorithms.

3.2 Algorithms

We now provide a high-level description of three algorithms: (1) Gaussian Process Regression (GPR), (2) Deep Neural Network (DNN), (3) Deep Deterministic Policy Gradient (DDPG), and (4) Latin Hypercube Sampling (LHS). Although there are other tuning algorithms that use the application’s workload to guide the search process [17], they are not usable to us in a real-world deployment. This restriction is because of privacy concerns that the queries contain user-identifiable data. Methods to sufficiently anonymize this data are outside the scope of this paper.

3.2.1 GPR

GPR is a machine learning algorithm that uses Gaussian process as a prior over functions [19]. It calculates the distance between the test point and all training points with the kernel functions to
predict the output of the given test point as well as its uncertainty. OtterTune’s GPR algorithm is the original tuning technique introduced in Chapter 2, Section 2.2. The iTuned tool also uses GPR for automatic DBMS knob configuration [11].

### 3.2.2 DNN

DNN is a deep learning algorithm that applies a series of linear combinations and non-linear activations to the input to derive the output. OtterTune’s DNN algorithm follows the same ML pipeline from Chapter 2, Section 2.2 except that it uses a DNN model instead of a GPR model in the configuration recommendation step. The network structure of OtterTune’s 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. OtterTune uses a popular technique called dropout regularization to address the overfitting problem with DNNs. Specifically, there is a dropout layer between the two hidden layers with a dropout rate of 0.5. OtterTune also adds Gaussian noise to the parameters of the neural network during knob recommendation step [18]. Adjusting the scale of this noise controls the amount of exploration versus exploitation.

### 3.2.3 DDPG

DDPG is a deep reinforcement learning algorithm that aims to find the optimal policy in an environment with continuous action space. OtterTune supports the DDPG algorithm proposed by CDBTune [25]. DDPG consists of an actor, a critic and a replay memory. In DBMS tuning, 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, which is trained by the critic, takes the previous database metrics as the 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, the DDPG algorithm first calculates the reward by comparing the current, the previous, and the base target objective values. It then inserts the tuple of `<previous metrics \( m_{prev} \), current knob \( k \), current reward \( r \), and current metrics \( m \) >` into the replay memory. Next, the algorithm fetches a mini-batch of the top ranked tuples from the replay memory, and updates the actor and critic weights with backpropagation. Finally, 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.

### 3.2.4 LHS

Latin Hypercube Sampling (LHS) [13] comes from the group of space-filling sampling techniques that attempt to distribute sample points more evenly across all possible values. Such techniques are generally more effective than simple random sampling in high-dimensional spaces, especially when collecting relatively few samples. When sampling a function of \( D \) parameters (i.e., dimensions), LHS first splits the range of each parameter into \( N \) equally probable intervals. It then selects one random point from each interval such that no two samples hit the same interval across the \( D \) parameters. The iTuned tool uses LHS for initialization before applying GPR [11].
3.3 Evaluation

To be written...
Chapter 4

Proposed Work

The work we have completed indicates that one can achieve more efficient tuning of new database workloads by learning from past experiences. In our deployment at Societe Generale, we evaluate the effectiveness of three ML-based techniques on a real-world workload. One assumption we made in both of these studies is that an organization can tune each database by copying the data onto a separate test machine with identical hardware and replaying a representative workload sample. But based on the feedback we received from this field study and our other industry partners, we now understand that such an arrangement is not always possible due to logistical and monetary constraints. The key takeaway is that tuning every database is not always practical.

Given this, we can consider other less-obtrusive tuning strategies. We organize these strategies into three levels, where increasing levels are capable of more custom tuning but also require a greater tuning effort.

**Level #1 – Advisory:** The lowest level requires the least amount of tuning effort but does not provide any custom tuning for the target database (i.e., no feedback loop). Instead, this strategy observes the metrics from the target database and maps them to an existing configuration that was custom tuned for a database with a similar workload in the past.

**Level #2 – Online:** The next level tunes a production database (replica) in an online fashion, which requires a moderate amount of tuning effort. This strategy is risky because bad configurations selected by the tool could interfere with the production DBMS and violate service-level agreements (SLAs). Organizations can mitigate some of this risk by tuning a replica instead of the master database. Another aspect of this strategy is that the tuner can only optimize “dynamic” knobs that do not require a restart since downtime of the production database is not allowed. The degree of custom tuning depends on the quantity and quality of the dynamic knobs supported by the target DBMS.

**Level #3 – Offline:** At the highest level, the DBA replicates the hardware, data, and workload of the production database on a testing platform and then performs the tuning offline. This is what we have assumed in our earlier work. This strategy requires the most tuning effort but can custom tune all of the knobs since restarts are permitted. The most challenging aspect of offline tuning is having access to a tool that is capable of replaying a production workload trace. Many commercial DBMSs include such tools to assist with this task. The same is not true for open-source DBMSs,
and implementing a replay tool in-house is a major engineering effort.

All of our work so far has focused on offline tuning. Although online tuning presents some interesting challenges, its efficacy depends on the capabilities of the target DBMS. As such, we will shift the focus of our research to address the challenges of tuning at the advisory level. We can already provide a basic advisory tool with OtterTune’s current workload mapping technique from Chapter 2, Section 2.2.3 as the scenario is equivalent to stopping the tuning session after the mapping step following the first observation period. With advisory tuning, however, getting the mapping right is critical since a poor mapping could cause performance problems on the production database.

My proposed work, therefore, seeks opportunities to adapt our current workload mapping technique into develop an effective advisory tool. We will first explore methods to improve our technique at the micro level (i.e., improving the individual components in our technique), and then at the macro level (i.e., holistic improvements to the overall technique). We also propose to investigate the possibility of a contextual bandits approach as an alternative to our workload mapping technique. We now discuss these three areas.

4.1 Micro Improvements

The purpose of OtterTune’s workload mapping technique is to identify which of the database workloads it has tuned in the past is the most similar to the target workload. OtterTune then trains a GP model with the data from both the past workload and the target to improve its predictions when selecting the next configuration. We recompute the workload mapping every iteration to incorporate new data from the target workload. We find that the mapping is inaccurate for the first 3–4 iterations but then improves quickly and thus does not impact the tuning result. But for advisory-level tuning, we only observe the metrics for the configuration installed on the target database. Computing an accurate mapping is challenging due to the limited data available from the target database.

Given this, we propose to explore methods to improve the individual components that collectively form our existing workload mapping technique. First, we will attempt to exploit the limited information we can get from the target database by collecting metrics over multiple observation periods. We hypothesize that, for example, taking the median of these metric values will reduce the amount of noise in the data.

Next, we will use model validation techniques to assess the prediction performance of the statistical models that we use to predict the metrics for knob configurations attempted by the target workload that are missing from our past workloads. Based on these results, we may explore the use of Ensemble methods to improve the quality of our predictions. We will also examine the impact of the amount and diversity of the training data from the past workloads on prediction performance. We suspect that training the statistical models with combined data from similar past workloads will improve the prediction quality.

We will then investigate weighting the metrics in the distance calculation. We expect that assigning a larger weight to the target objective than the pruned metrics will improve the mapping
Finally, to determine the best method for selecting the knob configuration to install on the target database, we will investigate two methods: (a) using the exact configuration from the most similar past workload, and (b) recommending the next configuration using the limited data available for the target workload and data from the most similar past workload. When tuning online and offline, OtterTune’s recommendation algorithms make a trade-off between exploration and exploitation to optimize the knob configuration over the tuning session. For (b), it is important that the recommendation algorithms use only the exploitation strategy to select a configuration close to the best known configuration for the most similar workload tuned in the past.

4.2 Macro Improvements

When tuning a database, an advisory tool must determine not only the best configuration to map but also the expected improvement in the target objective over the database’s current configuration. Getting this right is critical because the mapped configuration should only be installed on the target database if we are confident that it will improve the performance. Unfortunately, deciding whether to install the mapped configuration is nontrivial, as it depends on several factors. The advisory tool must consider all of these factors when determining whether to install a mapped configuration onto the target database.

The workload mapping algorithm first trains a statistical model for each of the previously-tuned workloads to predict the metric values for the target database’s current knob configuration. These models are necessary since, due to the large search space, it is unlikely that the same configuration was attempted while tuning the previous workload. The algorithm then selects the most similar workload by comparing the target workload’s metric values with the predicted ones for each of the previous workloads. Thus, the correctness of the similarity calculations depends on the accuracy of the model’s predicted metric values. Two factors can impact the accuracy of these estimates: the prediction error of the model and the uncertainty in the predicted metric values. Given this, we will explore policies for handling inaccurate predictions caused by these factors, such as omitting previous workloads from further consideration or accounting for the prediction error and uncertainty in the distance calculations.

Once the workload mapping step finishes, the advisory tool must then decide whether to install the knob configuration from the most similar workload onto the target database. To do this, it needs to estimate the expected improvement in performance from applying the mapped configuration. Two factors that influence this estimation are the degree of similarity between the target workload and the most similar previous workload and amount of noise in the target workload data. We will investigate methods to estimate the expected improvement given these factors.
4.3 Contextual Bandits

Our current workload mapping technique uses the Euclidean distance calculation to determine which past workload is the most similar to the target workload. This technique is straightforward but only incorporates knowledge from a single past workload to help tune others. Contextual bandits is an approach that is similar to OtterTune’s current recommendation algorithm that uses GPR, but it includes context about the environment or other factors to improve the optimization [14].

We propose to explore the possibility of a better workload mapping technique using contextual bandits. This approach would allow us to use the data from all past workloads to select a customized configuration for the target workload, even if there is limited data available. This method could also be used to custom tune a database both online and offline. A general problem with GPR approaches is that they are less effective at optimizing high-dimensional continuous spaces than other methods such as reinforcement learning. Adding context can further exacerbate this problem.
Chapter 5

Timeline

Dec 2019
June 2020
July 2020
Aug 2020
Sep 2020
Oct 2020

Tuning in the Real World
Propose
SIGMOD Deadline
Micro Improvements
Macro Improvements
Contextual Bandits
Write Thesis
Defend
Bibliography


