On Automatic Database Management System Tuning Using Machine Learning

Dana Van Aken

CMU-CS-21-104

February 2021

School of Computer Science
Computer Science Department
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.

This research was sponsored by National Science Foundation award numbers: IIS-1846158, IIS-1423210, and CCF-1438955; by a National Science Foundation Graduate Research Fellowship award; by the Intel Corporation award number 1011856; and by the University Industry Research Corporation award numbers: 1030834 and 1030845. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: database management systems, machine learning, database tuning, configuration tuning

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.

Acknowledgments

I would like to thank my husband for standing by me through out my entire Ph.D. career. I could not have done this without his support.

I would also like to thank my thesis committee for their guidance and feedback. Their help not only made me a better researcher, but it also taught me to be a better person.

Contents

Abstract 3

Acknowledgments 5

1 Introduction 18

2 OtterTune Tuning Service 22
2.1 Runtime Metrics 22
2.2 Configuration Knobs 23
2.3 System Overview 23
2.4 Configuration Tuning Procedure 24
2.5 Statistics Collection 25
2.6 Assumptions & Limitations 25

3 Tuning via Gaussian Process Regression 27
3.1 Workload Characterization 27
3.1.1 Pruning Redundant Metrics 28
3.2 Identifying Important Knobs 30
3.2.1 Feature Selection with Lasso 31
3.2.2 Dependencies 32
3.2.3 Incremental Knob Selection 33
3.3 Automated Tuning 33
3.3.1 Step #1 – Workload Mapping 33
3.3.2 Step #2 – Configuration Recommendation 34
3.4 Experimental Evaluation 36
3.4.1 Workloads 36
3.4.2 Training Data Collection 38
3.4.3 Number of Knobs 39
3.4.4 Tuning Evaluation 40

3.4.5 Execution Time Breakdown . . . 42
3.4.6 Efficacy Comparison . . . 43

4 Tuning in the Real World 47
4.1 Motivation . . . 48
4.2 Automated Tuning Field Study . . . 50
4.2.1 Target Database Application . . . 51
4.2.2 Deployment . . . 52
4.2.3 Tuning . . . 54
4.3 Tuning Algorithms . . . 56
4.3.1 DNN — OtterTune (2019) . . . 56
4.3.2 DDPG — CDBTune (2019) . . . 57
4.4 Evaluation . . . 57
4.4.1 Performance Variability . . . 58
4.4.2 Tuning Knobs Selected by DBA . . . 60
4.4.3 Tuning Knobs Ranked by OtterTune . . . 63
4.4.4 Adaptability to Different Workload . . . 65
4.4.5 Execution Time Breakdown . . . 65
4.5 Lessons Learned . . . 67

5 Advisory-Level Tuning 70
5.1 Taxonomy . . . 70
5.1.1 Level #1 – Advisory . . . 71
5.1.2 Level #2 – Online . . . 71
5.1.3 Level #3 – Offline . . . 71
5.2 Workloads . . . 72
5.3 Motivation . . . 73
5.4 Workload Mapping . . . 74
5.4.1 Optimization #1 — Hyperparameter Tuning . . . 75
5.4.2 Optimization #2 — Static Metrics . . . 77
5.5 Contextual Bandits . . . 77
5.6 Evaluation . . . 78
5.6.1 Workload Mapping . . . 78
5.6.2 One-Shot — Workload Models . . . 79
5.6.3 One-Shot — CB Algorithms . . . 81
5.7 Lessons Learned . . . 82

6 Related Work **85**
6.1 Physical Database Design 85
6.2 Configuration Tuning for Databases 86
6.3 Configuration Tuning for Data Analytics Systems 89

7 Future Work **90**

A Tuning via Gaussian Process Regression **92**
A.1 Identifying Important Knobs 92
A.2 Efficacy Comparison 94

B Tuning in the Real World **98**

Bibliography **121**

List of Figures

1.1 **Motivating Examples** – Figures 1.1a to 1.1c show performance measurements for the YCSB workload running on MySQL (v5.6) using different configuration settings. Figure 1.1d shows the number of tunable knobs provided in MySQL and Postgres releases over time. 19

2.1 **OtterTune Architecture** – An overview of the components in the OtterTune system. The controller connects to the DBMS and collects information about the performance of the system. This information is then sent to the tuning manager where it is stored in its repository. It then builds models that are used to select an optimal configuration for the DBMS. 24

3.1 **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** (Section 3.1) component that identifies the most distinguishing DBMS metrics. Next, the **Knob Identification** (Section 3.2) component generates a ranked list of the most important knobs. All of this information then fed into the **Automatic Tuner** (Section 3.3) component where it maps the target DBMS's workload to a previously seen workload and generates better configurations. 28

3.2 **Metric Clustering** – Grouping DBMS metrics using k-means based on how similar they are to each other as identified by Factor Analysis and plotted by their (`f1`, `f2`) coordinates. The color of each metric shows its cluster membership. The triangles represent the cluster centers. 31

3.3 **Number of Knobs** – The performance of the DBMSs for TPC-C and TPC-H during the tuning session using different configurations generated by OtterTune that only configure a certain number of knobs. 37

3.4 **Tuning Evaluation (TPC-C)** – A comparison of the OLTP DBMSs for the TPC-C workload when using configurations generated by OtterTune and iTuned. 40

3.5 **Tuning Evaluation (Wikipedia)** – A comparison of the OLTP DBMSs for the Wikipedia workload when using configurations generated by OtterTune and iTuned. 41

3.6 **Tuning Evaluation (TPC-H)** – Performance measurements for Vector running two sub-sets of the TPC-H workload using configurations generated by OtterTune and iTuned. 41

3.7 **Execution Time Breakdown** – The average amount of time that OtterTune spends in the parts of the system during an observation period. 43

3.8 **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) DBA configuration, and (5) Amazon RDS configuration. 44

3.9 **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. 44

4.1 **DBMS Tuning Comparison** – Throughput measurements for the TPC-C benchmark running on three versions of MySQL (v5.6, v5.7, v8.0) and Postgres (v9.3, v10.1, v12.3) using the (1) default configuration, (2) buffer pool & redo log configuration, (3) GPR configuration, and (4) DDPG configuration. 48

4.2 **Operating Environment** – I/O latency of local versus non-local storage for four different I/O workloads over a three-day period. 49

4.3 **Database Contents Analysis** – The number of tuples, columns, and indexes per table for the TicketTracker database. 53

4.4 **TicketTracker Workload Analysis** – Execution information for the TicketTracker queries extracted from the workload trace. 54

4.5 **DDPG Tuning Pipeline** – The raw data is converted to states, actions and rewards and then inserted into the replay memory. The tuples in the replay memory are ranked by the error of the predicted Q-value. In the training process, a batch of top tuples are fetched to update the critic and the actor. After training, the prediction error in the replay memory is updated and the actor recommends the next configuration to run. 55

4.6 **Performance Variability** – Performance for the TicketTracker workload using the default configuration on multiple VMs over six months. 58

4.7 **Effect of I/O Latency Spikes** – Runtime measurements of DBMS performance with CPU utilization and I/O latency. 59

4.8 **Tuning Knobs Selected by DBA (Per VM)** – The performance improvement of the best configuration per algorithm running on separate VMs relative to the performance of the SG default configuration measured at the beginning of the tuning session. 60

4.9 **Tuning Knobs Selected by DBA** – Performance measurements for 10, 20, and 40 knob configurations for the TicketTracker workload. The shading on each bar indicates the minimum and maximum performance of the best configurations from three tuning sessions. 61
4.10 **Tuning Knobs Ranked by OtterTune (Per VM)** – The performance improvement of the best configuration per algorithm running on separate VMs relative to the performance of the SG default configuration measured at the beginning of the tuning session. 62
4.11 **Tuning Knobs Ranked by OtterTune** – Performance measurements for the ML algorithm configurations using 10 and 20 knobs selected by OtterTune's Lasso ranking algorithm. The shading on each bar indicates the min and max performance of the best configurations from three tuning sessions. 63
4.12 **Adaptability to Different Workloads** – Performance comparison when applying the model trained on TPC-C data to the TicketTracker workload. 66

5.1 **Workload Tuning Comparison (MySQL v8.0)** – Throughput measurements for each workload running on MySQL (v8.0) using the (1) default configuration, (2) buffer pool & redo log configuration, (3) MySQL's dedicated server flag, and (4) OtterTune's configuration. 75
5.2 **Swapping Optimized Configurations (MySQL v8.0)** – Throughput measurements for the workloads when using the optimized configurations from all other workloads. The striped bar for each workload indicates its the optimal configuration for that workload. 76
5.3 **One-Shot Configurations** – Configurations recommended by models trained on data from the most similar past workload determined in the workload mapping step. 83
5.4 **CB One-Shot Configurations** – Configurations recommended by CB models trained on all previous workload data with 16 DBMS runtime metrics as the workload context. 84

A.1 Lasso Path (MySQL) 92

A.2 Lasso Path (Postgres) 93
A.3 Lasso Path (Vector) 94

List of Tables

4.1 **Query Plan Operators** – The percentage of queries in the TicketTracker workload that contain each operator type. 52
4.2 **Most Important Knobs** – The three most important knobs for the TicketTracker workload with their default and best observed values. 62
4.3 **Execution Time Breakdown** – The median amount of time spent in different parts of the system during a tuning iteration. 66
4.4 **Workload Replay Time per Algorithm** – The median workload execution time and the percentage of replays canceled for the algorithms. 67

5.1 **Workload Characteristics** 72
5.2 **Workload Mapping (all metrics)** – The distance measurements between workloads computed by the original workload mapping technique using all 86 pruned metrics. A smaller distance indicates the workload is more similar. The distance between a given workload and itself is shown in gray. 80
5.3 **Workload Mapping (eight pruned metrics)** – The distance measurements between workloads computed by the original workload mapping technique using eight pruned metrics. A smaller distance indicates the workload is more similar. The distance between a given workload and itself is shown in gray. 80
5.4 **Workload Mapping (eight pruned metrics + eight static metrics)** – The distance measurements between workloads computed by the optimized workload mapping technique using eight pruned metrics and eight static metrics. A smaller distance indicates the workload is more similar. The distance between a given workload and itself is shown in gray. 81

A.1 **Efficacy Comparison – DBA Configuration (MySQL)** 95
A.2 **Efficacy Comparison – OtterTune Configuration (MySQL)** 95
A.3 **Efficacy Comparison – Amazon RDS Configuration (MySQL)** 95
A.4 **Efficacy Comparison – Tuning Script Configuration (MySQL)** 96
A.5 **Efficacy Comparison – DBA Configuration (Postgres)** 96
A.6 **Efficacy Comparison – OtterTune Configuration (Postgres)** 96
A.7 **Efficacy Comparison – Amazon RDS Configuration (Postgres)** 97

A.8 Efficacy Comparison – Tuning Script Configuration (Postgres) 97

B.1 Tuning Knobs Selected by DBA – GPR (10 knobs) 98
B.2 Tuning Knobs Selected by DBA – DNN (10 knobs) 99
B.3 Tuning Knobs Selected by DBA – DDPG (10 knobs) 99
B.4 Tuning Knobs Selected by DBA – DDPG++ (10 knobs) 99
B.5 Tuning Knobs Selected by DBA – LHS (10 knobs) 100
B.6 Tuning Knobs Selected by DBA – GPR (20 knobs) 100
B.7 Tuning Knobs Selected by DBA – DNN (20 knobs) 101
B.8 Tuning Knobs Selected by DBA – DDPG (20 knobs) 102
B.9 Tuning Knobs Selected by DBA – DDPG++ (20 knobs) 103
B.10 Tuning Knobs Selected by DBA – LHS (20 knobs) 104
B.11 Tuning Knobs Selected by DBA – GPR (40 knobs) 105
B.12 Tuning Knobs Selected by DBA – DNN (40 knobs) 106
B.13 Tuning Knobs Selected by DBA – DDPG (40 knobs) 107
B.14 Tuning Knobs Selected by DBA – DDPG++ (40 knobs) 108
B.15 Tuning Knobs Selected by DBA – LHS (40 knobs) 109
B.16 Tuning Knobs Ranked by OtterTune – GPR (10 knobs) 110
B.17 Tuning Knobs Ranked by OtterTune – DNN (10 knobs) 110
B.18 Tuning Knobs Ranked by OtterTune – DDPG (10 knobs) 110
B.19 Tuning Knobs Ranked by OtterTune – DDPG++ (10 knobs) 111
B.20 Tuning Knobs Ranked by OtterTune – LHS (10 knobs) 111
B.21 Tuning Knobs Ranked by OtterTune – GPR (20 knobs) 112
B.22 Tuning Knobs Ranked by OtterTune – DNN (20 knobs) 113
B.23 Tuning Knobs Ranked by OtterTune – DDPG (20 knobs) 114
B.24 Tuning Knobs Ranked by OtterTune – DDPG++ (20 knobs) 115
B.25 Tuning Knobs Ranked by OtterTune – LHS (20 knobs) 116
B.26 Adaptability to Different Workloads – GPR (20 knobs) 117
B.27 Adaptability to Different Workloads – DNN (20 knobs) 118
B.28 Adaptability to Different Workloads – DDPG (20 knobs) 119
B.29 Adaptability to Different Workloads – DDPG++ (20 knobs) 120

Chapter 1

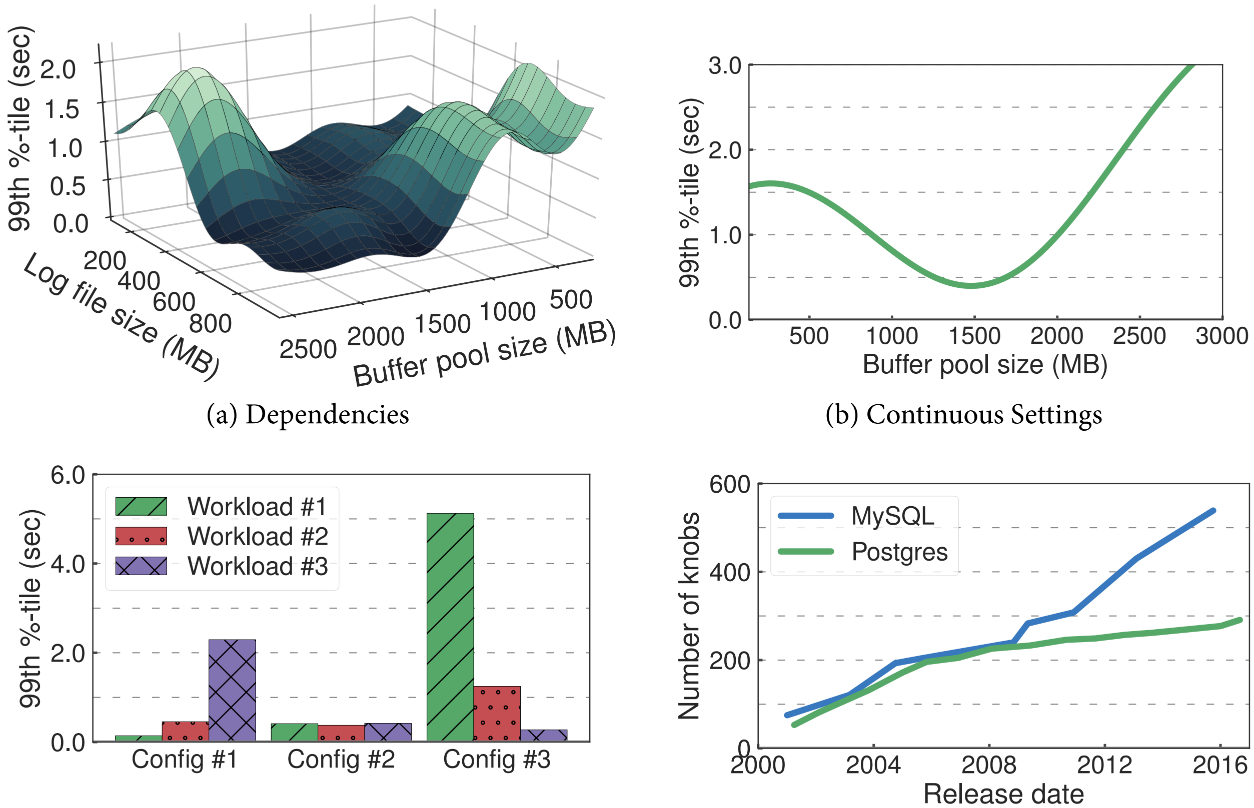
Introduction

The ability to collect, process, and analyze large amounts of data is paramount for being able to extrapolate new knowledge in business and scientific domains [38, 64]. DBMSs are the critical component of data-intensive ("Big Data") applications [89]. The performance of these systems is often measured in metrics such as throughput (e.g., how fast it can collect new data) and latency (e.g., how fast it can respond to a request).

Achieving good performance in DBMSs is non-trivial as they are complex systems with many tunable options that control nearly all aspects of their runtime operation [37]. Such configuration knobs allow the database administrator (DBA) to control various aspects of the DBMS's runtime behavior. For example, they can set how much memory the system allocates for data caching versus the transaction log buffer. Modern DBMSs are notorious for having many configuration knobs [35, 67, 90]. Part of what makes DBMSs so enigmatic is that their performance and scalability are highly dependent on their configurations. Further exacerbating this problem is that the default configurations of these knobs are notoriously bad because the their settings are based on the DBMS's minimum hardware requirements.

There are general rules or "best practice" guidelines available for tuning DBMSs, but these do not always provide good results for a range of applications and hardware configurations. Although one can rely on certain precepts to achieve good performance on a particular DBMS, they are not universal for all applications. Thus, many organizations resort to hiring expensive experts to tune their system. For example, a 2013 survey found that 40% of engagement requests for a large Postgres service company were for DBMS tuning and knob configuration issues [67].

One common approach to tuning a DBMS is for the DBA to copy the database to another machine and manually measure the performance of a sample workload from the real application. Based on the outcome of this test, they will then tweak the DBMS's configuration according to some combination of tuning guidelines and intuition based on past experiences. The DBA then repeats the experiment to see whether the performance improves [90]. Such a "trial-and-error" approach to DBMS tuning is tedious, expensive, and inefficient because (1) many of the knobs are not independent [37], (2) the values for some knobs are continuous, (3) one often cannot reuse the same configuration from one application to the next, and (4) DBMSs are always adding new knobs.



(a) Dependencies

(b) Continuous Settings

(c) Non-Reusable Configurations

(d) Tuning Complexity

Figure 1.1: Motivating Examples – Figures 1.1a to 1.1c show performance measurements for the YCSB workload running on MySQL (v5.6) using different configuration settings. Figure 1.1d shows the number of tunable knobs provided in MySQL and Postgres releases over time.

We now discuss these issues in further detail. To highlight their implications, we ran a series of experiments using MySQL (v5.6) that execute variations of the YCSB workload with different knob settings.

Dependencies: DBMS tuning guides strongly suggest that a DBA only change one knob at a time. This is wise but woefully slow given the large number of knobs. It is also not entirely helpful because changing one knob may affect the benefits of another. But it is difficult enough for humans to understand the impact of one knob let alone the interactions between multiple ones. The different combinations of knob settings means that finding the optimal configuration is NP-hard [92]. To demonstrate this point, we measured the performance of MySQL for different configurations that vary the size of its buffer pool[1] and the size of its log file.[2] The results in Figure 1.1a show that the DBMS achieves better performance when both the buffer pool and log file sizes are large. But in general, the latency is low when the buffer pool size and log file size are "balanced." If the buffer pool is large and the log file size is small, then the DBMS maintains a smaller number of dirty pages

[1]**MySQL Knob:** `INNODB_BUFFER_POOL_SIZE`
[2]**MySQL Knob:** `INNODB_LOG_FILE_SIZE`

and thus has to perform more flushes to disk.

Continuous Settings: Another difficult aspect of DBMS tuning is that there are many possible settings for knobs, and the differences in performance from one setting to the next could be irregular. For example, the size of the DBMS's buffer pool can be an arbitrary value from zero to the amount of DRAM on the system. In some ranges, a 0.1 GB increase in this knob could be inconsequential, while in other ranges, a 0.1 GB increase could cause performance to drop precipitously as the DBMS runs out of physical memory. To illustrate this point, we ran another experiment where we increase MySQL's buffer pool size from 10 MB to 3 GB. The results in Figure 1.1b show that the latency improves continuously up until 1.5 GB, after which the performance degrades because the DBMS runs out of physical memory.

Non-Reusable Configurations: The effort that a DBA spends on tuning one DBMS does not make tuning the next one any easier. This is because the best configuration for one application may not be the best for another. In this experiment, we execute three YCSB workloads using three MySQL knob configurations. Each configuration is designed to provide the best latency for one of the workloads (i.e., config #1 is the best for workload #1, same for #2 and #3). Figure 1.1c shows that the best configuration for each workload is the worst for another. For example, switching from config #1 to config #3 improves MySQL's latency for workload #3 by 90%, but degrades the latency of workload #1 by 3500%. Config #2 provides the best average performance overall. But both workloads #1 and #3 improve by over 2× using their optimized configurations.

Tuning Complexity: Lastly, the number of DBMS knobs is always increasing as new versions and features are released. It is difficult for DBAs to keep up to date with these changes and understand how that will affect their system. The graph in Figure 1.1d shows the number of knobs for different versions of MySQL and Postgres dating back to 2001. This shows that over 15 years the number of knobs increased by 3× for Postgres and by nearly 6× for MySQL.

The above examples show how tricky it is to configure a DBMS. This complexity is a major contributing factor to the high total cost of ownership for database systems. Personnel is estimated to be almost 50% of the total ownership cost of a large-scale DBMS [82], and many DBAs spend nearly 25% of their time on tuning [34]. Given this, there is strong interest in automatic techniques for tuning a DBMS.

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 [2, 5, 32, 62, 110]. These tools, however, are unable to fully optimize a DBMS's knobs. This is partly because they only target 10 – 15 knobs believed by experts 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 [37, 61, 65, 103, 109]. 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 production 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 inefficient and infeasible for deployments with hundreds or thousands of databases.

This dissertation 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 ML 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 dissertation is on DBMS configuration tuning, many of our solutions can be applied to other optimization problems with expensive black-box functions.

We provide 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 summarize the technical contributions of this thesis as follows:

- 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.
- 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.
- A field study and evaluation of the efficacy of three state-of-the-art ML-based tuning tools on a production database. As part of our analysis, we characterize how much of the tuning process is automated in our experiments. We also present several optimizations that we developed for OtterTune and the ML algorithms that we evaluated, which were needed to support this study.
- An investigation of less-obtrusive tuning strategies that recommend knob configurations using information observed passively from the target database without actually tuning it. We present two methods to exploit the similarity between database workloads.

Chapter 2

OtterTune Tuning Service

In this chapter, we present OtterTune, a DBMS configuration tuning service that can automatically optimize a database's configuration knobs for an application's workload. It maintains a repository of data collected from previous tuning sessions, and uses this data to build models of how the DBMS responds to different knob configurations. For a new application, 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.

Before discussing the details of our system, we first provide some background on the DBMS's configuration knobs and runtime metrics. We then discuss how OtterTune's controller collects knobs and metrics from the DBMS. Finally, we describe the steps OtterTune's procedure automatically tunes a DBMS's configuration and discuss its limitations.

2.1 Runtime Metrics

A DBMS's metrics are counters that record the activities of its internal runtime components. Engineers add these metrics to enable DBAs and monitoring tools to observe the system's behavior and diagnose performance problems. The DBMSs also use them internally to trigger maintenance operations, such as garbage collection in MVCC systems and compaction in LSM systems. A DBMS configuration tuning service uses the runtime metrics to learn the cost/benefit of the knobs under varying conditions and make informed recommendations.

There are three categories of runtime metrics: (1) *accumulating*, (2) *aggregation*, and (3) *status*. The first are *accumulating* metrics that count the number of events that have occurred since some point of time. For example, the DBMS can record the number of pages read from disk since it started. The second category are *aggregation* metrics that record the average number of events over a time window. An example of this category is the average time to acquire a row lock for tables. Metrics in the third category record the current state of an activity or option. Examples of status metrics include the last time the garbage collector ran and the SSL protocol version.

The metrics are not standardized; both the metrics available and the names of the metrics can vary from one DBMS vendor to the next. All DBMS vendors expose the metrics they record in some form through either a standardized ANSI "information schema" interface or from its system catalog, or both. Several vendors also provide programmatic access to meta-data about the metrics, such as the type of data (e.g., integer or string). A DBMS configuration tuning service can use this meta-data to process the metric data during a tuning session.

Some systems, like MySQL, only report "global" statistics for the entire DBMS. Other systems, however, provide separate statistics for tables or databases. Commercial DBMSs even maintain separate statistics for individual components (e.g., IBM DB2 tracks statistics per buffer pool instance).

Since collecting the metrics adds some overhead to query execution, most DBMSs provide knobs to disable the collection of metrics or set it to a minimum level. A number of them also support more fine-grained control, such as setting how frequently to collect metrics or even disable individual metrics.

2.2 Configuration Knobs

A DBMS's configuration knobs control aspects of its runtime operations. A DBMS tuning service optimizes these knobs for an application's workload. The three categories of knobs are (1) *resources*, (2) *policies*, and (3) *locations*. Knobs in the first category specify how much of a resource the system uses for a task. These can be either for fixed components (e.g., the number of garbage collection threads) or for dynamic activities (e.g., the amount of memory to use per query). Policy configuration knobs control how the DBMS behaves for specific tasks. For example, a knob can control whether the DBMS flushes the write-ahead log to disk when a transaction commits. Lastly, the location knobs specify where the DBMS finds resources that it needs (e.g., file paths) and how it interacts with the outside world (e.g., network port number).

Like runtime metrics, a DBMS's configuration knobs are not standardized across DBMS vendors. Vendors also expose the knobs and any knob meta-data they provide in the same way as the metrics (i.e., through a common interface or the system catalog). To change a DBMS's configuration knobs, vendors support the `SET` command that is part of the SQL standard. They also provide configuration files for setting knobs; however, the format and other details may differ between vendors. Vendors frequently a provide convenient syntax for setting them or the ability to set them from the command line when starting the server.

2.3 System Overview

OtterTune has a flexible and extensible architecture that facilitates the addition of other DBMSs and tuning algorithms. Figure 2.1 shows an overview of OtterTune's architecture. The system is comprised of two parts. The first is the client-side *controller* that interacts with the target DBMS to be tuned. It collects runtime metrics and configuration knobs from the DBMS using a standard API (e.g., JDBC) and installs new configurations.

The second part is OtterTune's *tuning manager*. It receives the information collected from the controller and stores it in its repository with data from previous tuning sessions. This repository does not contain any confidential information about the DBMSs or their databases; it only contains knob configurations and performance data. OtterTune organizes this data per major DBMS version (e.g., Postgres v10 data is separate from Postgres v11). This prevents OtterTune from tuning knobs

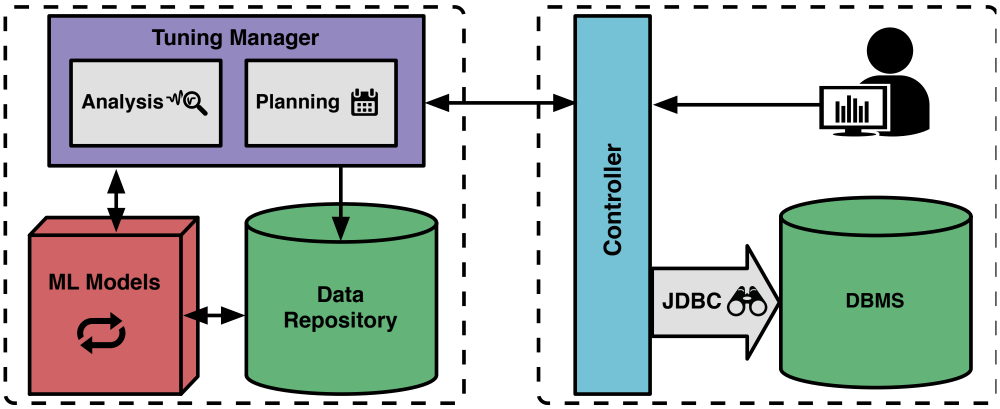


Figure 2.1: OtterTune Architecture – An overview of the components in the OtterTune system. The controller connects to the DBMS and collects information about the performance of the system. This information is then sent to the tuning manager where it is stored in its repository. It then builds models that are used to select an optimal configuration for the DBMS.

from older versions of the DBMS that may be deprecated in newer versions, or tuning knobs that only exist in newer versions. The manager is also supported by background processes that continuously analyze new data and refine OtterTune's internal ML models. These models allow it to identify the relevant knobs and metrics without human input, and find workloads in the repository that are similar to the target.

2.4 Configuration Tuning Procedure

At the start of a new tuning session, the DBA tells OtterTune what metric to optimize when selecting a configuration (e.g., latency, throughput). The OtterTune controller then connects to the target DBMS and collects its hardware profile and current knob configuration. We assume that this hardware profile is a single identifier from a list of pre-defined types (e.g., an instance type on Amazon EC2).

The controller then starts the first *observation period*. This is some amount of time where the controller will observe the DBMS and measure DBMS-independent external metrics chosen by the DBA (e.g., latency). The DBA may choose to execute either a set of queries for a fixed time period or a specific workload trace. If the DBA chooses the first option, then the length of the observation period is equal to the fixed time period. Otherwise, the duration depends on how long it takes for the DBMS to replay the workload trace. Fixed observation periods are well-suited for the fast, simple queries that are characteristic of OLTP workloads, whereas variable-length periods are often necessary for executing the long-running, complex queries present in OLAP workloads.

At the end of the observation period, the controller then collects DBMS's configuration knobs and runtime metrics. The controller reports the "ground truth" knob values from the DBMS instead of the recommended values not only because it is a good policy for preventing bugs but also because there can be differences due to the DBMS's policies for setting them. For example, Postgres uses an internal unit of 8kB for the knob that controls the size of its buffer cache and will automatically adjust the value to an 8kB boundary when it is set.

When OtterTune's tuning manager receives the result of a new observation period from the controller, it first stores that information in its repository. From this, OtterTune then computes the next configuration that the controller should install on the target DBMS. To assist the DBA with deciding whether to terminate the tuning session, OtterTune provides the controller with an estimate of how much better the recommended configuration is compared to the best configuration that it has seen so far. This process continues until the DBA is satisfied with the improvements over the initial configuration.

2.5 Statistics Collection

OtterTune provides an extensible plug-in interface that allows its controller to easily retrieve the knobs and metrics for a DBMS using the appropriate API and then re-factor them into a universal format. The service provides built-in support for retrieving these metrics from the DBMS itself via its query API and the OLTP-Bench framework [36]. Supporting other benchmarking frameworks or third-party monitoring services (e.g., Prometheus, Druid) requires only minor changes to OtterTune's controller.

At the beginning of each observation period, the controller first resets all of the metrics for the target DBMS. It then retrieves the new metric data at the end of the period. Since at this point, OtterTune does not know which metrics are useful, it collects every metric that the DBMS makes available and stores it as a key/value pair in its repository.

A key consideration in this collection process is how to represent metrics for sub-elements of the DBMS and database (see Section 2.1). The problem with this fine-grained data is that the DBMS provides multiple metrics with the same name. One potential solution is to prefix the name of the sub-element to the metric's name. For example, Postgres' metric for the number of blocks read for the table "foo" would be stored in the repository as `foo.heap_blks_read`. But this approach means that it is unable to map this metric to other databases since they will have different names for their tables. OtterTune instead stores the metrics with the same name as a single sum scalar value. This works because OtterTune currently only considers global knobs. We defer the problem of tuning table- or component-specific knobs as future work.

2.6 Assumptions & Limitations

There are several aspects of OtterTune's capabilities that we must address. Foremost is that we assume that the controller has administrative privileges to modify the DBMS's configuration (including restarting the DBMS if necessary). If this is not possible, then the DBA can deploy a second copy of the database on separate hardware for OtterTune's tuning trials. This requires the DBA either to replay a workload trace or to forward queries from the production DBMS. This is the same approach used in previous tools [37].

Restarting the DBMS is often necessary because some knobs only take effect after the system is stopped and started. Some knobs also cause the DBMS to perform extra processing when it comes back on-line (e.g., resizing log files), which can potentially take several minutes depending on the database and the hardware. OtterTune currently ignores the cost of restarting the DBMS in its recommendations. We defer the problem of automatically identifying these knobs and taking the cost of restarting into consideration when choosing configurations as future work.

Because restarting the DBMS is undesirable, many DBMSs support changing some knobs dynamically without having to restart the system. OtterTune stores a list of the dynamic knobs that are available on each of the DBMS versions that it supports, as well as the instructions on how to update them. It then restarts the DBMS only when the set of knobs being tuned requires it. The DBA can also elect to tune only dynamic knobs at the start of the tuning session. This is another alternative that is available to the DBA when restarting the DBMS is prohibited. We maintain a curated black-list of knobs for each DBMS version that is supported by OtterTune. The DBA is provided with this black-list of knobs at the start of each tuning session. The DBA is permitted to add to this list any other knobs that they want OtterTune to avoid tuning. Such knobs could either be ones that do not make sense to tune (e.g., path names of where the DBMS stores files), or ones that could have hidden or serious consequences (e.g., potentially causing the DBMS to lose data). Again, automatically determining whether changing a knob will cause the application to potentially lose data is beyond the scope of our work here.

Lastly, we also assume that the physical design of the database is reasonable. That means that the DBA has already installed the proper indexes, materialized views, and other database elements. There has been a considerable amount of research into automatic database design [27] that the DBA can utilize for this purpose. Investigating how to apply these same techniques to tune the database's physical design is beyond the scope of this thesis and we leave it as future work (see Chapter 7).

Chapter 3

Tuning via Gaussian Process Regression

In this chapter, we present a tuning algorithm based on Gaussian Process Regression that reuses training data gathered from previous sessions to tune new DBMS deployments. To do this, we extended the OtterTune tuning service to support a multi-stage ML pipeline. This ML pipeline enables OtterTune to train 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. An overview of this process is shown in Figure 3.1.

We begin with a discussion of our technique for pruning DBMS metrics in Section 3.1. We then discuss our method for identifying the knobs that have the most impact in Section 3.2, followed by a description of our technique for recommending settings in Section 3.3. In Section 3.4, we present our experimental evaluation.

3.1 Workload Characterization

The first step in the tuning system is to discover a model that best represents the distinguishing aspects of the target workload so that it can identify which previously seen workloads in the repository are similar to it. This enables OtterTune to leverage the information that it has collected from previous tuning sessions to help guide the search for a good knob configuration for the new application.

We might consider two approaches to do this. The first is to analyze the target workload at the logical level. This means examining the queries and the database schema to compute metrics, such as the number of tables/columns accessed per query and the read/write ratio of transactions. These metrics could be further refined using the DBMS's "what-if" optimizer API to estimate additional runtime information [26], like which indexes are accessed the most often. The problem with this approach, however, is that it is impossible to determine the impact of changing a particular knob because all of these estimates are based on the logical database and not the actual runtime behavior of queries. Furthermore, how the DBMS executes a query and how the query relates to internal

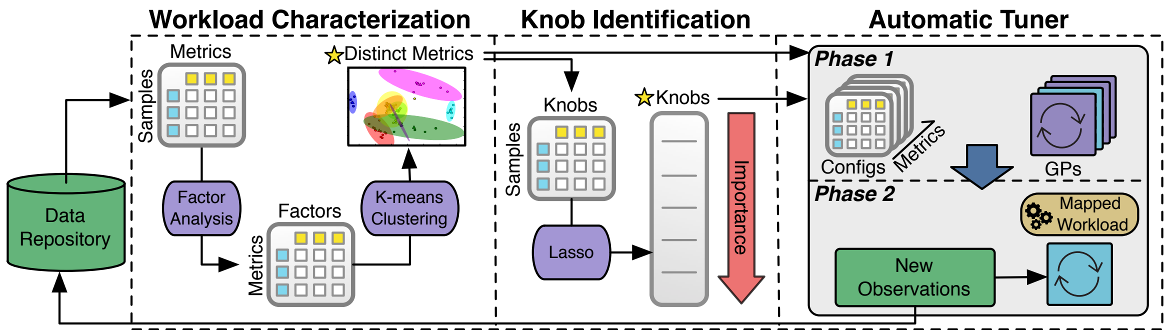


Figure 3.1: OtterTune Machine Learning Pipeline – This diagram shows the processing path of data in OtterTune. All previous observations reside in its repository. This data is first then passed into the **Workload Characterization** (Section 3.1) component that identifies the most distinguishing DBMS metrics. Next, the **Knob Identification** (Section 3.2) component generates a ranked list of the most important knobs. All of this information then fed into the **Automatic Tuner** (Section 3.3) component where it maps the target DBMS's workload to a previously seen workload and generates better configurations.

components that are affected by tuning knobs is dependent on many factors of the database (e.g., size, cardinalities, working set size). Hence, this information cannot be captured just by examining the workload.

A better approach is to use the DBMS's internal runtime metrics to characterize how a workload behaves. All modern DBMSs expose a large amount of information about the system. For example, MySQL's InnoDB engine provides statistics on the number of pages read/written, query cache utilization, and locking overhead. OtterTune characterizes a workload using the runtime statistics recorded while executing it. These metrics provide a more accurate representation of a workload because they capture more aspects of its runtime behavior. Another advantage of them is that they are directly affected by the knobs' settings. For example, if the knob that controls the amount of memory that the DBMS allocates to its buffer pool is too low, then these metrics would indicate an increase in the number of buffer pool cache misses. All DBMSs provide similar information, just with different names and different granularities. But as we will show, OtterTune's model construction algorithms do not require metrics to be labeled.

We now discuss how OtterTune's prunes redundant metrics to reduce the complexity of our recommendation problem.

3.1.1 Pruning Redundant Metrics

The next step is to automatically remove the superfluous metrics. It is important to remove such elements so that OtterTune only has to consider the smallest set of metrics that capture the variability in performance and distinguishing characteristics for different workloads. Reducing the size of this set reduces the search space of ML algorithms, which in turn speeds up the entire process and increases the likelihood that the models will fit in memory on OtterTune's tuning manager. We will show in subsequent sections that the metrics available to OtterTune are sufficient to distinguish between workloads for DBMSs deployed on the same hardware.

Redundant DBMS metrics occur for two reasons. The first are ones that provide different granularities for the exact same metric in the system. For example, MySQL reports the amount of data read in terms of bytes[1] and pages.[2] The two metrics are the same measurement just in different units, thus it is unnecessary to consider both of them. The other type of redundant metrics are ones that represent independent components of the DBMS but whose values are strongly correlated. For example, we found from our experiments that the Postgres metric for the number of tuples updated[3] moves almost in unison with the metric that measures the number of blocks read from the buffer for indexes.[4]

We use two well-studied techniques for this pruning. The first is a dimensionality reduction technique, called *factor analysis* (FA) [7], that transforms the (potentially) high dimensional DBMS metric data into lower dimensional data. We then use the second technique, called *k-means* [8], to cluster this lower dimensional data into meaningful groups. Using a dimensionality reduction technique is a preprocessing step for many clustering algorithms because they reduce the amount of "noise" in the data [55, 56]. This improves the robustness and the quality of the cluster analysis.

Given a set of real-valued variables that contain arbitrary correlations, FA reduces these variables to a smaller set of *factors* that capture the correlation patterns of the original variables. Each factor is a linear combination of the original variables; the factor coefficients are similar to and can be interpreted in the same way as the coefficients in a linear regression. Furthermore, each factor has unit variance and is uncorrelated with all other factors. This means that one can order the factors by how much of the variability in the original data they explain. We found that only the initial factors are significant for our DBMS metric data, which means that most of the variability is captured by the first few factors.

The FA algorithm takes as input a matrix X whose rows correspond to metrics and whose columns correspond to knob configurations that we have tried. The entry X_{ij} is the value of metric i on configuration j. FA gives us a smaller matrix U: the rows of U correspond to metrics, while the columns correspond to factors, and the entry U_{ij} is the coefficient of metric i in factor j. We can scatter-plot the metrics using elements of the ith row of U as coordinates for metric i. Metrics i and j will be close together if they have similar coefficients in U — that is, if they tend to correlate strongly in X. Removing redundant metrics now means removing metrics that are too close to one another in our scatter-plot.

We then cluster the metrics via k-means, using each metric's row of U as its coordinates. We keep a single metric for each cluster, namely, the one closest to the cluster center. One of the drawbacks of using k-means is that it requires the optimal number of clusters (K) as its input. We use a simple heuristic [78] to fully automate this selection process and approximate K. Although this approach is not guaranteed to find the optimal solution, it does not require a human to manually interpret a graphical representation of the problem to determine the optimal number of clusters. We

[1] **MySQL Metric:** `INNODB_DATA_READ`
[2] **MySQL Metric:** `INNODB_PAGES_READ`
[3] **Postgres Metric:** `PG_STAT_DATABASE.TUP_UPDATED`
[4] **Postgres Metric:** `PG_STATIO_USER_TABLES.IDX_BLKS_HIT`

compared this heuristic with other techniques [91, 99] for choosing K and found that they select values that differ by one to two clusters at most from our approximations. Such variations made little difference in the quality of configurations that OtterTune generated in our experimental evaluation in Section 3.4.

The visualization in Figure 3.2 shows a two-dimensional projection of the scatter-plot and the metric clusters in MySQL and Postgres. In the MySQL clusters in Figure 3.2a, OtterTune identifies a total of nine clusters. These clusters correspond to distinct aspects of a DBMS's performance. For example, in the case of MySQL, the metrics that measure the amount of data written[5], the amount of data read[6], and the time spent waiting for resources[7] are all grouped into the same cluster. In Figure 3.2b we see that OtterTune selects eight clusters for Postgres' metrics. Like MySQL, the metrics in each cluster correspond to similar measurements. But in Postgres the metrics are clustered on specific components in the system, like the background writer[8,9] and indexes.[10,11]

An interesting finding with this clustering is that OtterTune tends to group together useless metrics (e.g., SSL connection data). It does not, however, have a programmatic way to determine that they are truly useless and thus it has to include them in further computations. We could provide the system with a hint of one or more of these metrics and then discard the cluster that it gets mapped to.

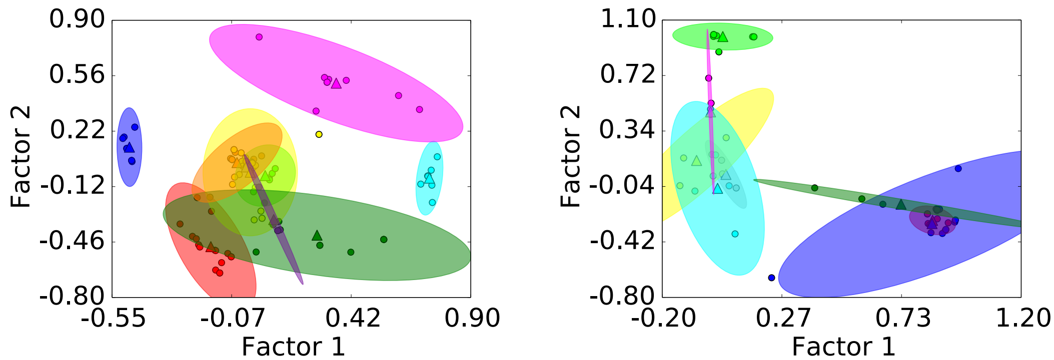
From the original set of 131 metrics for MySQL and 57 metrics for Postgres, we are able to reduce the number of metrics by 93% and 82%, respectively. Note that OtterTune still collects and stores data for all of the DBMS's metrics in its repository even if they are marked as redundant. The set of metrics that remain after pruning the FA reduction is only considered for the additional ML components that we discuss in the next sections.

3.2 Identifying Important Knobs

After pruning the redundant metrics, OtterTune next identifies which knobs have the strongest impact on the DBA's target objective function. DBMSs can have hundreds of knobs, but only a subset actually affect the DBMS's performance. Thus, reducing the number of knobs limits the total number of possible DBMS configurations that must be considered. We want to discover both negative and positive correlations. For example, reducing the amount of memory allocated for the DBMS's buffer pool is likely to degrade the system's overall latency, and we want to discover this strong (albeit negative) influence on the DBMS's performance.

OtterTune uses a popular feature selection technique for linear regression, called *Lasso* [98], to expose the knobs that have the strongest correlation to the system's overall performance. In order to detect nonlinear correlations and dependencies between knobs, we also include polynomial features in our regression.

[5] **MySQL Metrics:** `INNODB_DATA_WRITTEN`, `INNODB_BUFFER_POOL_WRITE_REQUESTS`
[6] **MySQL Metrics:** `INNODB_ROWS_READ`, `BYTES_SENT`
[7] **MySQL Metrics:** `INNODB_LOG_WAITS`, `INNODB_ROW_LOCK_TIME_MAX`
[8] **Postgres Metric:** `PG_STAT_BGWRITER.BUFFERS_CLEAN`
[9] **Postgres Metric:** `PG_STAT_BGWRITER.MAXWRITTEN_CLEAN`
[10] **Postgres Metric:** `PG_STATIO_USER_TABLES.IDX_BLKS_READ`
[11] **Postgres Metric:** `PG_STATIO_USER_INDEXES.IDX_BLKS_READ`



(a) MySQL (v5.6) (b) Postgres (v9.3)

Figure 3.2: Metric Clustering – Grouping DBMS metrics using k-means based on how similar they are to each other as identified by Factor Analysis and plotted by their (`f1`, `f2`) coordinates. The color of each metric shows its cluster membership. The triangles represent the cluster centers.

OtterTune's tuning manager performs these computations continuously in the background as new data arrives from different tuning sessions. In our experiments, each invocation of Lasso takes $\sim$20 min and consumes $\sim$10 GB of memory for a repository comprised of 100k trials with millions of data points. The dependencies and correlations that we discover are then used in OtterTune's recommendation algorithms, presented in Section 3.3.

We now describe how to use Lasso to identify important knobs and the dependencies that may exist between them. Then we discuss how OtterTune uses this during the tuning process.

3.2.1 Feature Selection with Lasso

Linear regression is a statistical method used to determine the strength of the relationship between one or more dependent variables (y) and each of the independent variables (X). These relationships are modeled using a linear predictor function whose weights (i.e., coefficients) are estimated from the data.

The most common method of fitting a linear regression model is *ordinary least squares* (OLS), which estimates the regression weights by minimizing the residual squared error. Such a model allows one to perform statistical tests on the weights to assess the significance of the effect of each independent variable [23]. Although OtterTune could use these measurements to determine the knob ordering, OLS suffers from two shortcomings that make it an unsatisfactory solution in high(er) dimensional settings. First, the estimates have low bias but high variance, and the variance continues to increase as more features are included in the model. The latter issue degrades the prediction and variable selection accuracy of the model. Second, the estimates become harder to interpret as the number of features increases, since extraneous features are never removed (i.e., OLS does not perform feature selection).

To avoid these problems, OtterTune employs a regularized version of least squares, known as Lasso, that reduces the effect of irrelevant variables in linear regression models by penalizing models with large weights. The major advantage of Lasso over other regularization and feature selection methods is that it is interpretable, stable, and computationally efficient [39, 98]. There is also both practical and theoretical work backing its effectiveness as a consistent feature selection algorithm [12, 100, 101, 108].

Lasso works by adding an L_1 penalty that is equal to a constant λ times the sum of absolute weights to the loss function. Because each non-zero weight contributes to the penalty term, Lasso effectively shrinks some weights and forces others to zero. That is, Lasso performs feature selection by automatically selecting more relevant features (i.e., those with non-zero weights), and discarding the others (i.e., those with zero weights). The number of features that it keeps depends on the strength of its penalty, which is controlled by adjusting the value of λ. Lasso improves the prediction accuracy and interpretability of the OLS estimates via its shrinkage and selection properties: shrinking small weights towards zero reduces the variance and creates a more stable model, and deselecting extraneous features generates models that are easier to interpret.

As in the usual regression scenario, OtterTune constructs a set of independent variables (X) and one or more dependent variables (y) from the data in its repository. The independent variables are the DBMS's knobs (or functions of these knobs) and the dependent variables are the metrics that OtterTune collects during an observation period from the DBMS. OtterTune uses the *Lasso path algorithm* [50] to determine the order of importance of the DBMS's knobs. The algorithm starts with a high penalty setting where all weights are zero and thus no features are selected in the regression model. It then decreases the penalty in small increments, recomputes the regression, and tracks what features are added back to the model at each step. OtterTune uses the order in which the knobs first appear in the regression to determine how much of an impact they have on the target metric (e.g., the first knob selected is the most important). We provide more details and visualizations of this process in Appendix A.1.

Before OtterTune computes this model, it executes two preprocessing steps to normalize the knobs data. This is necessary because Lasso provides higher quality results when the features are (1) continuous, (2) have approximately the same order of magnitude, and (3) have similar variances. It first transforms all of the categorical features to "dummy" variables that take on the values of zero or one. Specifically, each categorical feature with n possible values is converted into n binary features. Although this encoding method increases the number of features, all of the DBMSs that we examined have a small enough number of categorical features that the performance degradation was not noticeable. Next, OtterTune scales the data. We found that standardizing the data (i.e., subtracting the mean and dividing by the standard deviation) provides adequate results and is easy to execute. We evaluated more complicated approaches, such as computing deciles, but they produced nearly identical results as the standardized form.

3.2.2 Dependencies

As we showed in Chapter 2, many of a DBMS's knobs are non-independent. This means that changing one may affect another. It is important that OtterTune takes these relationships into consideration when recommending a configuration to avoid nonsensical settings. For example, if the system does not "know" that it should not try to allocate the entire system memory to multiple purposes controlled by different knobs, then it could choose a configuration that would cause the

DBMS to become unresponsive due to thrashing. In other cases, we have observed that a DBMS will refuse to start when the requested configuration uses too much memory.

Within the feature selection method described above, we can capture such dependencies between knobs by including polynomial features in the regression. The regression and feature selection methods do not change: they just operate on polynomial features of the knobs instead of the raw knobs themselves. For example, to test whether the buffer pool memory allocation knob interacts with the log buffer size knob, we can include a feature which is the product of these knobs' values: if Lasso selects this product feature, we have discovered a dependence between knobs.

3.2.3 Incremental Knob Selection

OtterTune now has a ranked list of all knobs. The Lasso path algorithm guarantees that the knobs in this list are ordered by the strength of statistical evidence that they are relevant. Given this, OtterTune must decide how many of these knobs to use in its recommendations. Using too many of them increases OtterTune's optimization time significantly because the size of the configuration space grows exponentially with the number of knobs. But using too few of them would prevent OtterTune from finding the best configuration. The right number of knobs to consider depends on both the DBMS and the target workload.

To automate this process, we use an incremental approach where OtterTune dynamically increases the number of knobs used in a tuning session over time. Expanding the scope gradually in this manner has been shown to be effective in other optimization algorithms [33, 43]. As we show in our evaluation in Section 3.4.3, this always produces better configurations than any static knob count.

3.3 Automated Tuning

Now at this point OtterTune has (1) the set of non-redundant metrics, (2) the set of most impactful configuration knobs, and (3) the data from previous tuning sessions stored in its repository.

OtterTune repeatedly analyzes the data it has collected so far in the session and then recommends the next configuration to try. It executes a two-step analysis after the completion of each observation period in the tuning process. In the first step, the system identifies which workload from a previous tuning session is most emblematic of the target workload. It does this by comparing the session's metrics with those from the previously seen workloads to see which ones react similarly to different knob settings. Once OtterTune has matched the target workload to the most similar one in its repository, it then starts the second step of the analysis where it chooses a configuration that is explicitly selected to maximize the target objective. We now describe these steps in further detail.

3.3.1 Step #1 – Workload Mapping

The goal of this first step is to match the target DBMS's workload with the most similar workload in its repository based on the performance measurements for the selected group of metrics. We find that the matched workload varies for the first few experiments before converging to a single workload. This suggests that the quality of the match made by OtterTune increases with the amount of data gathered from the target workload, which is what we would expect. For this reason, using a *dynamic mapping* scheme is preferable to *static mapping* (i.e., mapping one time after the end of the first observation period) because it enables OtterTune to make more educated matches as the tuning session progresses.

For each DBMS version, we build a set S of N matrices — one for every non-redundant metric — from the data in our repository. Similar to the Lasso and FA models, these matrices are constructed by background processes running on OtterTune's tuning manager (see Chapter 2). The matrices in S (i.e., $X_0, X_1, \ldots X_{N-1}$) have identical row and column labels. Each row in matrix X_m corresponds to a workload in our repository and each column corresponds to a DBMS configuration from the set of all unique DBMS configurations that have been used to run any of the workloads. The entry $X_{m,i,j}$ is the value of metric m observed when executing workload i with configuration j. If we have multiple observations from running workload i with configuration j, then entry $X_{m,i,j}$ is the median of all observed values of metric m.

The workload mapping computations are straightforward. OtterTune calculates the Euclidean distance between the vector of measurements for the target workload and the corresponding vector for each workload i in the matrix X_m (i.e., $X_{m,i,:}$). It then repeats this computation for each metric m. In the final step, OtterTune computes a "score" for each workload i by taking the average of these distances over all metrics m. The algorithm then chooses the workload with the lowest score as the one that is most similar to the target workload for that observation period.

Before computing the score, it is critical that all metrics are of the same order of magnitude. Otherwise, the resulting score would be unfair since any metrics much larger in scale would dominate the average distance calculation. OtterTune ensures that all metrics are the same order of magnitude by computing the deciles for each metric and then binning the values based on which decile they fall into. We then replace every entry in the matrix with its corresponding bin number. With this extra step, we can calculate an accurate and consistent score for each of the workloads in OtterTune's repository.

3.3.2 Step #2 – Configuration Recommendation

In the next step, OtterTune uses *Gaussian Process* (GP) regression [81] to recommend configurations that it believes will improve the target metric. GP regression is a state-of-the-art technique with power approximately equal to that of deep networks. There are a number of attractive features of GPs that make it an appropriate choice for modeling the configuration space and making recommendations. Foremost is that GPs provide a theoretically justified way to trade off exploration (i.e., acquiring new knowledge) and exploitation (i.e., making decisions based on existing knowledge) [58, 87]. Another reason is that GPs, by default, provide confidence intervals. Although methods like bootstrapping can be used to obtain confidence intervals for deep networks and other models that do not give them, they are computationally expensive and thus not feasible (yet) for an on-line tuning service.

OtterTune starts the recommendation step by reusing the data from the workload that it selected previously to train a GP model. It updates the model by adding in the metrics from the target workload that it has observed so far. But since the mapped workload is not exactly identical to the unknown one, the system does not fully trust the model's predictions. We handle this by increasing the variance of the noise parameter for all points in the GP model that OtterTune has not tried yet for this tuning session. That is, we add a ridge term to the covariance. We also add a smaller ridge term for each configuration that OtterTune selects. This is helpful for "noisy" virtualized environments where the external DBMS metrics (i.e., throughput and latency) vary from one observation period to the next.

Now for each observation period in this step, OtterTune tries to find a better configuration than the best configuration that it has seen thus far in this session. It does this by either (1) searching an unknown region in its GP (i.e., workloads for which it has little to no data for), or (2) selecting a configuration that is near the best configuration in its GP. The former strategy is referred to as *exploration*. This helps OtterTune look for configurations where knobs are set to values that are beyond the minimum or maximum values that it has tried in the past. This is useful for trying certain knobs where the upper limit might depend on the underlying hardware (e.g., the amount of memory available). The second strategy is known as *exploitation*. This is where OtterTune has found a good configuration and it tries slight modifications to the knobs to see whether it can further improve the performance.

Which of these two strategies OtterTune chooses when selecting the next configuration depends on the variance of the data points in its GP model. It always chooses the configuration with the greatest expected improvement. The intuition behind this approach is that each time OtterTune tries a configuration, it "trusts" the result from that configuration and similar configurations more, and the variance for those data points in its GP decreases. The expected improvement is near-zero at sampled points and increases in between them (although possibly by a small amount). Thus, it will always try a configuration that it believes is optimal or one that it knows little about. Over time, the expected improvement in the GP model's predictions drops as the number of unknown regions decreases. This means that it will explore the area around good configurations in its solution space to optimize them even further.

OtterTune uses *gradient descent* [50] to find the local optimum on the surface predicted by the GP model using a set of configurations, called the initialization set, as starting points. There are two types of configurations in the initialization set: the first are the top-performing configurations that have been completed in the current tuning session, and the second are configurations for which the value of each knob is chosen at random from within the range of valid values for that knob. Specifically, the ratio of top-performing configurations to random configurations is 1-to-10. During each iteration of gradient descent, the optimizer takes a "step" in the direction of the local optimum until it converges or has reached the limit on the maximum number of steps it can take. OtterTune selects from the set of optimized configurations the one that maximizes the potential improvement to run next. This search process is quick; in our experiments OtterTune's tuning manager takes 10–20 sec to complete its gradient descent search per observation period. Longer searches did not yield better results.

Similar to the other regression-based models that we use in OtterTune (see Sections 3.2.1 and 3.3.1), we employ preprocessing to ensure that features are continuous and of approximately the same scale and range. We encode categorical features with dummy variables and standardize all data before passing it as input to the GP model.

Once OtterTune selects the next configuration, it returns this along with the expected improvement from running this configuration to the client. The DBA can use the expected improvement calculation to decide whether they are satisfied with the best configuration that OtterTune has generated thus far.

3.4 Experimental 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 three different DBMSs in our evaluation: MySQL (v5.6), Postgres (v9.3), and Actian Vector (v4.2). MySQL and Postgres were installed using the OS's package manager. Vector was installed from packages provided on its website. 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.large` 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.

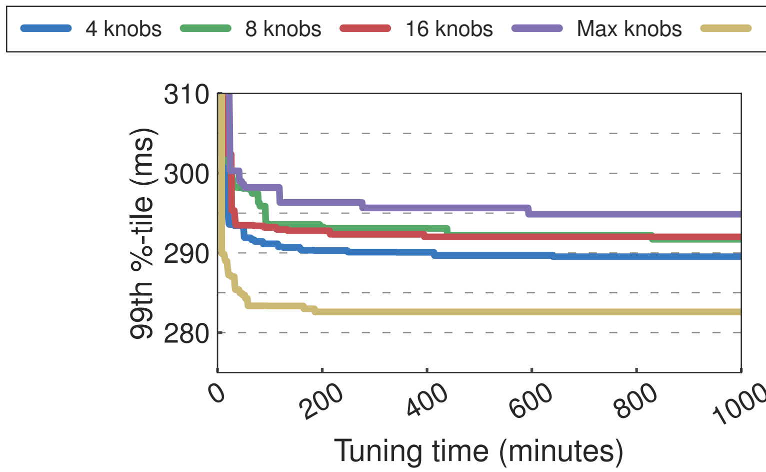
3.4.1 Workloads

For these experiments, we use workloads from the OLTP-Bench testbed that differ in complexity and system demands [3, 36]:

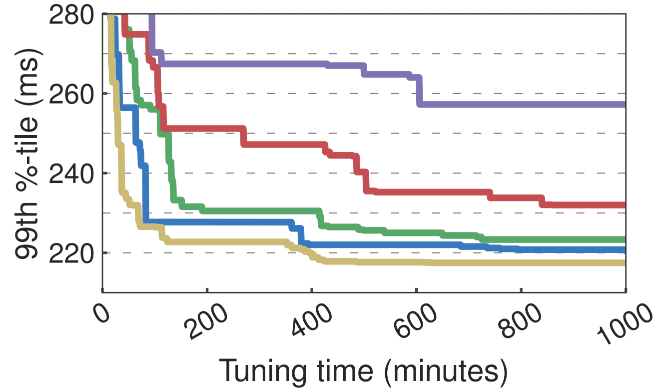
YCSB: The Yahoo! Cloud Serving Benchmark (YCSB) [30] is modeled after data management applications with simple workloads and high scalability requirements. It is comprised of six OLTP transaction types that access random tuples based on a Zipfian distribution. The database contains a single table with 10 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 [95]. 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.

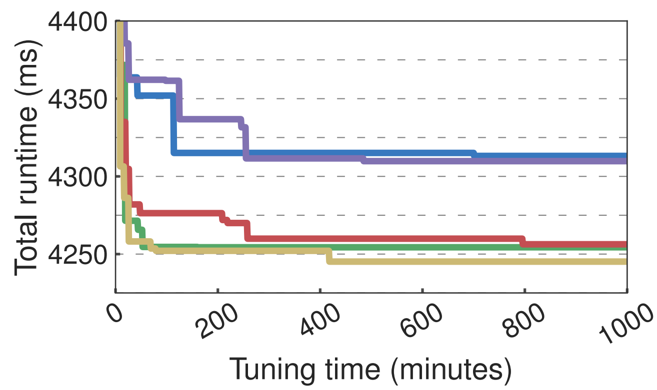
Wikipedia: This OLTP benchmark is derived from the software that runs the popular on-line encyclopedia. The database contains 11 tables and eight different transaction types. These transactions correspond to the most common operations in Wikipedia for article and "watchlist" management.



(a) MySQL (TPC-C)



(b) Postgres (TPC-C)



(c) Vector (TPC-H)

Figure 3.3: Number of Knobs – The performance of the DBMSs for TPC-C and TPC-H during the tuning session using different configurations generated by OtterTune that only configure a certain number of knobs.

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.

TPC-H: This is a decision support system workload that simulates an OLAP environment where there is little prior knowledge of the queries [96]. It contains eight tables in 3NF schema and 22 queries with varying complexity. We use a scale factor of 10 in each experiment (∼10 GB).

For the OLTP workloads, 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. For the OLAP workloads, OtterTune uses a variable-length observation period that is the total execution time of the target workload for that period. The workload's total execution time is the target metric for the OLAP experiments.

3.4.2 Training Data Collection

As discussed in Chapter 2, 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 used permutations of YCSB and TPC-H.

We created 15 variations of YCSB with different workload mixtures. For TPC-H, we divided the queries into four groups that are each emblematic of the overall workload [17]. All of the training data was collected using the DBMSs' default isolation level.

We also needed to evaluate different knob configurations. For each workload, we performed a parameter sweep across the knobs using random values. In some cases, we had to manually override the valid ranges of these knobs because the DBMS would refuse to start if any of the knob settings exceeded the physical capacity of any of the machine's resources (e.g., if the size of the buffer pool was set to be larger than the amount of RAM). This would not be a problem in a real deployment scenario because if the DBMS does not start then OtterTune is not able to collect the data.

We executed a total of over 30k trials per DBMS using these different workload and knob configurations. Each of these trials is treated like 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 initial setting after loading our training data. This is to avoid tainting our measurements with additional knowledge gained from tuning the previous experiments. For the OLAP experiments, we also ensure that OtterTune's ML models are not trained with data from the same TPC-H workload mixture as the target workload.

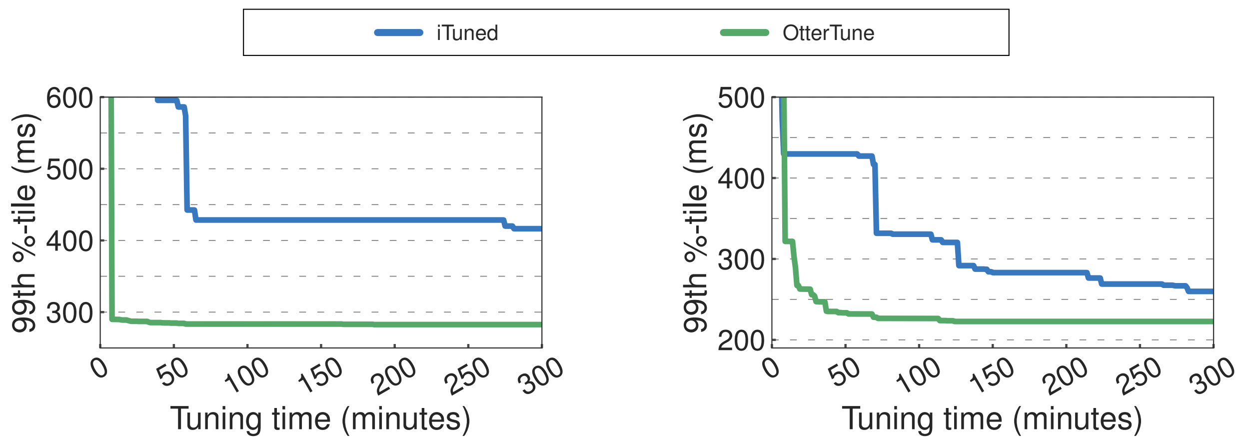
3.4.3 Number of Knobs

We begin with an analysis of OtterTune's performance when optimizing different numbers of knobs during each observation period. The goal of this experiment is to show that OtterTune can properly identify the optimal number of knobs for tuning each DBMS. Although using more knobs may allow OtterTune to find a better configuration, it also increases the computational overhead, data requirements, and memory footprint of its algorithms.

We use the TPC-C benchmark for the OLTP DBMSs (MySQL and Postgres) and TPC-H for the OLAP DBMS (Vector). We evaluate two types of knob count settings. The first is a *fixed* count where OtterTune considers the same set of knobs throughout the entire tuning session. The second is our *incremental* approach from Section 3.2.3 where OtterTune increases the number the knobs it tunes gradually over time. For this setting, the tuning manager starts with four knobs and then increases the count by two every 60 min. With each knob count setting, we select the top-k knobs ranked by their impact as described in Section 3.2. We use 15 hour tuning sessions to determine whether the fixed setting can ever achieve the same performance as the incremental approach; we note that this is longer than we expect that a DBA would normally run OtterTune.

MySQL: The results in Figure 3.3a show that the incremental approach enables OtterTune to find a good configuration for MySQL in approximately 45 min. Unlike Postgres and Vector, the incremental approach provides a noticeable boost in tuning performance for MySQL in contrast to the fixed knob settings. The next best knob count setting for MySQL is the fixed four knobs. These four knobs include the DBMS's buffer pool and log file sizes, as well as the method used to flush data to storage. The larger knob count settings include the ability to control additional thread policies and the number of pages prefetched into the buffer pool. But based on our experiments we find that these have minimal impact on performance for a static TPC-C workload. Thus, including these less impactful knobs increases the amount of noise in the model, making it harder to find the knobs that matter.

Postgres: The results in Figure 3.3b show that the incremental approach and the fixed four knob setting provide OtterTune with the best increase in the DBMS's performance. Similar to MySQL, Postgres has a small number of knobs that have a large impact on the performance. For example, the knob that controls the size of the buffer pool and the knob that influences which query plans are selected by the optimizer are both in the four knob setting. The larger fixed knob settings perform worse than the four knob setting because the additional knobs that they contain have little impact on the system's performance. Thus, also tuning these irrelevant knobs just makes the optimization problem more difficult. The incremental method, however, proves to be a robust technique for DBMSs that have relatively few impactful knobs for the TPC-C workload since it slightly outperforms the four knob setting. Its performance continues to improve after 400 min as it expands the number of knobs that it examines. This is because the incremental approach allows OtterTune to explore and optimize the configuration space for a small set of the most impactful knobs before expanding its scope to consider the others.



(a) MySQL (b) Postgres

Figure 3.4: Tuning Evaluation (TPC-C) – A comparison of the OLTP DBMSs for the TPC-C workload when using configurations generated by OtterTune and iTuned.

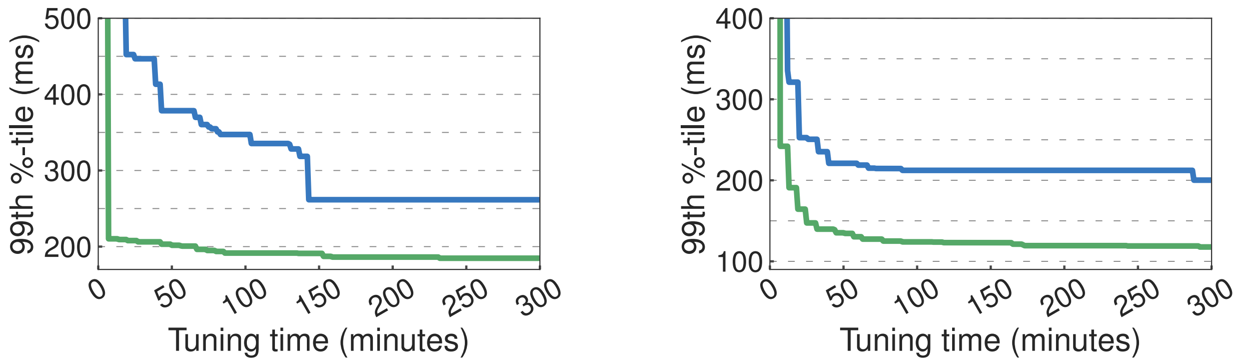
Vector: As shown in Figure 3.3c, OtterTune achieves the best tuning performance with the eight, 16, and the incremental knob settings. In contrast to MySQL and Postgres, tuning only four knobs does not provide the best tuning performance. This is because some of Vector's more impactful knobs are present in the eight knob setting but not in the four knob one. The top four knobs tune the level of parallelism for query execution, the buffer pool's size and prefetching options, and the SIMD capabilities of the DBMS. There is one knob that replaces Vector's standard LRU buffer replacement algorithm with a policy that leverages the predictability of disk page access patterns during long-running scans. This knob can incur overhead due to contention waiting for mutexes. Since the eight knob setting always disables this knob, it is likely the one that prevents the four knob setting from achieving comparable performance.

The optimal number of knobs for a tuning session varies per DBMS and workload, thus it is impossible to provide a universal knob setting. These results show that increasing the number of knobs that OtterTune considers over time is the best approach because it strikes the right balance between complexity and performance. Using this approach, OtterTune is able to tune DBMSs like MySQL and Postgres that have few impactful knobs, as well as DBMSs like Vector that require more knobs to be tuned in order to achieve good performance.

3.4.4 Tuning Evaluation

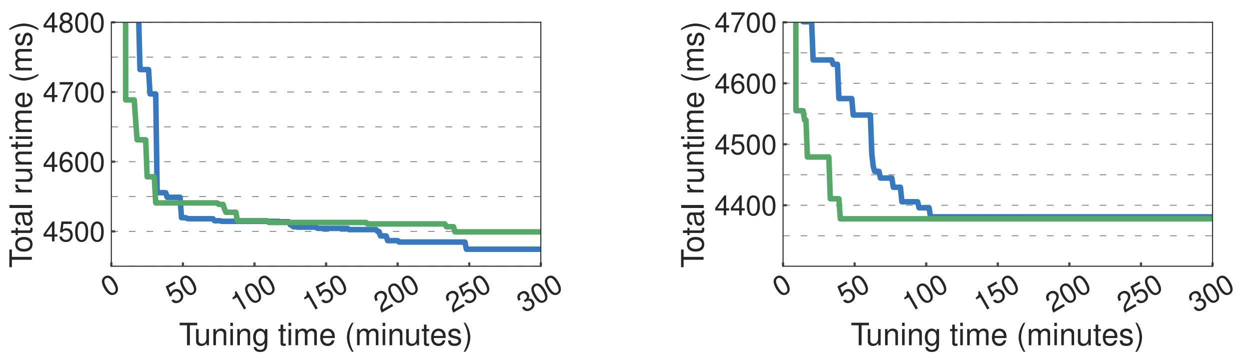
We now demonstrate how learning from previous tuning sessions improves OtterTune's ability to find a good DBMS knob configuration. To accomplish this, we compare OtterTune with another tuning tool, called iTuned [37], that also uses Gaussian Process models to search for an optimal DBMS configuration.

Unlike OtterTune, iTuned does not train its GP 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.



(a) MySQL (b) Postgres

Figure 3.5: Tuning Evaluation (Wikipedia) – A comparison of the OLTP DBMSs for the Wikipedia workload when using configurations generated by OtterTune and iTuned.



(a) Vector (TPC-H #1) (b) Vector (TPC-H #2)

Figure 3.6: Tuning Evaluation (TPC-H) – Performance measurements for Vector running two subsets of the TPC-H workload using configurations generated by OtterTune and iTuned.

iTuned uses the data from these initial experiments to train GP models that then search for the best configuration in same way as described in Section 3.3.2.

For this comparison, we use both the TPC-C and Wikipedia benchmarks for the OLTP DBMSs (MySQL and Postgres) and two variants of the TPC-H workload for the OLAP DBMS (Vector). OtterTune trains its GP models using the data from the most similar workload mixture determined in the last workload mapping stage. Both tuning tools use the incremental knob approach to decide how many knobs to tune during each observation period (see Section 3.2.3). The difference is that iTuned starts using this approach only after it has finished running its initial set of experiments.

TPC-C: The results in Figure 3.4 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 3.4a 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 3.5 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 the 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.

TPC-H: In this last experiment, we compare the performance of the configurations generated by the two tuning tools for two TPC-H workload mixtures running on Vector. Figure 3.6 show that once again OtterTune produces better configurations than iTuned, but that the difference is less pronounced than in the OLTP workloads. The reason is that Vector is less permissive on what values the tuning tools are allowed to set for its knobs. For example, it only lets the DBA set reasonable values for its buffer pool size, otherwise it will report an error and refuse to start. Compare this to the other DBMSs that we evaluate where the DBA can set these key knobs to almost anything. Thus, tuning Vector is a simpler optimization task than tuning MySQL or Postgres since the space of possible configurations is smaller.

3.4.5 Execution Time Breakdown

To better understand what happens to OtterTune when computing a new configuration at the end of an observation period, we instrumented its tuning manager to record the amount of time that it spends in the different parts of its tuning algorithm from Section 3.3. We used TPC-C for MySQL and Postgres, and TPC-H for Vector. The four categories of the execution time are as follows:

- **Workload Execution:** The time that it takes for the DBMS to execute the workload in order to collect new metric data.
- **Prep & Reload Config:** The time that OtterTune's controller takes to install the next configuration and prepare the DBMS for the next observation period (e.g., restarting if necessary).
- **Workload Mapping:** The time that it takes for OtterTune's dynamic mapping scheme to identify the most similar workload for the current target from its repository. This corresponds to Step #1 from Section 3.3.1.
- **Config Generation:** The time that OtterTune's tuning manager takes to compute the next configuration for the target DBMS. This includes the gradient descent search and the GP model computation. This is Step #2 from Section 3.3.2.

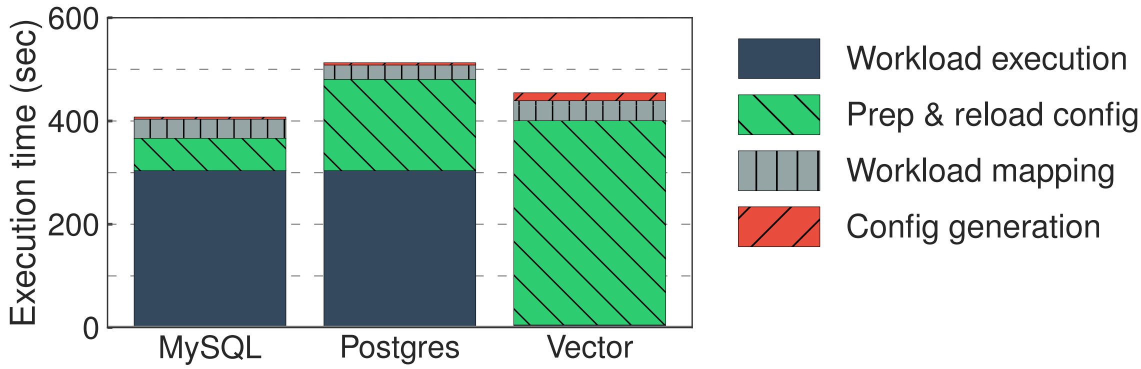


Figure 3.7: Execution Time Breakdown – The average amount of time that OtterTune spends in the parts of the system during an observation period.

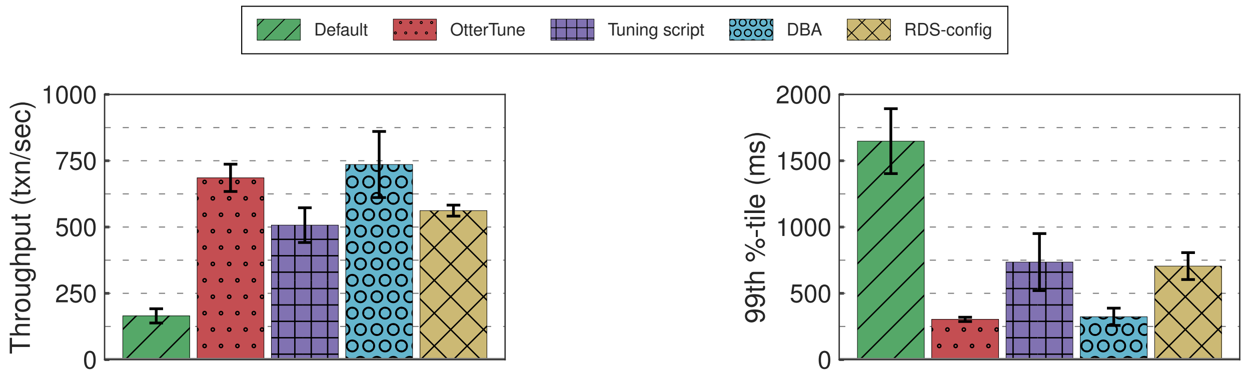
The results in Figure 3.7 show the breakdown of the average times that OtterTune spends during a tuning session. The workload execution time is the largest portion of OtterTune's total time for MySQL and Postgres. This is expected since both of these DBMSs execute the target workload for the 5 min observation period. In contrast, Vector executes a sequence of TPC-H queries that take an average of 5 sec to finish. These results show that it takes OtterTune's controller 62 sec to restart MySQL for each new configuration, whereas Postgres and Vector take an average of 3 min and 6.5 min to restart, respectively. Postgres' longer preparation time is a result of running the vacuum command between observation periods to reclaim any storage that is occupied by expired tuples. For Vector, the preparation time is longer because all data must be unloaded and then reloaded into memory each time the DBMS is restarted. All three DBMSs take between 30–40 sec and 5–15 sec to finish the workload mapping and configuration recommendation steps, respectively. This is because there is approximately the same amount of data available in OtterTune's repository for each of the workloads that are used to train the models in these steps.

3.4.6 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.[12] 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 rest of the experiments. We provide the configurations for these experiments in Appendix A.2.

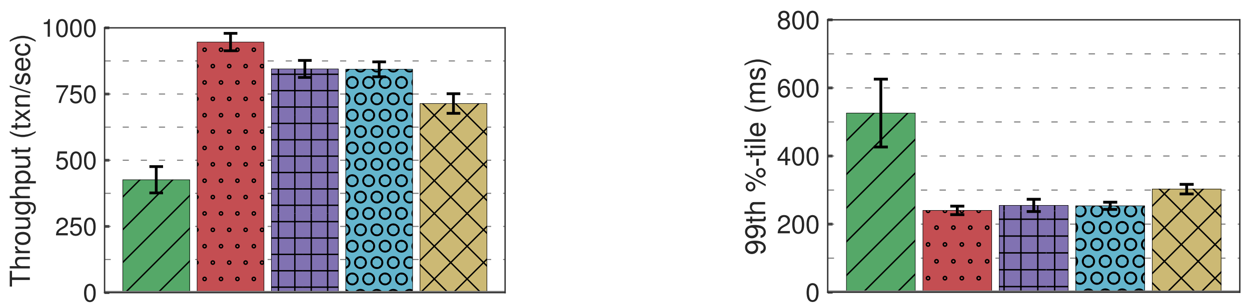
Each DBA was provided with the same EC2 setup used in all of our experiments. They were allowed to tune any knobs they wanted but were not allowed to modify things external to the DBMS (e.g., OS kernel parameters). On the client instance, we provided them with a script to execute the workload for the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

[12] We were unable to obtain a similar tuning tool for Vector in this experiment.



(a) TPC-C (Throughput) (b) TPC-C (99%-tile Latency)

Figure 3.8: 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) DBA configuration, and (5) Amazon RDS configuration.



(a) TPC-C (Throughput) (b) TPC-C (99%-tile Latency)

Figure 3.9: 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.

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 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 3.8 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. These results are consistent with our findings in Section 3.4.3 that show how most of the performance improvement for MySQL comes from tuning the top four knobs.

Both the latency and the throughput measurements in Figure 3.8 show that MySQL achieves ∼22% better throughput and ∼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. As discussed in Chapter 2, OtterTune is forbidden from tuning such knobs.

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` [5]) 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 3.9b 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 3.9 show that Postgres has ∼12% higher performance with OtterTune compared to the DBA and the tuning script, and ∼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 configurations 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.

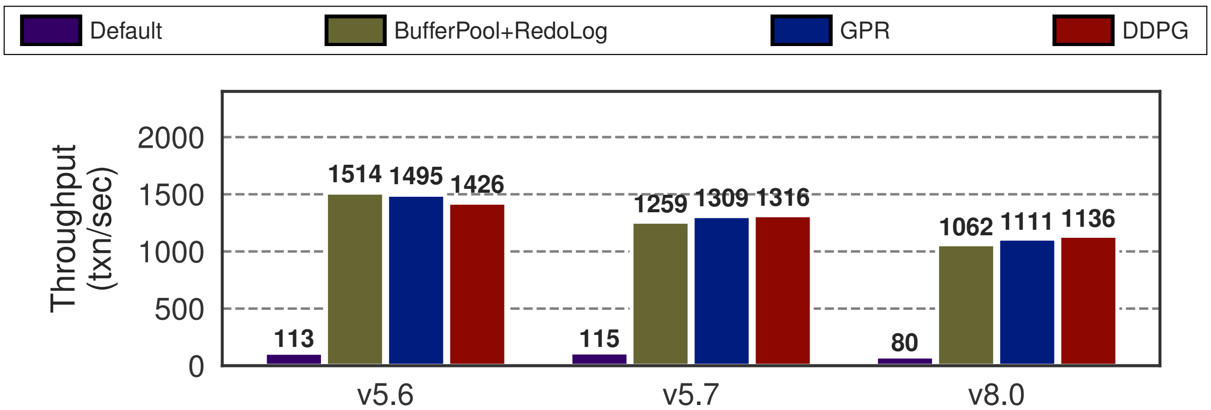
Chapter 4

Tuning in the Real World

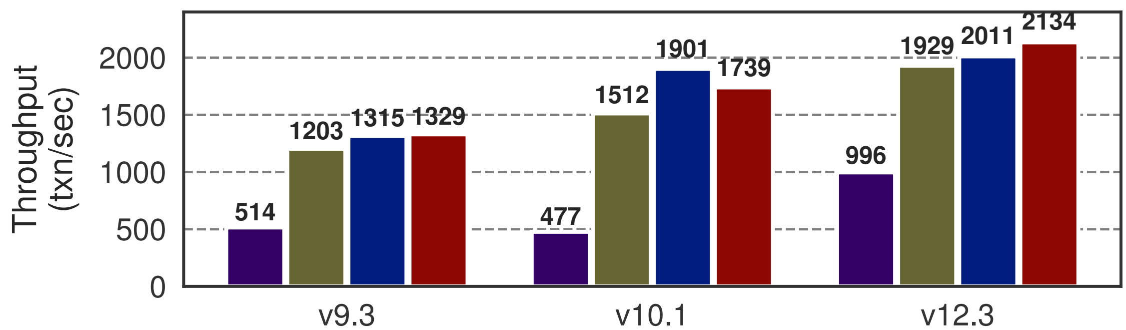
Our work in Chapter 3 and other recent results from ML-based approaches have demonstrated that they achieve better performance compared to human DBAs and other tuning tools on a variety of workloads and hardware configurations [65, 109]. These results are again promising, but up until now, the evaluations have been limited to (1) open-source DBMSs with limited tuning potential (e.g., Postgres, MySQL, MongoDB) and (2) synthetic benchmarks with uniform workload patterns. Additionally, although these evaluations used virtualized environments for the target DBMSs, to the best of our knowledge, they all used dedicated local storage (i.e., SSDs directly attached to the VM). Many real-world DBMS deployments, however, use non-local, shared-disk storage. Such non-local storage includes on-premise SANs and cloud-based block stores (e.g., Amazon EBS, Azure Disk Storage). These non-local storage devices have higher read/write latencies and incur more variance in their performance than local storage. It is unclear how these differences affect the efficacy of ML-based tuning algorithms. Lastly, previous studies are vague about how much of the tuning process was truly automated. For example, they do not specify how they select the bounds of the knobs they are tuning. This means that the quality of the configurations may still depend on a human initializing it with the right parameters.

Given these issues, this 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 an evaluation of state-of-the-art ML-based methods for tuning an enterprise Oracle DBMS (v12) installation running on virtualized computing infrastructure with non-local storage. For this work, we extended the **OtterTune** [4] tuning service to support three ML tuning algorithms: (1) Gaussian Process Regression (GPR) from OtterTune Chapter 3, (2) Deep Neural Networks (DNN) [107], and (3) Deep Deterministic Policy Gradient (DDPG) from CDBTune [109].

This chapter is organized as follows. Section 4.2 begins with an overview of our field study. We then describe the tuning algorithms that we evaluate in Section 4.3. In Section 4.4, we present our evaluation of these algorithms in tuning an enterprise database application. Lastly, we summarize the lessons learned from this study in Section 4.5.



(a) MySQL



(b) Postgres

Figure 4.1: DBMS Tuning Comparison – Throughput measurements for the TPC-C benchmark running on three versions of MySQL (v5.6, v5.7, v8.0) and Postgres (v9.3, v10.1, v12.3) using the (1) default configuration, (2) buffer pool & redo log configuration, (3) GPR configuration, and (4) DDPG configuration.

4.1 Motivation

Automated DBMS tuning services are an active research area. Our work in Chapter 3 and other recent studies have shown that they can generate configurations that are equivalent or exceed configurations created by expert DBAs [65, 109]. Despite these measurable benefits, we observe that there is a mismatch between aspects of the evaluations of previous research on ML-based tuning approaches versus what we see in real-world DBMS deployments. The three facets of this discrepancy are the (1) workload, (2) DBMS, and (3) operating environment. We now discuss them in further detail.

Workload Complexity: Gaining access to production workload data to evaluate new research ideas is non-trivial due to privacy constraints and other restrictions. Prior studies evaluate their techniques using synthetic benchmarks; the most complex benchmark used to evaluate ML-based tuning techniques to date is the TPC-C OLTP benchmark from the early 1990s. But previous studies

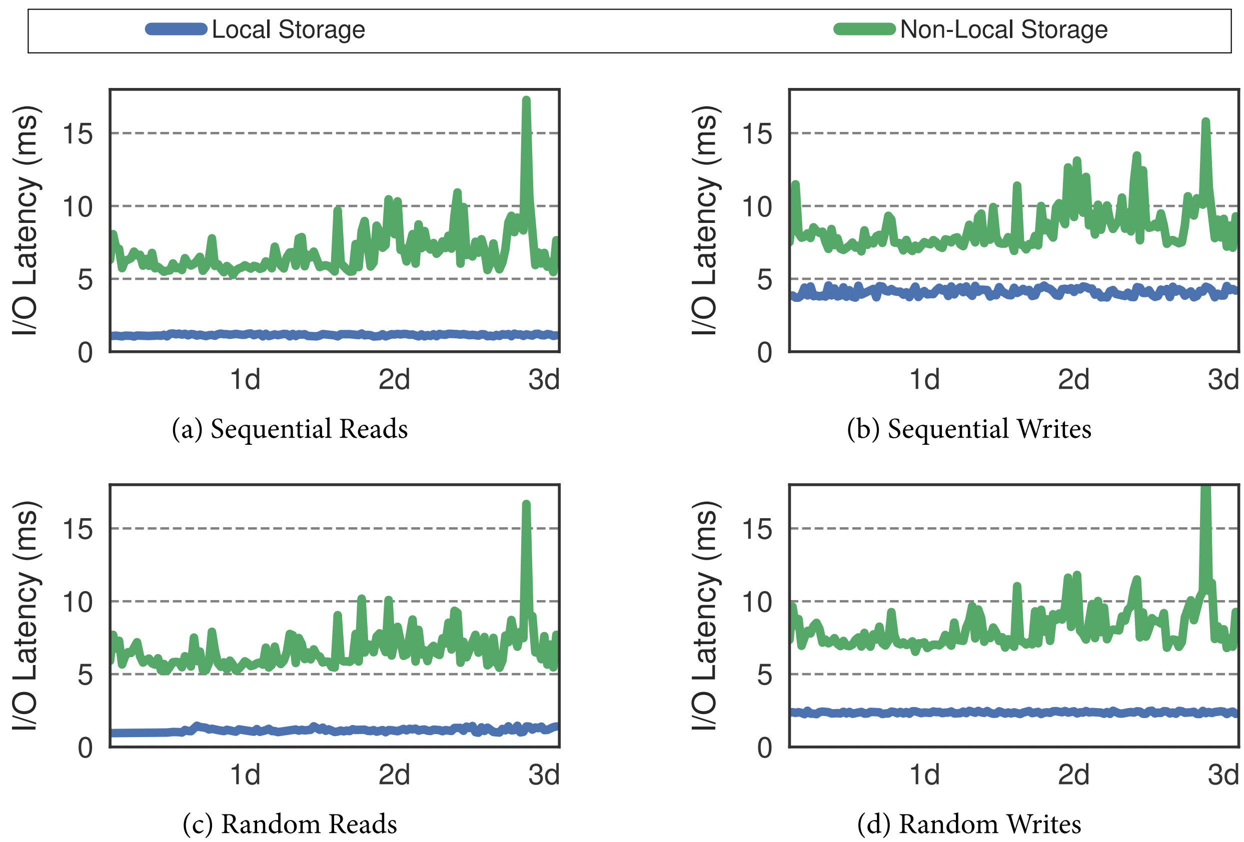


Figure 4.2: Operating Environment – I/O latency of local versus non-local storage for four different I/O workloads over a three-day period.

have found that the characteristics of TPC-C are not representative of real-world database applications [53, 59]. Many of the unrealistic aspects of TPC-C are due to its simplistic database schema and query complexity. Another notable difference is the existence of temporary and large objects in production databases. Some DBMSs provide knobs for tuning these objects (e.g., Postgres, Oracle), which have not been considered in prior work.

System Complexity: The simplistic nature of workloads like TPC-C means that there are fewer tuning opportunities in some DBMSs, especially for the two most common DBMSs evaluated in previous studies (i.e., MySQL, Postgres). For these two DBMSs, one can achieve a substantial portion of the performance gain from configurations generated by ML-based tuning algorithms by setting *two knobs* according to the DBMS's documentation. These two knobs control the amount of RAM for the buffer pool cache[1, 2] and the size of the redo log file on disk.[3, 4]

[1] **Postgres Knob** – `SHARED_BUFFERS`
[2] **MySQL Knob** – `INNODB_BUFFER_POOL_SIZE`
[3] **Postgres Knob** – `MAX_WAL_SIZE`
[4] **MySQL Knob** – `INNODB_LOG_FILE_SIZE`

To illustrate this issue, we ran a series of experiments on multiple versions of MySQL (v5.6, v5.7, v8.0) and Postgres (v9.3, v10.1, v12.3) using the TPC-C workload. We deployed the DBMSs on a machine running Ubuntu 18.04 with an Intel Core i7-8650U CPU (8 cores @ 1.90GHz, 2× HT) and 32 GB RAM. For each DBMS version, we measure the system's throughput under four knob configurations: (1) the OS's default configuration, (2) the recommended settings from the DBMS's documentation for the two knobs that control the buffer pool size and the redo log file size, (3) the configuration generated by OtterTune using GPR, and (4) the configuration generated by OtterTune using DDPG. We allowed OtterTune to tune 10 knobs for both DBMSs as selected by the tuning manager's ranking algorithm. We discuss the details of these algorithms in Section 4.3.

Figure 4.1 shows that the two-knob configuration and OtterTune-generated configurations improve the performance for TPC-C over the DBMS's default settings. This is expected since the default configurations for MySQL and Postgres are based on their minimal hardware requirements. More importantly, however, the configurations generated by ML algorithms achieve only 5–25% higher throughput than the two-knob configuration across the different versions of MySQL and Postgres. That is, one can achieve 75–95% of the performance obtained by ML-generated configurations by tuning only two knobs for the TPC-C benchmark.

Operating Environment: Disk speed is often the most important factor in a DBMS's performance. Although the previous studies used virtualized environments to evaluate their methods, to our knowledge, they deploy the DBMS on ephemeral storage that is physically attached to the host machine. But many real-world DBMS deployments use durable, non-local storage for data and logs, such as on-premise SANs and cloud-based block/object stores. The problem with these non-local storage devices is that their performance can vary substantially in a multi-tenant cloud environment [83].

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 using `Fio` [1]. We conducted the local storage experiments on a machine with a Samsung 960EVO M.2 SSD. We ran the non-local storage experiments on a VM with virtual storage deployed on an enterprise private cloud. The results in Figure 4.2 show that the read/write latencies for the local storage are stable across all workloads. In contrast, the read/write latencies for the non-local storage are higher and more variable. The spike on the third day also demonstrates the unpredictable nature of non-local storage.

4.2 Automated Tuning Field Study

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

We conducted an evaluation of the OtterTune framework at the **Société Générale** (SG) multinational bank in 2020 [9]. SG runs most of their database applications on Oracle on private cloud infrastructure. They provide self-service provisioning for DBMS deployments that use a pre-tuned configuration based on the expected workload (e.g., OLTP vs. OLAP). These Oracle deployments are managed by a team of skilled DBAs with experience in knob tuning. Thus, the goal of our field study is to see whether automated tuning could improve a DBMS's performance beyond what their DBAs achieve through manual tuning.

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

4.2.1 Target Database Application

The data and workload trace that we use in our study came from an internal issue tracking application (TicketTracker) for SG's IT infrastructure. The core functionality of TicketTracker is similar to other widely used project management software, such as Atlassian Jira and Mozilla Bugzilla. This application keeps track of work tickets submitted across the entire organization. SG has ∼140,000 employees spread across the globe [9], and thus TicketTracker's workload patterns and query arrival rate are mostly uniform 24-hours a day during the work week. SG currently runs TicketTracker on Oracle v12.1. We developed custom reporting tools to summarize the contents of the database and query trace. We now provide a high-level description of TicketTracker from this analysis.

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

The TicketTracker database contains 1226 tables, but 773 of them are empty tables from previous staging and testing efforts. We exclude them from our analysis here as no query accesses them. For the remaining 453 tables with data, the database contains 1647 indexes based on them. The charts in Figure 4.3 provide breakdowns of the number of tuples, columns, and indexes per table. Figure 4.3b shows that most of the tables have 20 or fewer columns. There is also a large percentage of tables that only have a single index; these are mostly tables with a small number of tuples (i.e., <10k).

Workload: We collected the TicketTracker workload trace using Oracle's Real Application Testing (RAT) tool. RAT captures the queries that the application executes on the production DBMS instance starting at the snapshot, along with meta-data, such as timing information. It then supports replaying those queries multiple times on a test database with the exact timing, concurrency, and transaction characteristics of the original workload [46]. Our trace is from a two-hour period during regular business hours and contains over 3.6m query invocations.

The majority of the queries (90.7%) that TicketTracker executes are read-only `SELECT` statements. They are short queries that access a small number of tuples. Figure 4.4a shows that the average execution time of `SELECT` queries on Oracle with SG's default configuration is 25 ms. The application executes some longer running queries (e.g., for dashboards), but these are rare. The 99th percentile latency for `SELECT` queries is only 370 ms.

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

Table 4.1: Query Plan Operators – The percentage of queries in the TicketTracker workload that contain each operator type.

We also counted the number of times that a `SELECT` query accesses each table. Only 2% of the queries perform a join between two or more tables; the remaining 98% only access a single table. The histogram in Figure 4.4b shows the top 10 most accessed tables in the workload. The remaining tables are accessed by 1% or less of the queries. These results indicate that there is no single table that queries touch significantly more than others.

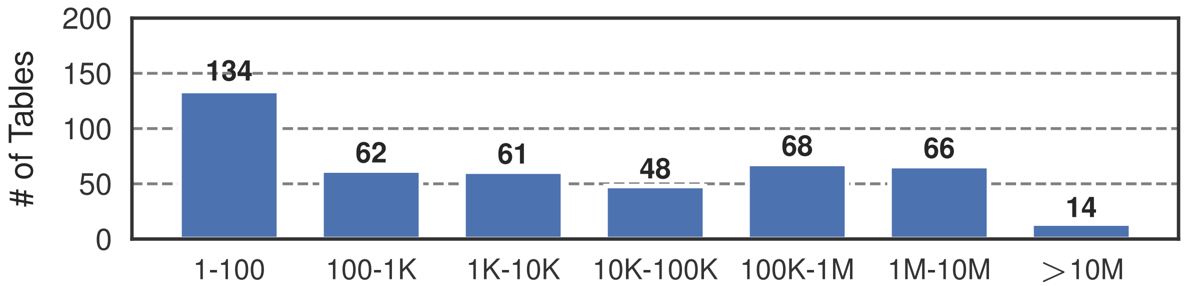
Since the workload trace includes query plans, we extracted the operators for each `SELECT` query to characterize their behavior. This analysis helped us understand whether the configurations selected by the algorithms in our experiments would even affect the queries. Table 4.1 provides a ranked list of the five most common operators. We see that almost all the queries perform index look-ups and scans. The most common operator (`TABLE ACCESS BY INDEX ROWID`) is when the query uses a non-covering index to get a pointer to the tuple. Only 5% of the queries execute a sequential scan on a table.

The rest of the TicketTracker workload contains `UPDATE` (5.2%), `INSERT` (3.4%), and `DELETE` (0.7%) queries. The average execution times of these queries are 18 ms, 97 ms, and 49 ms, respectively. But unlike `SELECT` queries, the 99th percentile latency for `INSERT` and `DELETE` is an order of magnitude longer than their average latency. For `INSERT`s, Figure 4.4a shows that some queries take 1260 ms to run. Our analysis also shows that a large portion of the modification queries are on tables with over 100k tuples. Some of the largest tables (i.e., >10m tuples) are never used in `SELECT` queries.

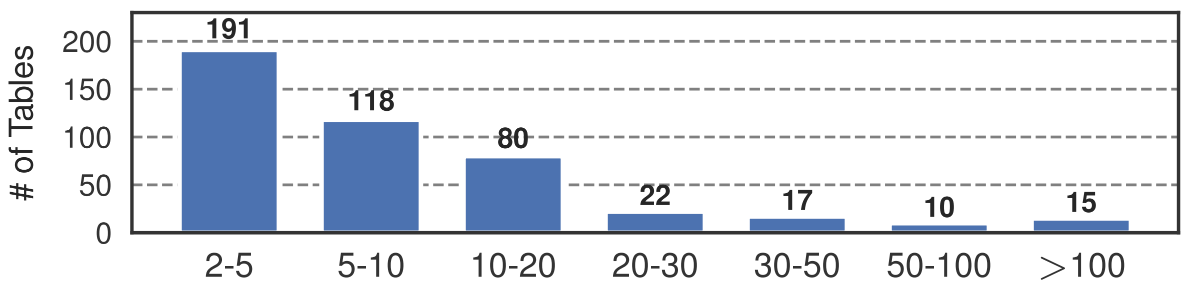
There are important differences in the TicketTracker application compared to the TPC-C benchmark used in previous ML tuning evaluations. Foremost is that the TicketTracker database has hundreds of tables and the TPC-C database only has nine. TPC-C also has a much higher write ratio for queries (46%) than the TicketTracker workload (10%). This finding is consistent with previous work that has compared TPC-C with real-world workloads [53, 59]. Prior to our study, it was unknown whether these differences affect the efficacy of ML-based tuning algorithms.

4.2.2 Deployment

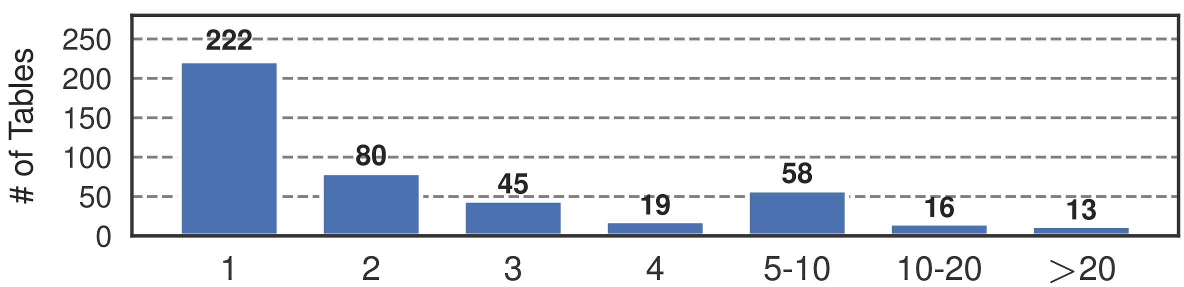
We deployed five copies of the TicketTracker database and workload on separate Oracle v12.2 installations in SG's private cloud. We used the same hardware configuration as the production instance. Each DBMS instance runs on a VM with 12 vCPUs (Intel Xeon CPU E5-2697v4 at 2.30 GHz)



(a) Number of Rows Per Table



(b) Number of Columns Per Table

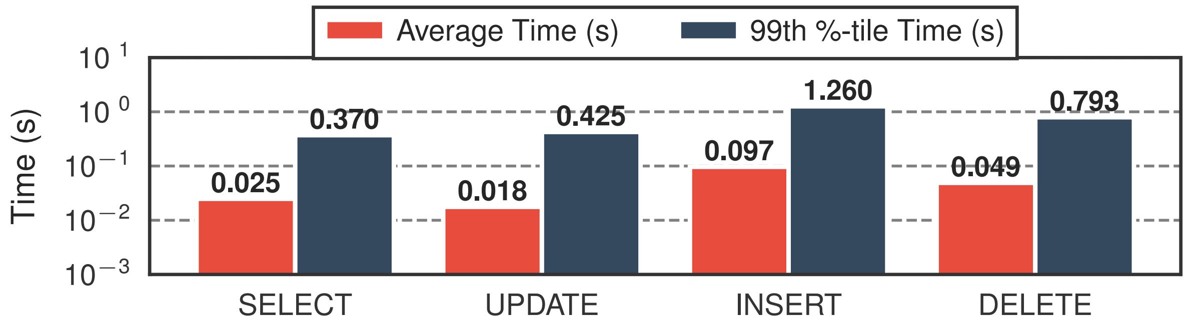


(c) Number of Indexes Per Table

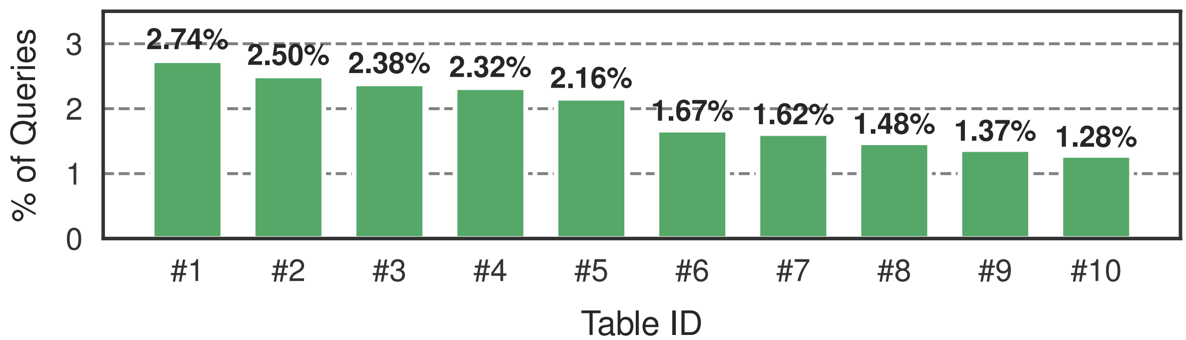
Figure 4.3: Database Contents Analysis – The number of tuples, columns, and indexes per table for the TicketTracker database.

and 64 GB RAM. We configured the VMs to write to a NAS shared-disk running in the same data center. As shown in our previous experiment in Figure 4.2, the average read and write latencies for this storage are $\sim$6.7 ms and $\sim$8.3 ms, respectively.

The initial knob configuration for each Oracle instance is selected from a set of pre-tuned configurations that SG uses for their entire fleet. The SG IT team provides their employees with a self-service web interface for provisioning new DBMSs. In addition to selecting the hardware configuration of a new DBMS (e.g., CPU cores, memory), a user must also specify the expected workload that the DBMS will support (e.g, OLTP, OLAP, HTAP). The provisioning system installs the knob configuration that has been pre-tuned by the SG administrators for the selected workload type. Although these configurations outperform Oracle's default settings, they only modify 4–6 knobs and



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



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

Figure 4.4: TicketTracker Workload Analysis – Execution information for the TicketTracker queries extracted from the workload trace.

are still not tailored to the individual applications' workloads. As such, for the TicketTracker workload, the DBA further customized some of the knobs in the pre-tuned configuration, including one that improves the performance of LOBs.[5]

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

4.2.3 Tuning

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

[5] **Oracle Knob** – `DB_32K_CACHE_SIZE`

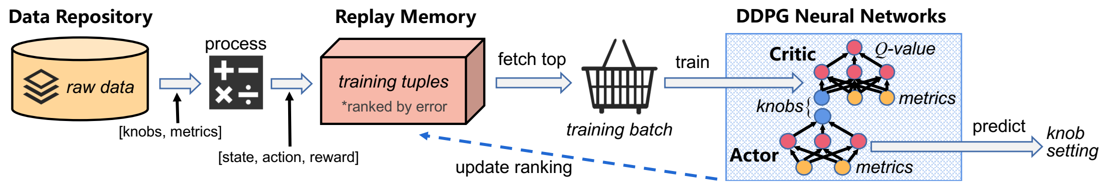


Figure 4.5: DDPG Tuning Pipeline – The raw data is converted to states, actions and rewards and then inserted into the replay memory. The tuples in the replay memory are ranked by the error of the predicted Q-value. In the training process, a batch of top tuples are fetched to update the critic and the actor. After training, the prediction error in the replay memory is updated and the actor recommends the next configuration to run.

Another issue is that Oracle could refuse to start if one of its knobs has an invalid setting. For example, if one sets the knob that controls the DBMS's buffer pool size[6] to be larger than the amount of physical memory on the underlying machine, then Oracle's will not start and prints an error message in the log. If the controller detects this failure, it halts the tuning iteration, reports the failure to the tuning manager, and then starts a new iteration with the next configuration. This failure is still useful for the tuning algorithms; we discuss how to handle this and other failure scenarios in Section 4.5.

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

After resetting the DBMS, the controller executes a `Fio` [1] microbenchmark on the DBMS's VM to collect the current performance measurements for its shared disk. This step is not necessary for tuning, and none of the algorithms use this data in their models. Instead, we use these metrics to explain the DBMSs' performance in noisy cloud environments (see Section 4.5).

Now the controller begins the execution step on the target DBMS using the current configuration. It first retrieves the current values for DBMS's metrics through Oracle-specific SQL commands. Oracle generates over 3900 metrics that are a mix of counters and aggregates. We only collect global metrics from the DBMS (i.e., there are no table- or index-specific metrics). We set the tuning algorithm's target objective function to *DB Time* [35, 40]. This is an Oracle-specific metric that measures the total time spent by the database in processing user requests. A key feature of DB Time is that it provides a "common currency" to measure the impact of any component in the system. It is the SG DBAs' preferred metric because it allows them to reason about the interactions between DBMS components to diagnose problems.

OtterTune's controller executes TicketTracker's workload trace using Oracle RAT. We use RAT's automatic setup option to determine the number of client threads that it needs to replicate the same concurrency as the original application. We configure RAT to execute a 10-minute segment (230k queries) from the original trace. We limit the replay time for two reasons. First, the segment's

[6] **Oracle Knob** – `DB_CACHE_SIZE`

timespan is based on the wall clock of when the trace was collected on the production DBMS. This means that when the trace executes on a DBMS with a sub-optimal configuration (which is often the case at the beginning of a tuning session), the 10-minute segment could take several hours to complete. We halt replays that run longer than 45 minutes. The second reason is specific to Oracle: RAT is unstable on large traces for our DBMS version. Oracle's engineers did provide SG with a fix, but only several months after we started our study, and therefore it was too late to restart our experiments.

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

4.3 Tuning Algorithms

Our goal is to understand how the DBMS configuration tuning algorithms proposed in recent years behave in a real-world setting and under what conditions one performs better than others. To this end, we evaluated three ML tuning algorithms: (1) Gaussian Process Regression (GPR), (2) Deep Neural Networks (DNN), and (3) Deep Deterministic Policy Gradient (DDPG). Although there are other algorithms that use query data to guide the search process [65], they are not usable at SG because of privacy concerns since the queries contain user-identifiable data. Methods to anonymize this data are outside the scope of this paper.

GPR is the original algorithm supported by the OtterTune tuning service from Chapter 3. We extended OtterTune to support DNN and DDPG, which we now discuss in further detail.

4.3.1 DNN — OtterTune (2019)

Previous research has argued that Gaussian process models do not perform well on larger data sets and high-dimensional feature vectors [61]. Given this, we modified our GPR-based algorithm from Chapter 3 to use a deep neural network (DNN) instead of the Gaussian process models. OtterTune's DNN algorithm follows the same ML pipeline as GPR (see Figure 3.1).

DNN relies on a deep learning algorithm that applies a series of linear combinations and non-linear activations to the input to derive the output. The network structure of the DNN model has two hidden layers with 64 neurons each. All of the layers are fully connected with *rectified linear units* (ReLU) as the activation function. We implemented a popular technique called *dropout regularization* to avoid overfitting the models and improve their generalization [88]. It uses a dropout layer between the two hidden layers with a dropout rate of 0.5. DNN also adds Gaussian noise to the parameters of the neural network during the knob recommendation step [79] to control the amount of exploration versus exploitation. Specifically, OtterTune increases exploitation throughout the tuning session by reducing the scale of the noise.

4.3.2 DDPG — CDBTune (2019)

This method was first proposed by CDBTune [109]. DDPG is a deep reinforcement learning algorithm that searches for the optimal policy in a continuous action space environment. The ability to work on a continuous action space means that DDPG can set a knob to any value within a range, whereas other reinforcement learning algorithms, such as Deep-Q learning, are limited to setting a knob from a finite set of pre-defined values. We first describe CDBTune's DDPG, and then we present an extension to it that we developed to improve its convergence rate.

As shown in Figure 4.5, DDPG consists of three components: (1) *actor*, (2) *critic*, and (3) *replay memory*. The actor is a neural network that chooses an action (i.e., what value to use for a knob) based on the given states. The critic is a second neural network that evaluates the selected action based on the states. In other words, the actor decides how to set a knob value, and then the critic provides feedback on this choice to guide the actor. In CDBTune, the critic takes the previous metrics and the recommended knobs as the input and outputs a Q-value, which is an accumulation of the future rewards. The actor takes the previous metrics as its input and outputs the recommended knobs. The replay memory stores the training data tuples ranked by the prediction error in descending order.

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

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

4.4 Evaluation

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

Random sampling methods serve as competitive baselines for judging optimization algorithms because they are simple yet surprisingly effective [14]. In our evaluation, we use a random sampling method called Latin Hypercube Sampling LHS) [52] as a baseline. LHS is a space-filling technique that attempts to distribute sample points evenly across all possible values. Such techniques are

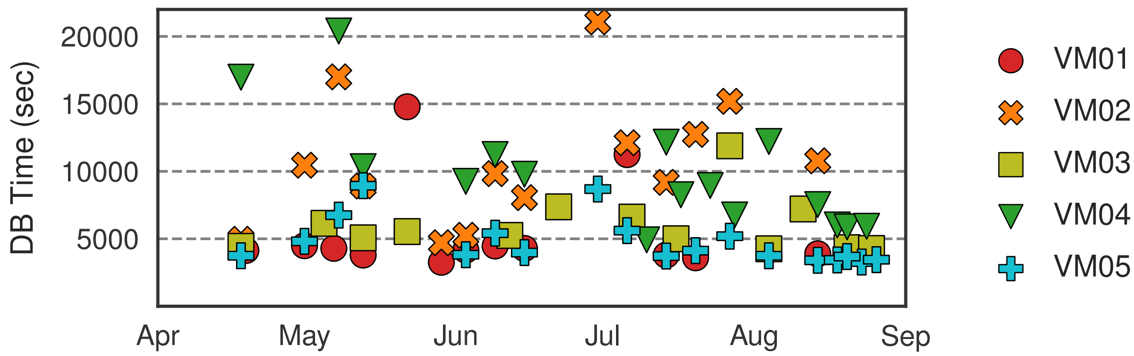


Figure 4.6: Performance Variability – Performance for the TicketTracker workload using the default configuration on multiple VMs over six months.

generally more effective than naïve random sampling in high-dimensional spaces, especially when collecting a small number of samples relative to the total number of possible values.

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

4.4.1 Performance Variability

Because each tuning session in our experiments takes multiple days to complete, we deployed the Oracle DBMS on multiple VMs to run the sessions in parallel. Our VMs run on the same physical machines during this time, but the other tenants on these machines or in the same rack may change. As discussed in Section 4.1, running a DBMS in virtualized environments with shared storage can lead to unexplained changes in the system's performance across instances with the same hardware allocations and even on the same instance.

To better understand the extent of this variability in SG's data center, we measured the performance of our VMs once a week over six months. We run the 10-minute segment of the TicketTracker workload using SG's default configuration. The results in Figure 4.6 show the DB Time metric for each VM instance over time. The first observation from this data is that the DBMS's performance on the same VM can fluctuate by as much as 4× even though the DBMS's configuration and workload are the same. For example, VM02's DB Time in July is higher than what we measured in the previous month. The next observation is that the relative performance of VMs can vary as well, even within a short time window.

We believe that these inconsistent results are due to latency spikes in the shared-disk storage. Figure 4.7 shows the DBMS's performance for one VM during a tuning session, along with its CPU busy time and I/O latency as measured by the DBMS. These results show a correlation between spikes in the I/O latency (three highlighted regions) and a degradation in the DBMS's performance. In this example, the algorithm had converged at this point of the tuning session, so the configuration was stable. Thus, it is likely that these latency spikes are due to external causes outside of the DBMS's control.

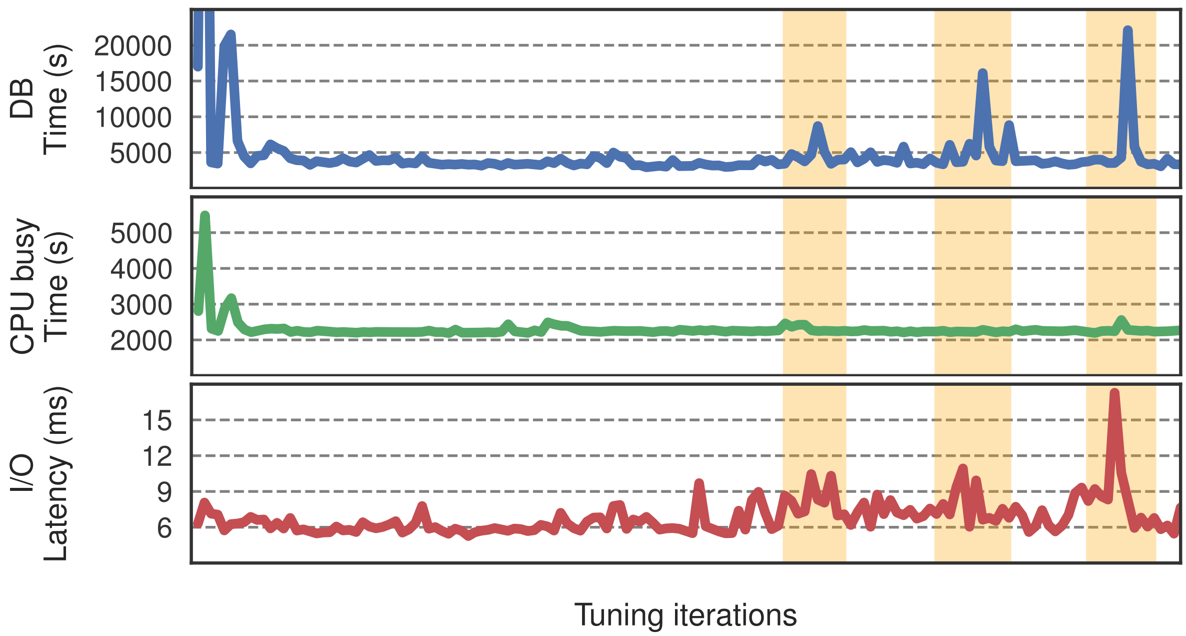


Figure 4.7: Effect of I/O Latency Spikes – Runtime measurements of DBMS performance with CPU utilization and I/O latency.

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

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

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

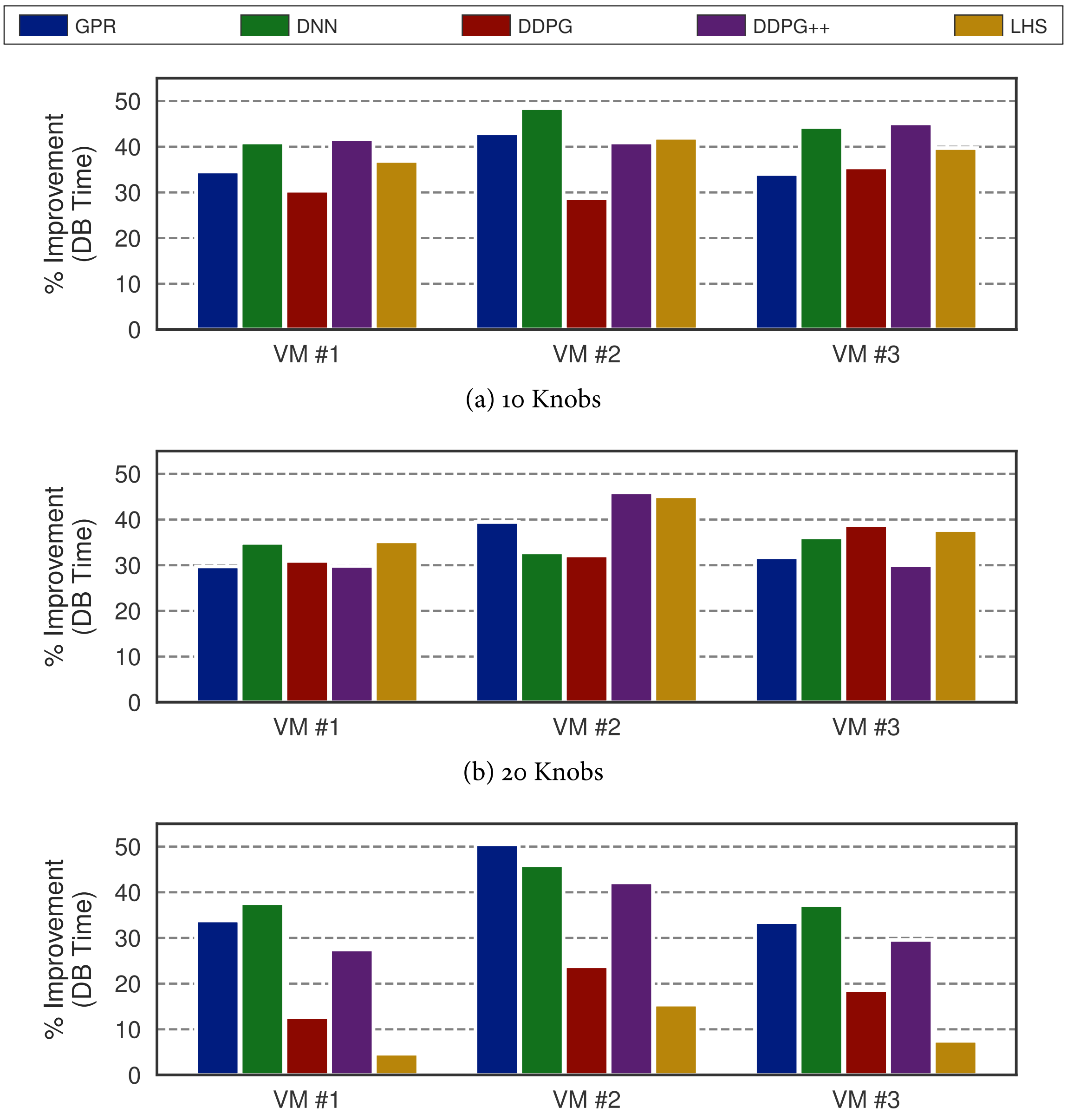


Figure 4.8: Tuning Knobs Selected by DBA (Per VM) – The performance improvement of the best configuration per algorithm running on separate VMs relative to the performance of the SG default configuration measured at the beginning of the tuning session.

4.4.2 Tuning Knobs Selected by DBA

This first experiment evaluates the quality of the configurations that the tuning algorithms generate when increasing the number of knobs that they tune. Although Oracle exposes over 400 knobs, we limit the maximum number of knobs tuned to 40. This limit is for two reasons. First, we want to evaluate how much better the ML algorithms are at ranking the importance of knobs

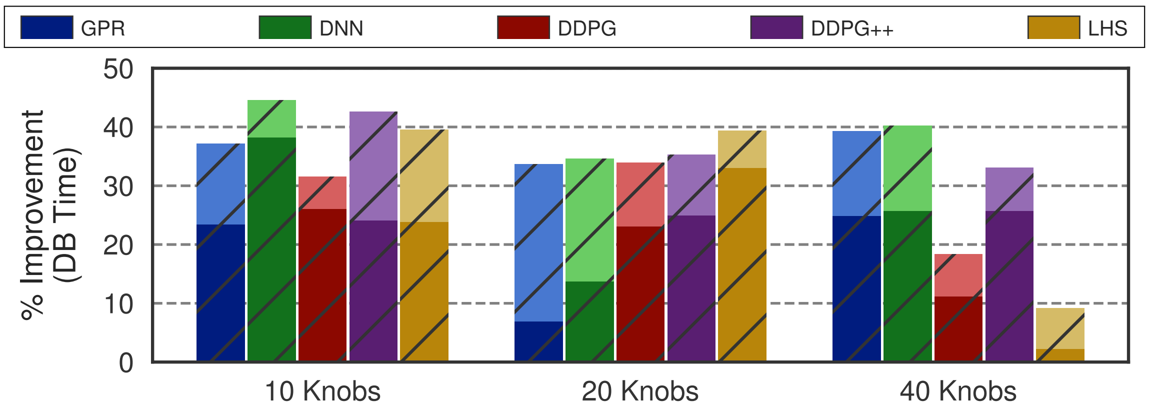


Figure 4.9: Tuning Knobs Selected by DBA – Performance measurements for 10, 20, and 40 knob configurations for the TicketTracker workload. The shading on each bar indicates the minimum and maximum performance of the best configurations from three tuning sessions.

versus a DBA-selected ranking. Asking a human to select more than 40 knobs to tune is unrealistic and will produce random results. The second reason is to reduce the time that the algorithms need to converge because the more knobs there are, the harder it is to tune. Since each iteration of the TicketTracker workload takes up to 45 minutes, it would potentially take weeks for the models to converge. Hence, we consider a maximum of 40 knobs that the DBA selected and ordered based on their expected impact on the DBMS's performance.

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

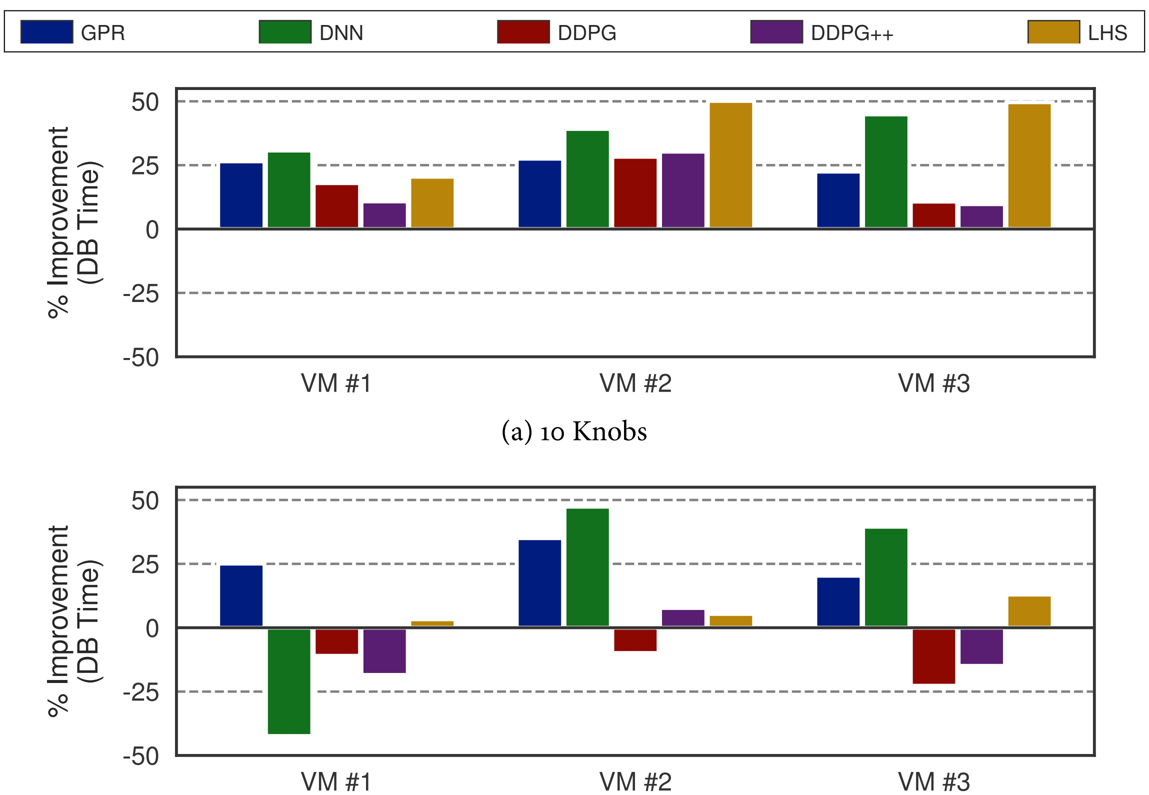
Figure 4.9 shows the performance improvement in the DB Time over SG's default configuration for the best configurations generated by the algorithms from three tuning sessions when optimizing 10, 20, and 40 knobs. The dark and light portions of each bar represent minimum and maximum performance per algorithm, respectively. Figure 4.8 shows the performance improvement of the best configuration per algorithm from Figure 4.9 by VM. The results show that although the absolute measurements vary, the performance rankings of the algorithms are consistent across the VMs.

To understand why the configurations perform differently, we manually examined each configuration and identified three Oracle knobs that have the most impact when the algorithms fail to set them correctly. Table 4.2 shows the knobs' value in the SG default configuration and their best observed value(s) from our experiments. The first two control the size of the DBMSs' main buffer caches. One of these caches is for the DBMS's 8 KB buffers for regular table data, and the other is for 32 KB buffers that the DBMS uses for LOB data. The third knob enables optimizer features based on an Oracle release; this is a categorical variable with seven possible values.

Figure 4.9 shows that the configurations recommended by DNN and DDPG++ that tune 10 knobs improve the DB Time by 45% over the default configuration. Although LHS, GPR, and DDPG achieve over 35% better DB Time, they do not perform as well as those generated by DNN and DDPG++ because they select a sub-optimal version of the optimizer features to enable.

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

Table 4.2: Most Important Knobs – The three most important knobs for the TicketTracker workload with their default and best observed values.



(a) 10 Knobs

(b) 20 Knobs

Figure 4.10: Tuning Knobs Ranked by OtterTune (Per VM) – The performance improvement of the best configuration per algorithm running on separate VMs relative to the performance of the SG default configuration measured at the beginning of the tuning session.

For the 20-knob configurations, Figure 4.9 shows that all the algorithms improve the DBMS's performance by 33–40% over the default configuration. Each algorithm, however, sets at least one of the important knobs in Table 4.2 incorrectly. This is why their configurations do not perform as well as the 10-knob configurations. We also see that DNN has the largest gap between its minimum and maximum best configurations. This is generally due to the randomness in the exploration of the algorithms and the amount of noise on the VM during a given tuning session.

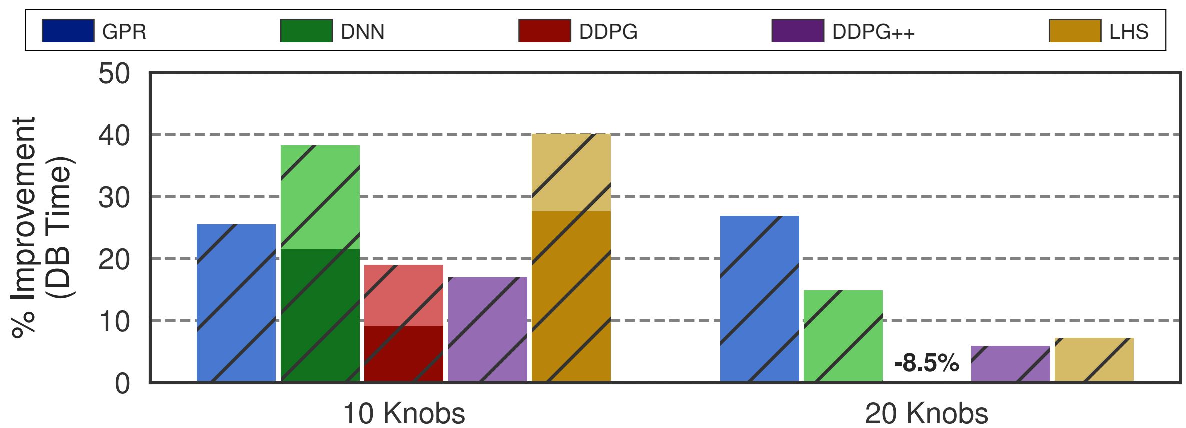


Figure 4.11: Tuning Knobs Ranked by OtterTune – Performance measurements for the ML algorithm configurations using 10 and 20 knobs selected by OtterTune's Lasso ranking algorithm. The shading on each bar indicates the min and max performance of the best configurations from three tuning sessions.

As shown in Figure 4.9, the configurations from DNN and GPR achieve 40% better DB Time than the default configuration. DDPG and DDPG++ only achieve 18% and 32% improvement, respectively. The reason is that neither of them can fully optimize the 40 knobs within 150 iterations. DDPG++ outperforms DDPG because of the optimizations that help it converge more quickly (see Section 4.3.2). With more iterations, DDPG would likely achieve similar performance to the other ML-based algorithms. But due to computing costs and labor time, it was not practical to run a session for more than 150 iterations in our evaluation. The LHS configuration performs the worst of all, achieving only 10% improvement over the default. This shows how sampling techniques like LHS can be inefficient for high-dimensional spaces.

In summary, we find that the configurations generated by all of the algorithms that tune 10, 20, and 40 knobs can improve the DBMS's performance over the default configuration. GPR always converges quickly, even when optimizing 40 knobs. The issue with GPR is that once it converges, it stops exploring and thus does not continue to improve after that point. GPR is also prone to getting stuck in local minima. The performance of GPR, therefore, depends on whether it explores the ranges of the impactful knobs that provide the most improvement in performance. We observe that the performance of GPR is influenced by the initial LHS samples executed at the start of the tuning session. This is consistent with findings from previous studies [61]. In contrast, DNN, DDPG, and DDPG++ require more data to converge and carry out more exploration. The configurations that tune 10 knobs perform the best overall. This is because the three most impactful knobs in Table 4.2 are present in the set of 10 knobs, and the lower complexity of the configuration space enables DNN and DDPG++ to find good settings for those knobs as well as others.

4.4.3 Tuning Knobs Ranked by OtterTune

Our comparison in the previous experiment used DBA-selected knobs. We next measure the quality of the configurations when we remove the human entirely from the tuning process and use

OtterTune's Lasso algorithm described in Section 3.2 to select the knobs to tune for all the algorithms. This arrangement is pertinent because, in real-world deployments, a DBA may not be available to choose what knobs to tune or may not be able to rank them correctly. To generate this list of knobs, we train Lasso on the data collected from the experiments in Section 4.4.2. We then use Lasso to rank the knobs based on their estimated influence on the target objective function and split this list into two sets of 10 and 20 for the algorithms to tune. We again initialize the ML models by executing 10 configurations generated by LHS.

When comparing the knob rankings selected by OtterTune and the DBA, we find that five of the top 10 knobs selected by OtterTune also appear in the DBA's top 10 knobs. For the top 20 OtterTune-selected knobs, 11 of them overlap with the ones chosen by the DBA. Crucially, OtterTune's top 10 knobs include the three most important knobs from Table 4.2.

Figure 4.11 shows the DBMS performance for 10 and 20 knob configurations. The results indicate that the LHS-generated configuration with 10 knobs improves the DB Time by 48% over the default configuration. The 10-knob configuration from DNN performs the next best, achieving 42% better DB Time performance. GPR, DDPG, and DDPG++ have similar improvements of 30% for 10 knobs. LHS and DNN generate configurations with the ideal settings for the three most important knobs in Table 4.2, whereas the other algorithms have incorrect settings for at least one of them. We could not identify any knob in LHS's configuration that explains the 3% improvement over the best configuration in Figure 4.9.

For 20 knobs, the results in Figure 4.11 show that the optimized configurations for GPR and DNN achieve 35% better DB Time than the default configuration. The configurations generated by DDPG++ and LHS improve the performance by only 10%. For DDPG, none of the optimized configurations from the three tuning sessions improved over the default configuration. Likewise, none of the worst-performing 20-knob configurations were able to achieve better performance than the default. We believe the overall poor performance of the 20-knob configurations is at least partly due to more shared storage noise at the beginning of August 2020 when we ran these experiments. The variability in the performance measurements at that time supports this explanation (see Figure 4.6). All the ML-based algorithms take longer to converge when the performance of the VM is unstable. This especially impacts DDPG and DDPG++ since they take longer to converge in general.

We also report the performance improvement of the best configuration for each algorithm by VM in Figure 4.10. For 10 knobs, the results show that the performance measurements from VM #1 are lower than the other VMs, but that the performance trends of the algorithms are similar. Figure 4.10b shows that for the 20-knob configurations, the improvements achieved by the algorithms are mostly stable across the three VMs. The exception is DNN, which performs the best overall on VMs #2 and #3 but then the worst on VM #1.

The improvements of the algorithms when tuning the top tune 10 and 20 knobs ranked by OtterTune are comparable to the DBA-ranked knobs shown in Figure 4.9. The knobs ranked by OtterTune achieve similar gains partly because the Lasso algorithm correctly identified the importance of the three knobs in Table 4.2. Although one set of knobs between those ranked by OtterTune and the DBA is likely to be superior, the cloud environment makes this difficult to determine since smaller improvements are attributed to noise.

4.4.4 Adaptability to Different Workload

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

We first train models for each algorithm using a TPC-C workload executed by OLTP-Bench [36]. We configured the benchmark to use 200 warehouses (∼20 GB) with 50 terminals. We then ran the workload for 10 minutes and captured the queries using Oracle's RAT tool. Next, we tune the top 20 knobs selected by the DBA and train each TPC-C model for 150 iterations. We then use the TPC-C model to tune the TicketTracker workload for 20 iterations.

Figure 4.12 shows the DBMS's performance for the best configurations selected by each algorithm. These results show that DNN's configuration performs the best of all the algorithms, improving the DB Time by 18% over the default configuration. DDPG performs nearly as well and achieves 15% better DB Time. The configurations generated by GPR and DDPG++ improve performance by 12%.

Figure 4.12a shows the improvement achieved by the algorithms per VM. Although the algorithms' rankings based on performance are similar for the three VMs, the performance gains on VM #3 are much larger than the other VMs. The reason is that we calculate the improvement of each algorithm relative to the performance of the SG default configuration, which is particularly bad on VM #3.

We examined the configurations for differences in the best-observed settings for TPC-C and TicketTracker that may explain why the algorithms were unable to achieve performance comparable to the results in Figures 4.9 and 4.11. We observed that none of the algorithms changed the sizes of the two buffer caches in Table 4.2 from their SG default settings. This is expected for the LOB buffer cache since none of TPC-C's data is stored in the 32 KB tablespace. For the main buffer cache, it is possible that the benefit of increasing its size is negligible since TPC-C's working set size is small.

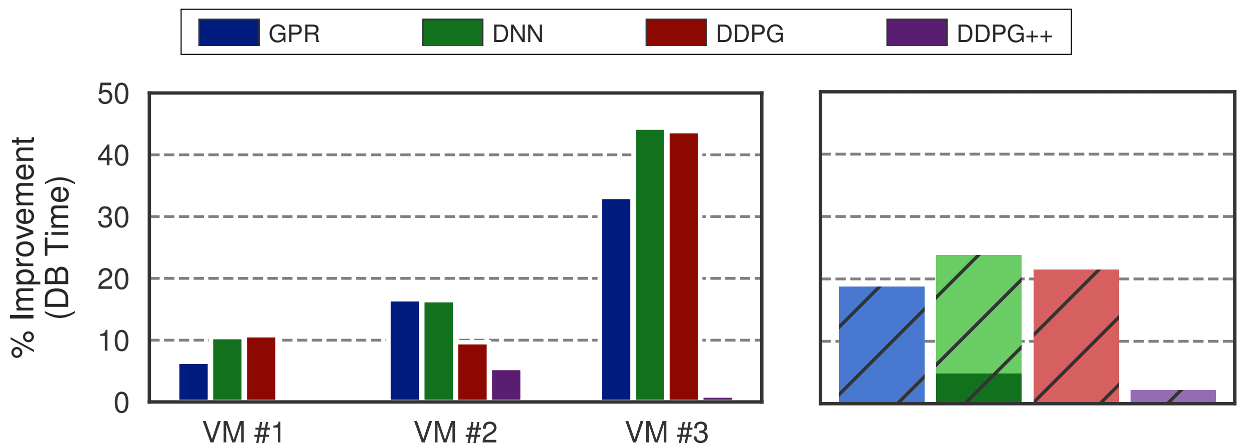
We also found contrary settings for the knob that specifies file I/O operations.[7] Its best observed setting for TicketTracker enables direct I/O whereas for TPC-C it disables it. One possibility is that the OS page cache is more efficient than the DBMS's buffer cache for TPC-C because it consists of mostly small writes. This would also explain why the size of the buffer cache was small.

4.4.5 Execution Time Breakdown

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

1. **Restore Database:** Reset the database to its initial state.
2. **Collect Storage Metrics:** Run the `Fio` microbenchmarks on the DBMS's underlying storage device.

[7] **Oracle Knob** – `FILESYSTEM_IO`



(a) Performance Per VM (b) Performance Summary

Figure 4.12: Adaptability to Different Workloads – Performance comparison when applying the model trained on TPC-C data to the TicketTracker workload.

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

Table 4.3: Execution Time Breakdown – The median amount of time spent in different parts of the system during a tuning iteration.

3. **Prepare Workload:** Run the Oracle RAT procedures to initialize and prepare for the next workload replay.
4. **Execute Workload:** Replay the workload trace.
5. **Collect Data:** Retrieve knob/metric data and generate the summary reports provided by Oracle.
6. **Download Configuration:** Upload the new result to the OtterTune service and download the next configuration to try.
7. **Update Configuration:** Install the next configuration.

Table 4.3 shows the breakdown of the median time spent in each category during a tuning iteration. As expected, most of the time is spent executing the workload trace. Although we replay a 10-minute segment of the trace, the actual time it takes to execute it can be longer if the DBMS's

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

Table 4.4: Workload Replay Time per Algorithm – The median workload execution time and the percentage of replays canceled for the algorithms.

configuration has bad settings or there are external factors affecting the VM's performance (e.g., high I/O latency due to resource contention). The median time it takes to replay the workload is $\sim$16 minutes, but it can take up to 45 minutes. Long-running replays lasting more than 45 minutes are automatically canceled (see Section 4.5).

The next highest percentage of time is spent restoring the database to its original state after the workload replay. This process takes approximately five minutes to complete since the system only needs to restore the modified pages.

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

Of the categories shown in Table 4.3, the only two that vary per algorithm are *Execute Workload* and *Download Config*. Table 4.4 shows the median workload execution time and the percentage of replays canceled for each algorithm. Both the execution time and the replay cancel rate are related to how quickly the algorithm converges. As an algorithm learns more, it is less likely to select poor configurations. Thus, the number of long-running replays decreases as the algorithm nears convergence. Table 4.4 shows that GPR has the fewest canceled replays. DDPG++ has fewer canceled replays than DDPG due to its improved convergence rate (see Section 4.3.2). LHS has the highest workload execution time and percentage of canceled replays because it is a sampling technique and never converges.

4.5 Lessons Learned

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

(1) Handling Long-running Configurations: As discussed in Section 4.1, prior studies on ML-based tuning relied on synthetic benchmarks in their evaluations. Benchmarks like TPC-C are fixed workloads that can be executed for a specific amount of time. Bad knob configurations and other performance factors do not affect the execution time. Conversely, the TicketTracker workload's execution time depends on how long it takes to replay the queries in that trace. Thus, the DBMS's performance affects how long this will take. We found in our experiments that a poor knob configuration could increase the execution of SG's 10-minute trace to several hours. We also observed that the trace took longer to execute during periods when the VMs were experiencing higher I/O latencies.

Given this, the controller needs to support an early abort mechanism that stops long-running workload replays. Setting the early abort threshold to lower values is beneficial because it reduces the total tuning time. This threshold, however, must be set high enough to account for variability in cloud environments. We found that the 45-minute cut-off worked well, but further investigation is needed on more robust methods.

For early aborted configurations, the DBMS's metrics, especially Oracle's DB Time, are incorrectly smaller because the workload is cut off. Thus, to correct this data, the controller calculates a *completion ratio* as the number of finished transactions divided by the number of total transactions in the workload. It then uses this ratio to scale all counter metrics to approximate what they would have been if the DBMS executed the full workload trace.

(2) Handling Failed Configurations: If the ML algorithms do not have prior training data, they will inevitably select poor configurations to install on the DBMS at the beginning of a tuning session. There are two kinds of configurations that cause failures, and each one must be handled by the tuning service differently. The first of these prevents the DBMS from even starting. The most common case is when a knob's value exceeds its allowable bounds. For example, some knobs related to memory cannot be set to a value higher than the available RAM. But this problem also occurs when an implicit dependency that exists between knobs is violated. For Oracle, such a dependency exists between two knobs that configure the DBMS's "shared" pool for SQL statements. One of these knobs controls the total size of the pool and the other specifies how much of the pool to reserve for large objects, which cannot be set to a value larger than half the total size of the pool.[8, 9]

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

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

[8] **Oracle Knob** – `SHARED_POOL_SIZE`
[9] **Oracle Knob** – `SHARED_POOL_RESERVED_SIZE`

The next problem is what to do with data collected for a configuration that causes the DBMS to fail during the workload replay. Simply discarding this data and starting the next iteration means that the tuning algorithms would fail to learn that the configuration was bad. But including the metrics from a delayed crash is risky because if they are not scaled correctly then the algorithms could improperly learn that those configurations improve the objective function. Our solution for configurations that cause the DBMS to crash is to set the result for that iteration to be twice the objective function value of the worst configuration ever seen. Because the DBMS is not operational with these failed configurations, it is valid to give them the same "score."

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

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

Additional research is needed on how to best handle maintenance tasks that are scheduled during a tuning session. One approach could be to discount the metrics collected in any iteration where a maintenance task was running. We believe that this is a more viable solution in cases where it is inappropriate to disable the task.

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

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

Chapter 5

Advisory-Level Tuning

The work we have completed thus far in this thesis indicates that one can achieve more efficient tuning of new database workloads by learning from past experiences. In our field study, we evaluated the effectiveness of three ML-based techniques on a real-world workload. One assumption we made in this study 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, such an arrangement is not always possible due to logistical and monetary constraints. For example, an organization may not have a test platform or tools to capture and replay a query trace. The key takeaway is that tuning every database using spare hardware is not always practical.

In this chapter, we consider other less-obtrusive tuning strategies and study how to further exploit the similarity between database workloads. We begin with a discussion of these tuning strategies how they trade tuning quality for tuning effort, followed by a description of the workloads we evaluate in this chapter. We then motivate the need for tuning at the "advisory" level by showing the utility that such methods would have if we could exploit the similarity information between workloads. Next, we discuss the methods that we evaluate for advisory-level tuning. Lastly, we provide an evaluation of these methods.

5.1 Taxonomy

When carrying out tuning activities, it is common practice for the DBA to prepare a copy of the production database and workload and then replay it on a separate test platform. This is especially useful for tuning critical applications where performance degradation or outages can have serious financial consequences. But this process is rarely automated and thus can be costly in terms of the DBA's time, computing resources, and administrative overheads, such as scheduling a time to record the query trace.

Given this, it is important for an automatic tuning service to support alternative tuning strategies that reduce the disruption to the database. These tuning strategies make a trade-off between custom tuning (i.e., tuning tailored to the target DBMS's workload and hardware) and tuning effort (i.e., the amount of effort to perform the tuning). We organize them into three levels, where increasing levels are capable of more custom tuning but also require a greater effort by the DBA.

5.1.1 Level #1 – Advisory

The lowest level passively observes the target database and *advises* a single configuration based on previous tuning experiences. In other words, an automatic tuning service has "one shot" to tune the database correctly. This level requires the least amount tuning effort but does not provide any custom tuning for the target database (i.e., no feedback loop). To support this strategy, an automated tuning service must have previous tuning data to learn from. This strategy is more effective when there is tuning data for a diverse set of workloads since it increases the likelihood that one of them will be similar to the target workload.

5.1.2 Level #2 – Online

The next level tunes a production database in an *online* fashion (i.e., while the database is running in production). This strategy requires less effort than the offline strategy since it tunes the production database directly and not on a testing platform. 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 primary database. Note that this replica is running in production and not a testing environment. That is, the replica is used to provide availability guarantees to the application for the 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. Given this, the degree of custom tuning depends on the quantity and quality of the dynamic knobs supported by the target DBMS.

5.1.3 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 arrangement is what we have assumed in our earlier work in Chapters 3 and 4. This strategy requires the most effort to set up by DBAs but enables them to custom tune all of the knobs since restarts are permitted. This strategy also requires sophisticated tools that are capable of capturing the queries that the application executes on the production DBMS and then replaying those queries multiple times on a test database. These tools also capture meta-data about the workload, such as timing information, to support replaying the queries with the exact timing, concurrency, and transaction characteristics of the original workload. Some commercial DBMSs include such tools to assist with this task (e.g., Oracle's Real Application Toolkit [46]). But the same is not true for open-source DBMSs, and implementing capture and replay tools in-house is a major engineering effort.

Again, all of our work so far in this thesis has focused on offline tuning. For this chapter, we could choose to focus on online tuning, however, the issue with this strategy is that the potential improvement and usefulness of the techniques is DBMS-specific. This is because the degree of custom

	YCSB (R/O)	Twitter	Wikipedia	CH-benCH	TPC-C	YCSB (U/O)
Size	18 GB	20 GB	14 GB	22 GB	20 GB	18 GB
Tables	1	5	12	12	9	1
Columns	11	18	122	106	92	11
Indexes	1	9	40	18	15	1
Txns	1	5	5	8	5	1
Read-only Txns	100.0%	0.9%	92.2%	54.0%	8.0%	0.0%
`SELECT`	100.0%	99.1%	90.8%	51.1%	50.7%	0.0%
`UPDATE`	0.0%	0.0%	4.7%	26.6%	26.8%	100.0%
`INSERT`	0.0%	0.9%	4.5%	20.8%	21.0%	0.0%
`DELETE`	0.0%	0.0%	0.0%	1.4%	1.5%	0.0%
Rows Read	100.0%	99.9%	99.8%	99.5%	72.4%	50.0%
Rows Updated	0.0%	0.0%	0.1%	0.3%	17.7%	50.0%
Rows Inserted	0.0%	0.1%	0.1%	0.2%	9.3%	0.0%
Rows Deleted	0.0%	0.0%	0.0%	0.0%	0.6%	0.0%
Data Read[1]	100.0%	99.6%	56.0%	80.7%	44.3%	35.7%
Data Written[2]	0.0%	0.4%	44.0%	19.3%	55.7%	64.3%

Table 5.1: Workload Characteristics

tuning, and thus, the quality of the configurations produced, depends on the impact of the dynamic knobs exposed by the DBMS. As such, we seek to better understand Level #1 approaches.

5.2 Workloads

We next introduce the workloads from the OLTP-Bench testbed that we use in this chapter [36]. As discussed in Section 5.1.1, when tuning at the advisory level, having a diverse set of workloads to extrapolate performance information about the DBMS from is important because OtterTune can make better recommendations if any of the past workloads are similar to the target workload. An overview of this their characteristics is shown in Table 5.1.

TPC-C: This is the current industry standard for evaluating the performance of OLTP systems [95]. It consists of five transactions with nine tables that simulate an order processing application. We use a database of 200 warehouses (∼20 GB) in each experiment.

YCSB: The Yahoo! Cloud Serving Benchmark (YCSB) [30] 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 10 columns. We create two variants of YCSB with different workload mixtures: **Read-only** (*100% reads*) and **Update-only** (*100% updates*). We use a database with 18m tuples (∼18 GB) for both workloads.

Wikipedia: This OLTP benchmark is derived from the software that runs the popular on-line encyclopedia. The database contains 12 tables and five different transaction types. These transactions correspond to the most common operations in Wikipedia for article and "watchlist" management.

The combination of a complex database schema with large secondary indexes makes this benchmark useful for stress-testing a DBMS. We configured OLTP-Bench to load a database of 60k articles that is ∼14 GB in total size.

CH-benCHmark: This is a complex HTAP benchmark that mixes the order entry processing of TPC-C with the OLAP query suite of TPC-H [29]. CH-benCHmark combines the original TPC-C schema and transactions with an adapted version of the TPC-H queries. We use a database of 200 warehouses (∼22 GB) in each experiment.

Twitter: This OLTP benchmark is inspired by the popular micro-blogging website. It is reflective of important characteristics of Twitter's system, such as heavily skewed many-to-many relationships. The database has five tables and five transactions. The database is ∼20 GB in size.

To better understand the potential performance improvement of each workload over the DBMS's default configuration, we measure the throughput of MySQL v8.0 using the following four knob configurations: (1) MySQL's default configuration, (2) the recommended settings from MySQL's documentation for the two knobs that control the buffer pool size[3] and the redo log file size[4], (3) the configuration generated by MySQL when its dedicated server flag[5] is enabled, and (4) the configuration generated by OtterTune using DDPG when tuning 28 knobs. MySQL added the dedicated server flag in 2018 with the first version of v8; it uses heuristics to configure four knobs based the assumption that the database can use all available system resources. In addition to the buffer pool size and the redo log file size, the flag also sets the knobs that control the number of redo logs[6] and the how data is flushed to disk[7].

The results in Figure 5.1 show that for all workloads, the configurations generated by OtterTune achieve higher throughput than the other configurations. This is expected since the default configurations is based on their minimal hardware requirements for MySQL. The level of improvement for each workload, however, varies substantially. Notably, the YCSB Read-only workload achieves only 10% higher throughput with OtterTune's configuration over the default configuration. The improvement compared to the other configurations is even less. This is because the short transactions in the Read-only workload require minimal configuration, namely, increasing the size of the buffer pool. But other workloads, in particular, those that are write-heavy like TPC-C and YCSB Update-only, benefit from further customization of their knob configuration settings due to their heavier resource usage.

5.3 Motivation

We now motivate the potential benefit of advisory-level tuning by illustrating how the optimal configurations for some workloads work better for others that are more similar. To demonstrate

[3]**MySQL Knob** – `INNODB_BUFFER_POOL_SIZE`
[4]**MySQL Knob** – `INNODB_LOG_FILE_SIZE`
[5]**MySQL Knob** – `INNODB_DEDICATED_SERVER`
[6]**MySQL Knob** – `INNODB_LOG_FILES_IN_GROUP`
[7]**MySQL Knob** – `INNODB_FLUSH_METHOD`

this point, we use OtterTune to optimize the MySQL v8.0 configuration separately for each of the six workloads from Table 5.1. Then for each workload, we measure the DBMS's performance using the optimized configuration selected for the other five workloads.

The results in Figure 5.2 show the throughput measurements for of the workloads when using the optimized configurations from all other workloads. The striped bar in each graph indicates its the optimal configuration for the given workload. Similar to the results in Figure 5.1, we see less variation in the DBMS's performance for the Read-only workload because it does not stress the DBMS as much as the other workloads. Thus, all of the configurations from the other workloads perform nearly as well as its own optimized configuration. For example, in Figure 5.2e, we see that using the Read-only configuration degrades the throughput of TPC-C whereas using the configurations of the other workloads that have modification queries achieves much higher performance.

Our goal is to determine the best methods for advisory tuning and whether they can achieve performance that is comparable to the Level #3 algorithms from the previous chapters where OtterTune tuned the DBMSs offline. We compare two methods in our evaluation: (1) Workload Mapping and (2) Contextual Bandits. The first is an adaptation of OtterTune's workload mapping technique from Section 3.3.1 that we optimized for advisory tuning.

The second method uses side information or "context" to make more informed decisions when choosing the next action to take. We now describe the details of these methods and how OtterTune uses them to tune a DBMS at the advisory level.

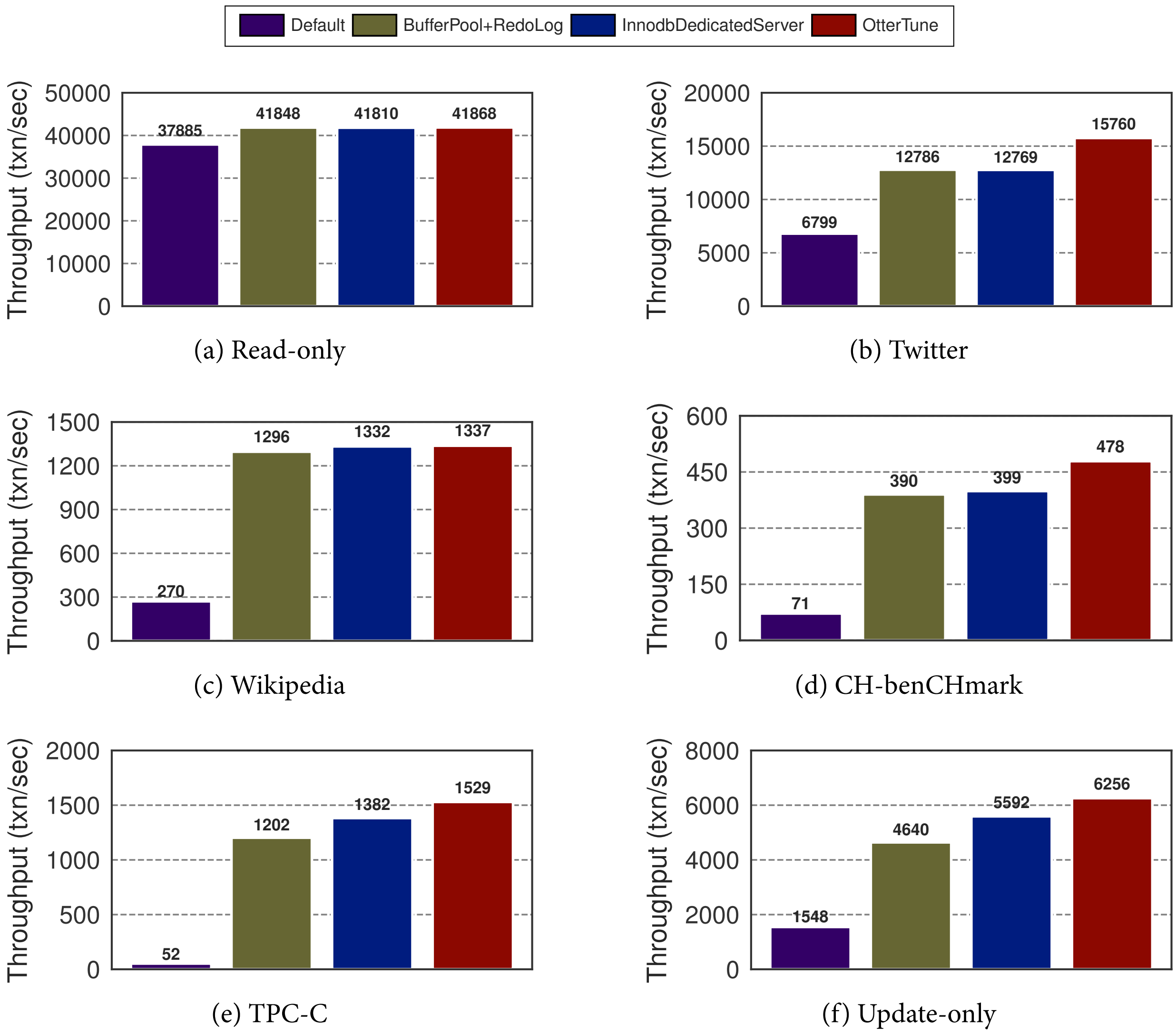
5.4 Workload Mapping

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. Recall from Section 3.3 that workload mapping is the first of two steps in the Automated Tuning stage in OtterTune's ML pipeline. In the final step, OtterTune bootstraps its models with the data from the mapped workload to reduce the time needed to optimize the knob configuration for the target workload.

In the workload mapping step, OtterTune computes the Euclidean distance between the metrics collected so far for the target workload and each past workload. But due to the large number of possible configurations, it is unlikely that any of the configurations attempted by the past workloads are the same as those tried by the target workload. OtterTune handles this by training Gaussian process (GP) models with the past workload data to predict any missing metric values. OtterTune chooses the past workload with the smallest Euclidean distance as the one that is most similar to the target workload. We refer to the most similar workload as the *mapped workload*. OtterTune recomputes the mapped workload every iteration to incorporate new data from the target workload.

We find that with OtterTune's original workload mapping technique, the mapping can be inaccurate for the first 3–4 iterations but then improves quickly. Thus, this does not impact the tuning result when using the online and offline tuning strategies. 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.

We next discuss the two optimizations to OtterTune's original workload mapping technique that improve its effectiveness for advisory-level tuning.



(a) Read-only (b) Twitter (c) Wikipedia (d) CH-benCHmark (e) TPC-C (f) Update-only

Figure 5.1: Workload Tuning Comparison (MySQL v8.0) – Throughput measurements for each workload running on MySQL (v8.0) using the (1) default configuration, (2) buffer pool & redo log configuration, (3) MySQL's dedicated server flag, and (4) OtterTune's configuration.

5.4.1 Optimization #1 — Hyperparameter Tuning

The first optimization increases the accuracy of the models that OtterTune uses to predict the missing metrics for past workloads. For this optimization, we compared the performance of the GP models from the original technique with two alternative approaches. The first uses the same GP models as the original technique except that we tune their hyperparameters.

The second approach is a popular ensemble method called a random forest (RF). RFs train multiple decision trees on sub-samples of the dataset and aggregate their individual predictions to improve accuracy and control over-fitting [19]. A limitation of tree-based methods such as RFs is that they cannot extrapolate to unseen data (i.e., data points that lie outside the range of the training

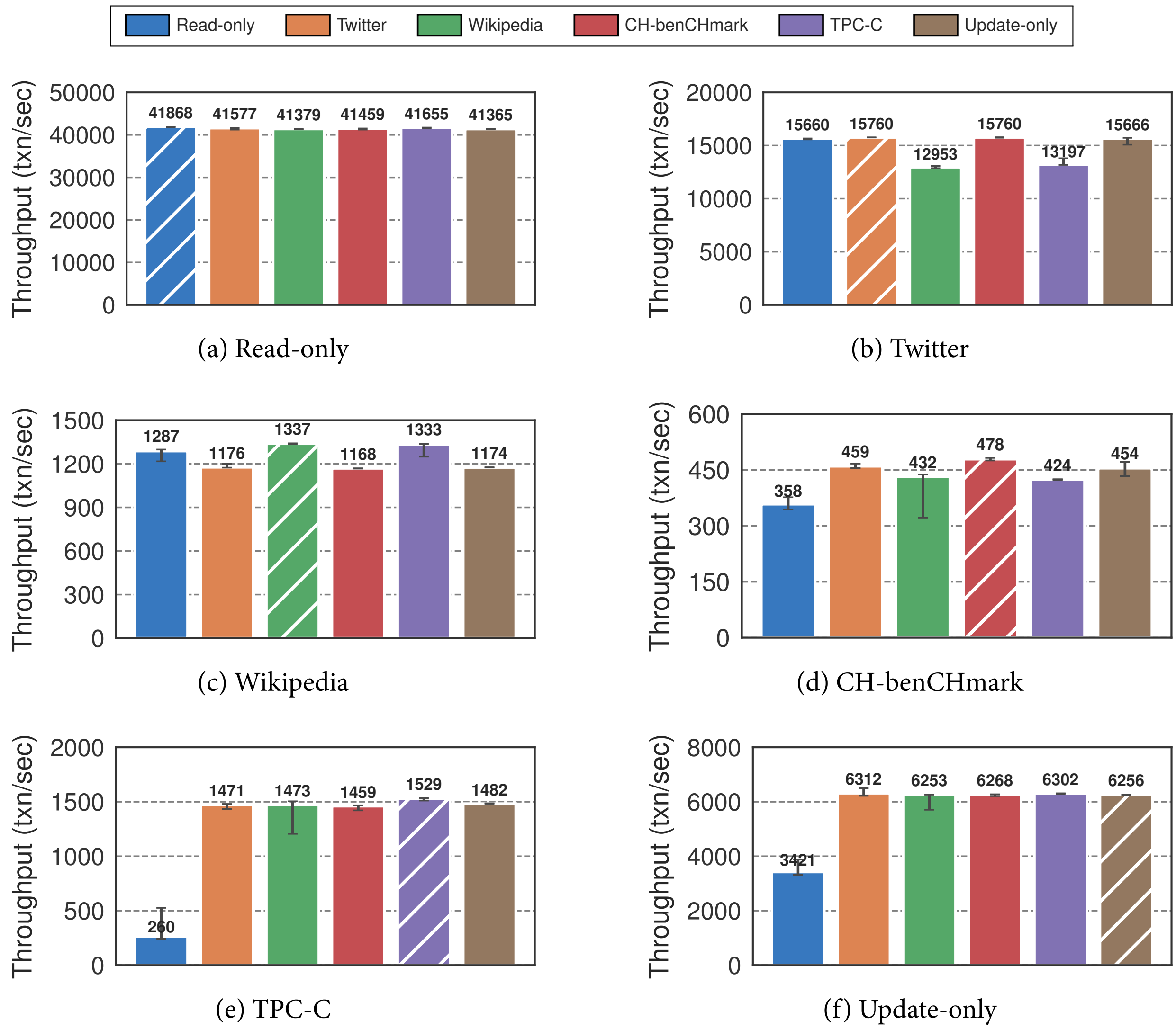


Figure 5.2: Swapping Optimized Configurations (MySQL v8.0) – Throughput measurements for the workloads when using the optimized configurations from all other workloads. The striped bar for each workload indicates its the optimal configuration for that workload.

data). This was not an issue in our experiments because the workload data we collected included samples near the ends of each knob's allowable range.

We trained separate models for each approach: one for the workload and one for the metric. For the new GP and RF approaches, we tuned their models' hyperparameters using a randomized search that sampled 50 settings. We ran cross validation and used the root mean squared error (RMSE) to measure the accuracy of the regression models. As expected, both the optimized GP and RF models outperformed the unoptimized GP models used in the original workload mapping technique. The optimized GP models achieved the best accuracy of all. As such, we improve OtterTune's original technique by optimizing the hyperparameters of the GP models.

We examined the validation results and found that the RF models' predictions were inaccurate when estimating metrics for regions in the configuration space that degrade performance. This is possibly due to the lack of the training data around these regions since OtterTune's recommendation algorithms avoid them. One configuration where the RF models' predictions are most inaccurate is with MySQL's default settings (i.e., the configuration that is installed with the MySQL's installation package). This is the first configuration tried at the start of a tuning session, but it almost always has the worst performance because the knobs' settings are based on MySQL's conservative minimum hardware requirements.

5.4.2 Optimization #2 — Static Metrics

Our second optimization to OtterTune's original workload mapping technique improves the quality of the distance calculations between the target workload and each past workload. We refer to these calculations as the *workload mapping result*.

OtterTune calculates the distances between workloads using the pruned metrics output by the Workload Characterization stage in the ML pipeline (see Section 3.1). Pruning the hundreds of DBMS metrics is necessary for some tasks in OtterTune's ML pipeline, such as predicting missing metric values, that would otherwise be infeasible to compute. OtterTune uses unsupervised learning techniques to prune the metrics. These techniques generally produce good results but the pruned metrics they choose can vary between invocations. The impact of these variations is negligible for online/offline tuning but can be significant at the advisory level.

We found that including static metrics about the workload, such as the read/write ratio or number of indexes (see Table 5.1), in the distance calculation helps to stabilize the workload mapping result. Using static metrics also improves the accuracy of the result, especially in cases where there are "invalid" pruned metrics, for example, due to bad predictions or noisy data.

After computing the mapped workload, OtterTune then begins the configuration recommendation step. OtterTune performs this step as described Section 3.3.2 usual to recommend a configuration using the data from the mapped workload and the single data point from the target workload. For advisory tuning, a simpler option is to recommend the optimized configuration for the mapped workload. This option may perform as well given the limited data from the target database.

5.5 Contextual Bandits

The second advisory-level method we evaluate is called *contextual bandits* (CB). CB is where, in each iteration, the algorithm receives context (i.e., side information about the environment or task), and then chooses an action from a set of possible actions to maximize the total payoff (i.e., reward) of the selected actions [58, 69]. The payoff depends on both the action chosen and the context. In contrast, context-free bandit problems, such as our configuration recommendation algorithm from Section 3.3.2, model situations where no side information is available and the payoff depends only on the action chosen. The main challenge of CB is how to efficiently trade off exploration (i.e., collecting data to understand the payoff function over the context-action space), and exploitation (i.e., choosing an action believed to be optimal based on existing data) [16].

CB solutions have been successfully deployed in areas like healthcare, recommendation systems, information retrieval [18, 66, 71, 94]. Recent efforts have focused on applying CB to web personalization tasks, such as ad and news article placement. The benefit of this is that OtterTune can build a single model with the data from all previous workloads. The difference is that OtterTune must also include context about the workloads (e.g., the pruned metrics) when training the CB model. With CB, OtterTune can skip workload mapping step and directly recommend a configuration. When selecting the next configuration, the CB model also considers the context of the target workload when making its decision.

Despite these advantages, successful applications of CB in practice have only been for discrete action spaces [72]. These are tasks with a finite number of actions to choose from (e.g., select the best of 10 ads to show to the user). But in knob configuration tuning and other practical tasks, the action chosen is continuous. Developing CB algorithms that can efficiently choose an action from a continuous space is challenging. As such, CB algorithms for continuous action spaces are less studied than their discrete counterparts.

We implemented two CB algorithms by extending OtterTune's GPR and DNN algorithms. We added an optional feature vector that specifies the context. In our evaluation, the the workload context we use for the CB models is the same workload information used by the workload mapping technique, namely, the set of pruned and static metrics.

5.6 Evaluation

We now present the results from our comparison of the above advisory-level tuning methods for the six workloads introduced in Section 5.2.

We use MySQL v8.0 as the target DBMS in our evaluation. We conducted all of our experiments on local machines with a Intel Core i7-8650U CPU (eight cores with 2× hyperthreading), 32 GB RAM, and a Samsung 960 EVO PCIe NVMe SSD storage. We ran each experiment runs on two machines. The first is used to run OtterTune's controller that we integrated with the OLTP-Bench framework. The second machine is for the target DBMS deployment, which we ran from a Docker container using the official MySQL v8.0 Docker image. We deployed OtterTune's tuning manager and repository on a local server with 20 cores and 128 GB RAM.

Each tuning session consists of 300 iterations and tunes 28 of MySQL's knobs that we selected based on performance tuning articles and those used in previous studies on automatic DBMS configuration tuning [103, 109]. For each tuning session, we set OtterTune's observation period (i.e., workload execution time) per iteration to five minutes and assign the target metric to be the throughput as measured by OLTP-Bench.

5.6.1 Workload Mapping

We begin with an initial evaluation of OtterTune's workload mapping technique. As discussed in Section 5.4, we train models to predict the missing metrics for each workload and thus using fewer metrics reduces the model training time.

In this experiment, we compare the workload mapping result when using (1) all of the DBMS's runtime metrics, (2) OtterTune's original workload mapping technique from Section 3.3.1, and (3) OtterTune's optimized workload mapping technique. The goal is to show that OtterTune's optimized technique improves the accuracy and stability of the original technique and produces a workload mapping result comparable to using all runtime metrics.

Although MySQL has hundreds of runtime metrics, several of them are never updated and have the same value for all of the workloads in our evaluation. After removing these, the total set of "active" metrics that we consider in this evaluation is reduced to 86 metrics. The results in Table 5.2 show the mapping results for each workload when using all 86 metrics. For each workload, we display the distance between it and all other workloads, where a smaller distance indicates that the algorithm believes the workloads are more similar. We also include the distance of the workload to itself in gray. This is useful for determining the quality of the mapping since the most similar workload should be to itself.

The results in Table 5.2 show that the relative ordering of workloads is as we expect for each workload. Specifically, the distance from each workload to itself is much smaller than the distance to the next most similar workload. Furthermore, the next most similar workloads that each workload is mapped to all appear to be reasonable in that they map to another workload that has a similar read/write ratio. For example, the Read-only workload maps to Twitter workload because it is also read-heavy. Likewise, the TPC-C workload maps to the Update-only workload because both of them are write-heavy.

Table 5.3 shows the results from our original workload mapping technique that uses a set of eight pruned metrics. There are several issues with the results. The first is that the Wikipedia workload is not the most similar to itself (i.e., its distance is not the smallest). This suggests that the mapping is less accurate using these eight pruned metrics than when using all metrics. The second is that for the Update-only workload, the two most similar workloads (other than itself) are read-heavy, which contradicts what we knew about the nature of these workloads. In particular, the distances between the Update-only workload and the next three similar workloads are small, which indicates that this result is unstable.

Table 5.4 shows the results from using our optimized workload mapping technique that uses the same eight pruned metrics as the original technique as well as eight static metrics. The static metrics include the query mixtures (i.e., the percentages of `SELECT`, `INSERT`, `UPDATE`, and `DELETE` queries), percentage of read-only queries, and characteristics of the schema (i.e., the number of tables, indexes per table, and columns per table). The larger distances between the workloads indicate a more stable result. The results also show that the relative orderings of the workloads are comparable to those calculated when using all metrics.

5.6.2 One-Shot — Workload Models

After the workload mapping step, we next investigate what knob configuration to recommend for the target workload. We consider two options. The first is to directly install the optimized configuration from the most similar workload on the target database. The second option is to train models on the data from the most similar workload and the new single data point for the target

	Read-only		Twitter		Wikipedia	
Rank	Workload	Distance	Workload	Distance	Workload	Distance
-	Read-only	14.63	Twitter	13.27	Wikipedia	6.93
1	Twitter	40.37	Read-only	36.35	Update-only	36.34
2	Wikipedia	43.45	Wikipedia	40.68	TPC-C	37.30
3	CH-benCH	46.39	TPC-C	41.26	CH-benCH	39.67
4	Update-only	46.43	CH-benCH	45.78	Read-only	42.95
5	TPC-C	46.69	Update-only	47.86	Twitter	49.84
	CH-benCH		**TPC-C**		**Update-only**	
Rank	Workload	Distance	Workload	Distance	Workload	Distance
-	CH-benCH	19.65	TPC-C	10.30	Update-only	15.39
1	TPC-C	29.46	Update-only	47.91	TPC-C	37.07
2	Wikipedia	41.83	Wikipedia	48.69	Read-only	37.47
3	Update-only	45.35	Twitter	49.34	Twitter	42.00
4	Read-only	45.63	Read-only	50.09	Wikipedia	43.74
5	Twitter	48.80	CH-benCH	58.00	CH-benCH	43.77

Table 5.2: Workload Mapping (all metrics) – The distance measurements between workloads computed by the original workload mapping technique using all 86 pruned metrics. A smaller distance indicates the workload is more similar. The distance between a given workload and itself is shown in gray.

	Read-only		Twitter		Wikipedia	
Rank	Workload	Distance	Workload	Distance	Workload	Distance
1	Read-only	7.00	Twitter	6.00	Update-only	7.62
2	Twitter	8.19	Read-only	8.89	CH-benCH	8.94
3	Update-only	12.65	Update-only	11.09	TPC-C	9.00
4	TPC-C	13.27	TPC-C	12.00	Wikipedia	10.30
5	CH-benCH	14.76	CH-benCH	12.77	Twitter	13.67
6	Wikipedia	16.37	Wikipedia	15.52	Read-only	14.14
	CH-benCH		**TPC-C**		**Update-only**	
Rank	Workload	Distance	Workload	Distance	Workload	Distance
1	CH-benCH	6.00	TPC-C	1.41	Update-only	5.92
2	Update-only	6.78	Update-only	1.73	Read-only	6.71
3	TPC-C	7.87	CH-benCH	3.00	Twitter	6.77
4	Twitter	8.31	Wikipedia	10.44	TPC-C	6.78
5	Read-only	10.30	Twitter	10.68	CH-benCH	9.33
6	Wikipedia	13.42	Read-only	11.96	Wikipedia	14.32

Table 5.3: Workload Mapping (eight pruned metrics) – The distance measurements between workloads computed by the original workload mapping technique using eight pruned metrics. A smaller distance indicates the workload is more similar. The distance between a given workload and itself is shown in gray.

workload (i.e., the set of knobs and metrics collected after observing the target workload with its current configuration). Recall from Section 5.1 that we refer to the latter option as "one shot" since the models have only one observation to make a good recommendation. We now compare these two options in our experiments.

	Read-only		Twitter		Wikipedia	
Rank	Workload	Distance	Workload	Distance	Workload	Distance
1	Read-only	1.41	Twitter	2.24	Wikipedia	2.65
2	Twitter	12.88	Read-only	14.66	CH-benCH	11.79
3	CH-benCH	18.68	CH-benCH	15.68	TPC-C	14.83
4	Wikipedia	19.95	Wikipedia	16.12	Twitter	15.84
5	Update-only	21.03	TPC-C	19.16	Update-only	17.49
6	TPC-C	22.96	Update-only	19.85	Read-only	18.47

	CH-benCH		TPC-C		Update-only	
Rank	Workload	Distance	Workload	Distance	Workload	Distance
1	CH-benCH	2.82	TPC-C	0.00	Update-only	1.73
2	TPC-C	8.19	CH-benCH	10.91	TPC-C	15.52
3	Twitter	14.70	Update-only	16.19	Twitter	16.97
4	Wikipedia	16.52	Twitter	17.52	CH-benCH	17.15
5	Update-only	17.52	Wikipedia	20.93	Read-only	17.92
6	Read-only	17.65	Read-only	21.28	Wikipedia	22.10

Table 5.4: Workload Mapping (eight pruned metrics + eight static metrics) – The distance measurements between workloads computed by the optimized workload mapping technique using eight pruned metrics and eight static metrics. A smaller distance indicates the workload is more similar. The distance between a given workload and itself is shown in gray.

Figure 5.3 compares the throughput for each of the six workloads we evaluate using four different configurations: (1) the optimized configuration from the mapped (i.e., most similar) workload, (2) the configuration recommended by DDPG, (3) the configuration recommended by DNN, and (4) the configuration recommended by GPR. The yellow line indicates the performance of the optimized configuration for the target workload. In the captions, the workload displayed inside the parenthesis indicates the most similar workload determined by the improved mapping technique from Table 5.4.

From the results in Figure 5.3, we again see that the best configurations for each workload are those based on other workloads with similar read/write ratios. For example, the best configurations for Twitter are from the Read-only workload, and the best configurations for TPC-C are from the CH-benCHmark and Update-only workloads. We also see from the results that the best configuration between the two options is the first: to directly use the optimized configuration from the most similar workload as opposed to the one-shot approach.

5.6.3 One-Shot — CB Algorithms

As discussed in Section 5.5, an alternative approach to workload mapping is to train ML models that includes the context of the workload in the model itself. The benefit of such an approach is that we must only train a single model from all of the workload data, which reduces the training time when there are several past workloads to consider. The disadvantage is that, due to the increased dimensionality in the number of features, these models are more complex to optimize, which makes it more difficult to achieve good recommendation results.

In this final experiment, we compare the efficacy of the CB algorithms from Section 5.5 with the workload mapping technique described in the previous section. In addition, we include the

DDPG algorithm from Section 4.3.2 in our comparison because it also uses contextual information to inform its decisions. We use the same 16 metrics (eight pruned metrics and eight static metrics) used by the workload mapping technique as the context since these metrics are emblematic of the workload.

Figure 5.4 shows the throughput for each of the six workloads when using the configurations recommended by the CB models. The yellow line displays the performance of each workload's own optimized configuration. We see from the results that of the contextual algorithms, DDPG performs the best overall, followed by GPR. This is possibly because of the replay memory that DDPG uses compared to the other contextual models (see Section 4.3.2).

Comparing these results with Figure 5.3, however, also shows that mapping to the optimized configuration from the most similar workload achieves comparable or better performance than the GPR and DNN contextual models. From these results, we conclude that the simplest method is the best for MySQL with this particular set of six workloads.

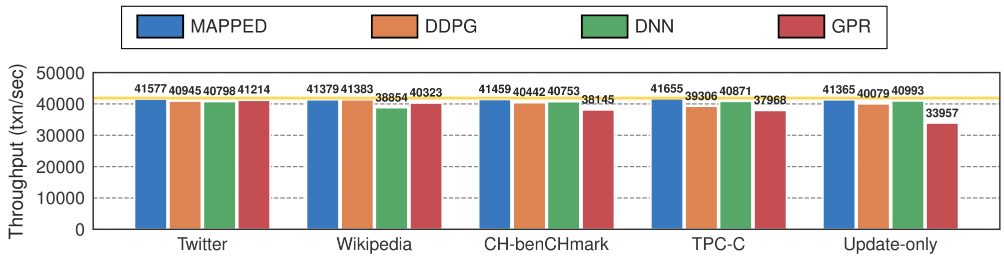
5.7 Lessons Learned

This chapter explored less-obtrusive tuning strategies and presented methods to exploit the similarity between database workloads. Again, the goal here is to evaluate "advisory" tuning methods with OtterTune that do not require either (1) multiple iterations with DBMS running in production or (2) a cloned DBMS workload for offline tuning. We now discuss some key takeaways from our experiments.

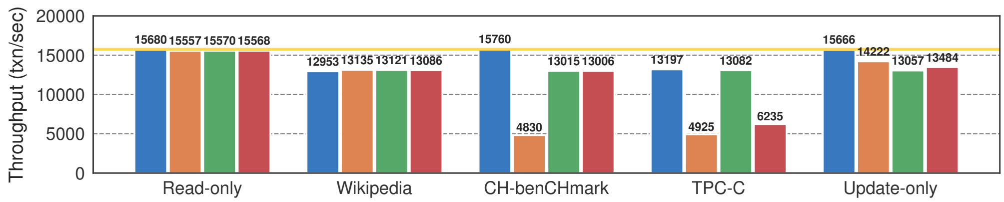
The first lesson that we learned was that the most straightforward strategy performed the best for MySQL in our evaluation. The DBMS achieved the best performance under advisory tuning when OtterTune used the enhanced workload mapping technique and then selected the mapped workload's optimized configuration. This method worked well in our evaluation because the workloads were stable. The mixture of read versus write queries and the complexity of those queries were the same during each iteration. Prior research has shown that there are a large number of applications with similar stable workload patterns [70]. Investigating how changes in the volume of the workload (i.e., the number of queries that the application executes) will affect the generated configuration remains future work.

But even if an application's workload composition shifts (e.g., executing write-heavy OLTP queries during the day and then read-only OLAP queries at night), we believe that the above advisory tuning method would still be sufficient because it is easy to deploy. A DBA could configure a service like OtterTune to run periodically during the day in a Level #1 advisory mode and determine whether the workload has changed enough to merit a change to the DBMS's configuration. Of course, this assumes that the updated configuration does not require the DBMS to restart or that restarting the DBMS is allowed. The latter is a value judgment that has to be made by humans since OtterTune's algorithms cannot determine how a restart will affect the application.

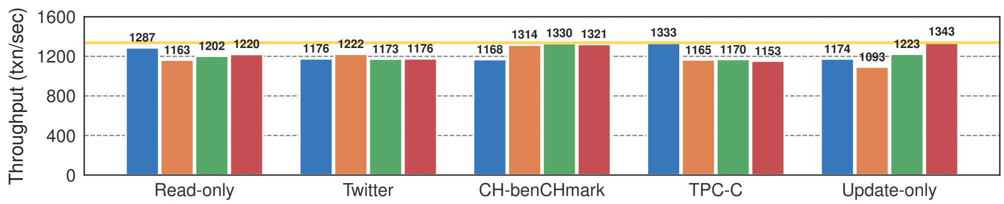
There are also active learning methods that we expect may outperform the improved workload mapping technique from Section 5.4. For example, OtterTune could use Gaussian Coppulas to "normalize" the distributions of the workloads. This method could potentially allow the algorithms to combine the data across disparate data sets.



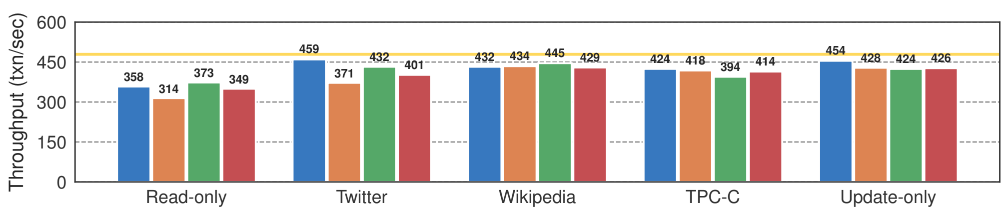
(a) Read-only (Twitter)



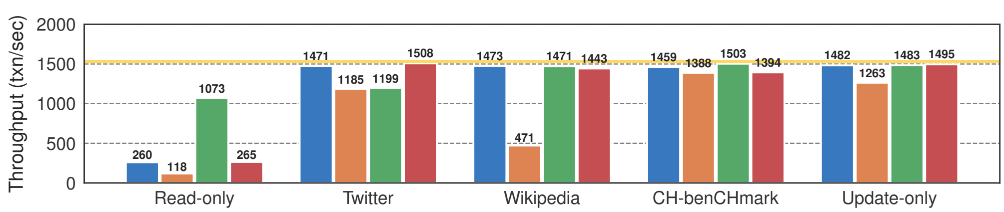
(b) Twitter (Read-only)



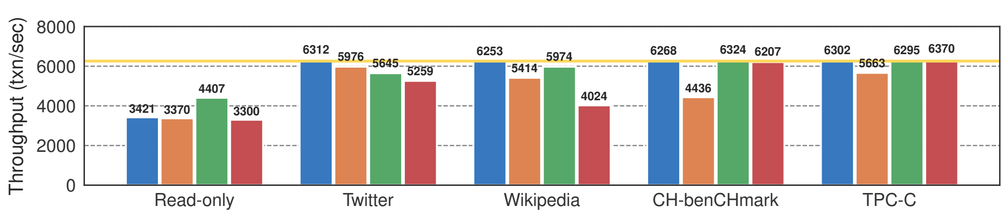
(c) Wikipedia (CH-benCHmark)



(d) CH-benCHmark (TPC-C)



(e) TPC-C (CH-benCHmark)



(f) Update-only (TPC-C)

Figure 5.3: One-Shot Configurations – Configurations recommended by models trained on data from the most similar past workload determined in the workload mapping step.

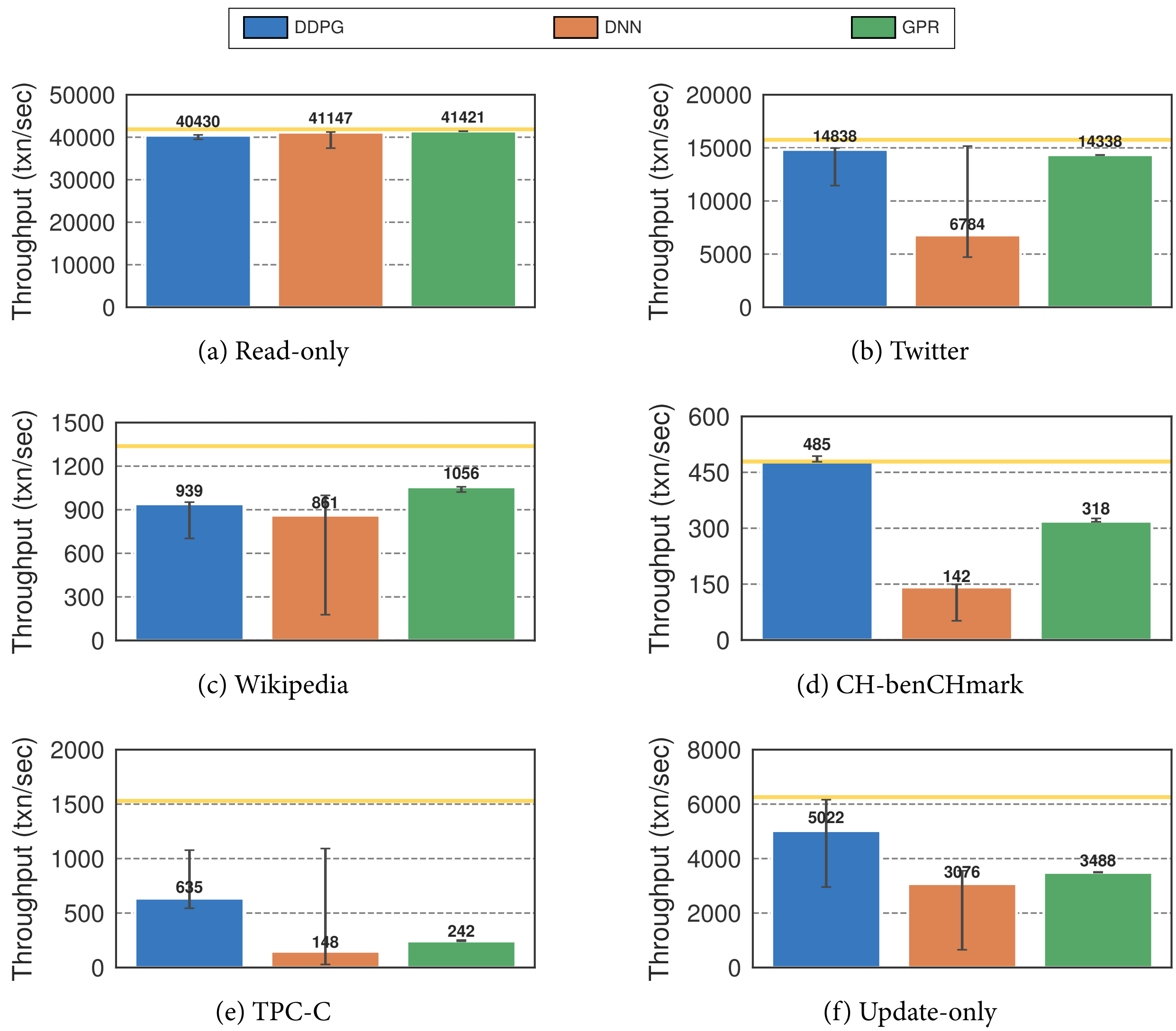


Figure 5.4: CB One-Shot Configurations – Configurations recommended by CB models trained on all previous workload data with 16 DBMS runtime metrics as the workload context.

Chapter 6

Related Work

The need for tools that can automatically optimize complex systems and applications has been well-known for decades [15, 28]. As such, there have been a number of efforts to automate the tuning of DBMSs since the 1970s [27]. Much of the previous work has focused on optimizing the physical design of the database. In the early 2000s, researchers began to explore automatic knob configuration because optimizing a DBMS to meet the needs of an application is highly dependent on a number of factors that are beyond what humans can reason about. There has also been research on configuration tuning for data analytics systems. We now discuss the details of previous work.

6.1 Physical Database Design

The physical design problem for DBMSs can be described as the following: given an arbitrary database and its target workload (e.g., queries, transactions), select the best physical design of that database that optimizes one or more *metrics* in the DBMS for the workload and satisfies *budget constraints*. Such a physical design can include indexes, table partitioning, table replication, and materialized views. The application's target metrics are the desired performance outcomes of the DBMS when executing the workload, such as to maximize its throughput or to minimize its latency. The budget constraints are defined in terms of storage limits (e.g., the amount of memory to allocate for indexes) or CPU overhead (e.g., the time it takes to update an index). The most notable advancements for this research area are from two commercial database vendors: Microsoft's SQL Server AutoAdmin [10, 11, 25, 26, 74] and IBM's DB2 Database Advisor [80, 112].

These physical design algorithms are guided by different cost models that are tailored to the type of workload that the database needs to support. In the 1970s and 1980s, these tools used theoretical cost models that resided outside of the DBMS to estimate the gains (if any) of one particular design decision [42, 44, 47, 48, 54]. In the 1990s, Microsoft's AutoAdmin tool pioneered the use of leveraging the DBMS's built-in cost models from its query optimizer to estimate the performance benefits of indexes [26]. This allowed them to avoid the disconnect between what the external models chooses as a good index and what the system actually uses at runtime when it generates query plans.

Regardless of what search algorithm or cost model a design tool uses, the administrator must decide which metrics will guide the design decision process. For large-scale OLAP applications [75, 84], the tool should seek to generate designs that maximize the spread of data across multiple locations to maximize intra-query parallelism [11, 24, 68, 80, 111]. This is not applicable to OLTP applications, however, because the coordination required to achieve transaction consistency across these locations dominates the performance gains obtained by this type of parallelism [49]. By contrast, OLTP workloads are comprised of short-running transactions that need to access a small amount of data quickly, thus these applications need designs that minimize the lock contention of transactions [31, 77].

6.2 Configuration Tuning for Databases

Unlike physical database design tools, configuration tools cannot use the built-in cost models of DBMSs' query optimizers. This is because these models generate estimates on the amount of work that the system is expected to perform for a particular query. These estimates are intended to compare alternative query execution strategies for a single DBMS with a fixed execution environment [86]. The configuration tools that are available are limited in scope and are mostly restricted to commercial systems. All of the major DBMS vendors have their own proprietary tools that vary in the amount of automation that they support [35, 60, 73]. They are primarily focused on configuring the memory allocated to buffer pools, as these are one of the most important targets for tuning a DBMS [45].

In the early 2000s, IBM released the DB2 Performance Wizard tool that asks the administrator questions about their application (e.g., whether the workload is OLTP or OLAP) and then provides knob settings based on their answers [63]. It uses models manually created by DB2 engineers and thus may not accurately reflect the actual workload or operating environment. IBM later released a version of DB2 with a self-tuning memory manager that uses runtime heuristics to determine how to allocate the DBMS's memory to its internal components [90, 97].

Oracle developed a similar internal monitoring system for their DBMS to identify bottlenecks due to misconfiguration in the system's internal components [35, 60]. It then provides the administrator with actionable recommendations to alleviate them. Like IBM's tool, the Oracle system employs heuristics based on performance measurements to manage memory allocation and thus is not able to tune all possible knobs. Later versions of Oracle include a SQL analyzer tool that estimates the impact on performance from making modifications to the DBMS, such as upgrading to a newer version or changing the system's configuration [13, 106]. This approach has also been used with Microsoft's SQL Server [73]. But for both DBMSs, using this tool is still a manual process: the administrator provides the knob settings that they want to change and then the tool executes experiments with and without applying that change. The administrator then decides what action to take based on the results that the tool reports.

More automated feedback-driven techniques have been used to iteratively adjust DBMS configuration knobs to maximize certain objectives [21, 37, 104]. These tools typically contain an experiment "runner" that executes a workload sample or benchmark in the DBMS to retrieve performance data. Based on this data, the tool then applies a change to the DBMS configuration and then re-executes that workload again to determine whether the performance improves [105]. This continues until the administrator either halts the process or the tool recognizes that additional performance gains from running more experiments are unlikely. This same approach has also been used for automatic tuning in operating systems [20, 41].

The COMFORT tool uses this on-line feedback approach to solve individual tuning issues like load control for locking and dynamic data placement [104]. It uses a technique from control theory that can adjust a single knob up or down at a time. It could not uncover dependencies between multiple knobs.

The work by Sullivan et al. uses influence diagrams to model probabilistic dependencies between configuration knobs [92]. This approach uses the knobs' conditional independences to infer expected outcomes of a particular configuration setting in BerkeleyDB. The problem, however, is that these influence diagrams must be created manually by a domain expert. Thus, this approach is unlikely to scale for more complex problems. This is evident from their experimental evaluation where they only consider four knobs to tune.

An alternative technique is to use linear and quadratic regression models to map knobs to performance [102]. With this approach, the experiment data is fit with an equation. Others have looked at using hillclimbing techniques for this problem [105]. This work, however, does not address how to retrain these models using new data or how to guide the experiment process to learn about new areas in the solution space.

The BestConfig tool employs a search-based technique that recursively reduces the solution space using heuristics [110]. It uses a technique similar to Latin Hypercube Sampling to divides the domain of each knob into subspaces and then samples in each subspace. Starting with an initial set of samples, it iteratively bounds the space around the best configuration from the latest set of samples before executing the next set. To avoids getting stuck in a sub-optimal subspace, it restarts the search if none of its current samples outperform the best one from the previous set. The problem is that this approach cannot learn from the data it collects and may end up trying similar configurations after restarting. It can also take much longer to optimize due restarts, especially in high-dimensional spaces since restarts are more likely.

The iTuned tool is the closest work that is related to our Gaussian process regression technique in Chapter 3 [37]. Its experiment runner continuously measures the impact of changing certain knobs on the system's performance using a "cycle stealing" strategy that makes minor changes to the DBMS configuration and then executes a workload sample whenever the DBMS is not fully utilized. It uses a Gaussian process model to represent the response surface and an adaptive sampling technique to explore the solution space and converge to a near-optimal configuration. The tool constructs unique models for every application by running new experiments. It bootstraps these models by executing an initial set of experiments generated using a sampling technique called Latin Hypercube Sampling [52]. The key difference between our technique and iTuned is that we leverage the knowledge gained from previous tunings to inform the decisions about future deployments. The iTuned tool can take up to seven hours to tune the DBMSs, whereas our results in Section 3.4.4 show that our technique achieves this in less than 60 min.

Deep reinforcement learning (DRL) is another ML-based method that has been adapted for database configuration tuning. The CDBTune tool uses a policy-based method called DDPG that is based on the popular Q-learning algorithm but can operate over continuous action spaces [109]. Its reward function is designed to model the judgment of a human DBA with the goal of achieving better performance than the initial configuration. The reward considers the change in performance from both the previous configuration and the initial configuration to account for configurations that perform worse than the previous configuration but better than the initial configuration. It also improves the convergence speed for offline training using a method called priority experience replay. The DDPG algorithm we evaluated in Section 4.3.2 is based on the CDBTune tool.

In DDPG, the agent considers only the database state when tuning a database and cannot utilize query information. The QTune tool uses a double-state DDPG (DS-DDPG) model, which can embed query information and consider the effects of both the database state and the workload [65]. It adds a featurization step to transform SQL queries to vectors used by its models. It supports tuning individual queries and cluster-level tuning, which groups queries based on their best knob configurations and then tunes groups of queries. Although QTune supports more fine-grained tuning, privacy constraints may prevent some organizations from sharing this query information since the queries contain sensitive user data.

Alibaba developed the iBTune tool that uses deep learning to optimize the memory allocation of individual database instances by adjusting buffer pool sizes dynamically according to the miss ratio [93]. It uses a technique based on a large deviation analysis for LRU caching models to compute the target buffer pool size for a given instance according to the tolerable miss ratios obtained from similar instances. It then predicts an upper bound for the response time due to adjusting the buffer pool size by employing a pairwise DNN over pairs of similar instances. The tool performs the final adjustment only if the upper bound does not violate the SLA of response times.

All of these feedback-driven tools must decide which configuration knobs to tune. Tuning fewer knobs reduces the optimization time since the search space grows exponentially with the number of knobs. Our results in Figure 3.3 show that only a small subset of these knobs actually affect the performance of MySQL and Postgres for the TPC-C workload. Other recent work has reported similar results for Cassandra and Postgres using YCSB [57]. These findings further motivates the need for methods that can automatically detect the most important knobs to tune. The SARD tool generates a relative ranking of a DBMS's knobs based on their impact on performance using a well-known technique called the *Plackett-Burman design* [34]. Others have developed statistical techniques for inferring from these experiments how to discretize the potential values for knobs [92]. The Plackett-Burman design of experiments has also been applied to evaluating the impact of knobs in the context of storage benchmarks [76] and file systems [22].

Kanellis et al. uses random forest and classification and regression trees (CART) methods to determine the importance of knobs based on their reduction in variance of the target objective [57]. Random forests are able to capture non-linear relationships among variables naturally. Thus, they do not need to include interaction terms in the input features as with our lasso technique for ranking knobs, which is based on linear regression. But random forests lack the interpretability of linear models and, unlike lasso, are unable to automate feature selection since they cannot remove features.

6.3 Configuration Tuning for Data Analytics Systems

Configuration tuning for data analytics systems, such as Spark and Hadoop, is an active area related to database tuning. Like databases, these systems also have hundreds of configuration knobs to control their runtime operation.

RelM is a multi-level algorithm for automatically tuning the memory allocation of data analytics systems [61]. It first uses Guided Bayesian Optimization to derive additional metrics that help distinguish the most suitable region of the configuration space. The exploration is guided by a "white box" model that has knowledge about application memory requirements. RelM uses a DDPG model and includes these metrics in the database state to provide extra visibility into the internal memory pools it tunes.

Starfish is a self-tuning system built for Hadoop that hierarchically optimizes the performance of clusters at the job, workload, and workflow levels [51]. For job-level tuning, it uses dynamic profiling to capture the runtime behavior of jobs and tunes and tunes their knobs based on estimated resource consumption (e.g., CPU, memory). For workflow-level tuning, it uses a scheduler that optimizes workflows by addressing issues such as minimizing the impact of unbalanced data layout. At the workload-level, it uses the Elastisizer to automates provision decisions using simulation and model-based estimation techniques to address what-if questions on workload performance.

MRTuner is a toolkit developed by IBM to enable holistic optimization for MapReduce jobs [85]. It uses Producer-Transporter-Consumer (PTC) cost model that characterizes the tradeoffs in the parallel execution. It uses four key factors from PTC to derive the optimal values and bounds for other important parameters, and employs a fast search method to reduce to find the optimal execution plan

Chapter 7

Future Work

The use of ML methods to automatically optimize DBMS knob configurations with a service like OtterTune is still an early field. As such, there are several open questions that we believe are worth exploring beyond the topics covered in this thesis. We now present some of these future research directions that either apply our work to other problem domains or extend it for tuning DBMSs.

Increased Automation: There are a number of tasks in the setup and deployment of an automatic DBMS tuning service that are still performed manually. For example, in all of our experiments in this thesis, we have to set the proper minimum and maximum values allowed for knobs. This is because these ranges can change per hardware configuration. Most DBMSs set their default allowable ranges for knobs that control memory allocations and file sizes to unreasonably large values. For example, both Postgres and MySQL use the maximum amount of memory addressable in a 64-bit CPU as the default maximum value for some of these knobs. But in an automated tuning service, it is important to limit these ranges to appropriate values for the target database's hardware before initiating a tuning session. Failing to do this can cause the algorithms to select values will cause the DBMS to not start or crash while executing the workload. Thus, one important area of further research that is needed is on how to automatically determine the proper ranges for each new database with requiring an administrator to provide them. One approach is to automatically set the allowable knob ranges using heuristics based on the hardware resources of the DBMS. But this approach requires rules for setting the range of each knob.

Another manual task in the current version of OtterTune is how to determine which knobs are dangerous to tune. For example, which knobs could cause the DBMS to potentially lose data when the DBMS crashes because changes were not safely written to disk before a transaction committed. In other knobs, there may be specific values that can put the DBMS in a dangerous configuration. As we described in Section 2.6, the OtterTune service uses a hand-curated black-list of knobs for each DBMS version that it supports. But for every new version, we have to manually inspect the DBMS's documentation to make sure there are no changes to existing knobs or the introduction of new knobs that are potentially problematic. This tedious step requires a human that is knowledgable in a DBMS's internals to make a decision about whether a knob belongs on OtterTune's blacklist. Given this, we contend that more work is needed to automatically determine whether changing a knob will cause the application to potentially lose data. We envision that this would require a sandbox environment that programmaticaly crashes the DBMS and determines under what configuration settings does the DBMS lose data.

Knowledge Transfer Across DBMS Versions: Another area that we plan to investigate is how to enable OtterTune to reuse information across minor versions of the same DBMS in its models. Currently, OtterTune can only use models that are generated for the same DBMS version. This is because different versions may introduce new knobs/metrics (or deprecate existing ones), and thus it may select not be able to reuse its training data. We believe that we will have to perform an additional localized optimization pass using stochastic gradient descent to choose the best configuration for the target DBMS version since the configurations for each component are not independent from each other. This means that OtterTune will not be to able to completely reuse previous tuning data like it did before. We believe that more advanced methods for OtterTune's ML pipeline, such as knowledge transfer, will allow the service to extrapolate information collected from disparate databases.

More Complex Knob Configurations: Another research area that is worth exploring is how to extend our algorithms to support more complex knob configurations. Right now, OtterTune only handles "global" knobs that affect the overall operation of the DBMS (e.g., the buffer pool size for the entire DBMS). Some DBMSs, however, provide knobs for tuning individual components of the database. For example, Facebook's MyRocks supports tuning the size of the in-memory cache for each individual table [6]. Supporting the optimization of such individualized knobs is challenging since there is no longer a one-to-one mapping between the knobs from one database instance to another. Each application will have a different number of database components (i.e., tables, indexes) and the correct way to tune each of them is highly dependent on the part of the workload that accesses that component. In other words, we will want to tune the cache size differently for tables that are read-only versus update-heavy. One approach could be to add an additional stage to OtterTune's ML pipeline that will treat each database component as a separate workload and use contextual bandits [58] identify a good set of policies for tuning database components that exhibit similar workload characteristics.

White-box Tuning Methods: White-box methods are another interesting direction for future work on DBMS configuration tuning. Such methods can incorporate additional information to provide hints about the optimization, which can be particularly useful when there is little training data available. These models, for example, could be used to include the expert advice of DBAs or documentation from tuning manuals.

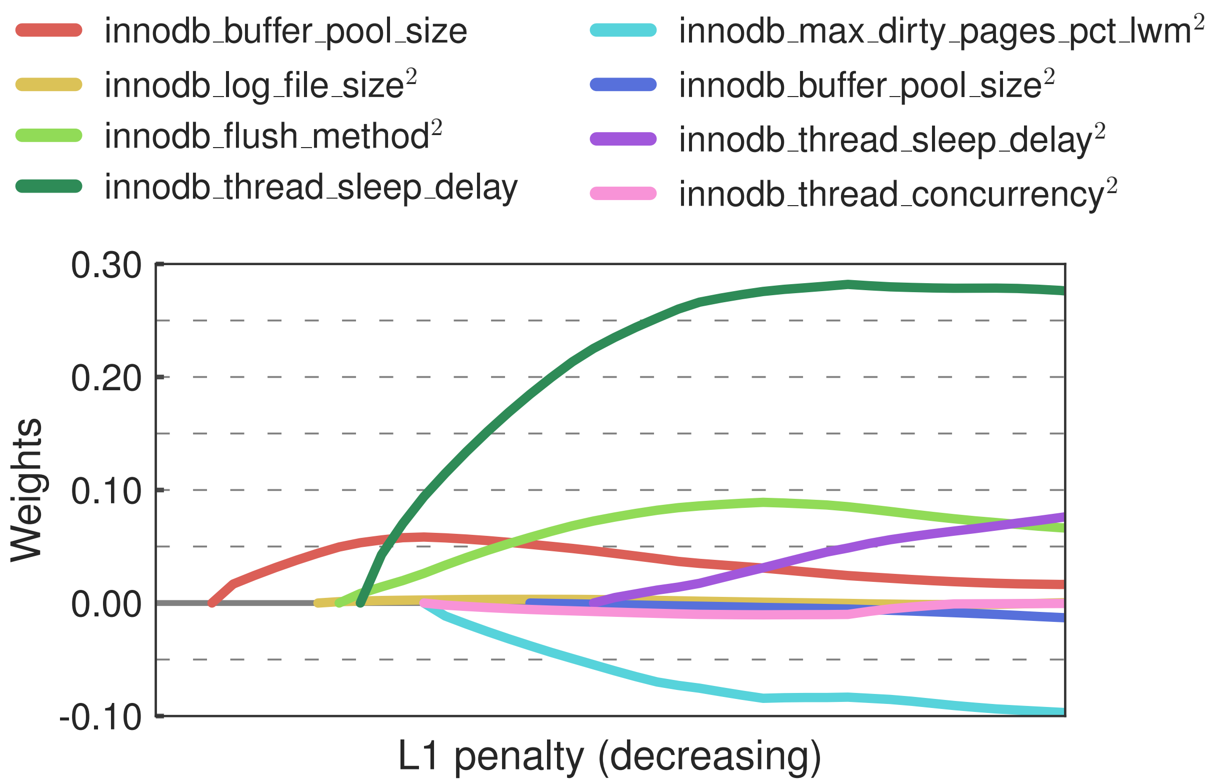


Figure A.1: Lasso Path (MySQL)

Appendix A

Tuning via Gaussian Process Regression

A.1 Identifying Important Knobs

This section extends the discussion of the Lasso path algorithm presented in Section 3.2.1. The results in Figures A.1 to A.3 show the Lasso paths computed for the 99th percentile latency for MySQL, Postgres, and Vector, respectively. For clarity, we show only the eight most impactful fea-

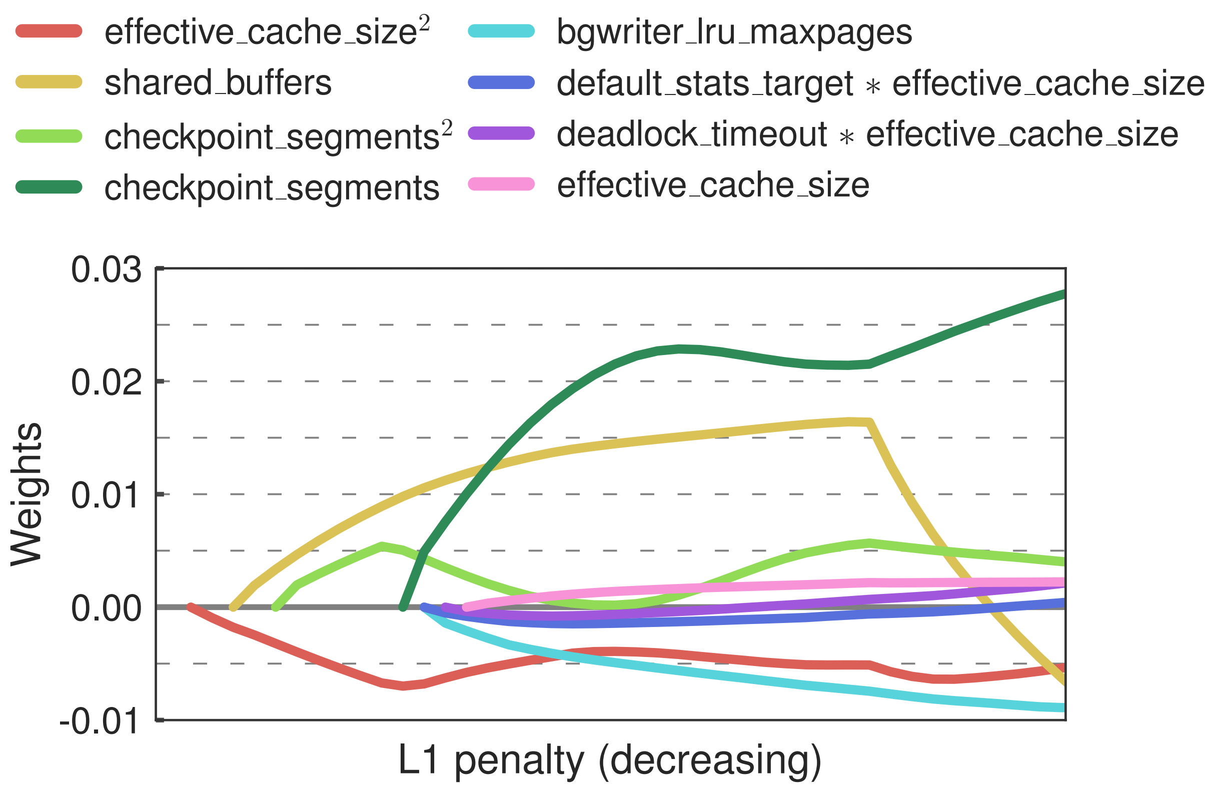


Figure A.2: Lasso Path (Postgres)

tures in these results. Each curve represents a different feature of the regression model's weight vector. These figures show the paths of these weights by plotting them as a function of the L_1 penalty. The order in which the weights appear in the regression indicates how much of an impact the corresponding knobs (or function of knobs) have on the 99th percentile latency. OtterTune uses this ordering to rank the knobs from most to least important.

As described in Section 3.2.2, OtterTune includes second-degree polynomial features to improve the accuracy of its regression models. The two types of features that result from the second-order polynomial expansion of the linear features are products of either two distinct knobs or the same knob. The first type are useful for detecting pairs of knobs that are non-independent. For example, Figure A.2 shows that a dependency exists between two of the knobs that control aspects of the query optimizer: `default_statistics_target` and `effective_cache_size`.

The second type reveals whether a quadratic relationship exists between a knob and the target metric. When we say that a relationship is "quadratic", we do not mean that it is an exact quadratic, but rather that it exhibits some nonlinearity. If the linear and quadratic terms for a knob both appear in the regression around the same time, then its relationship with the target metric is likely quadratic. But if the linear term for a knob appears in the regression much earlier than the quadratic term then the relationship is nearly linear. One knob that the DBMSs have in common is the size of the buffer pool. Figures A.1 to A.3 show that, as expected, the relationship between the buffer pool size knob

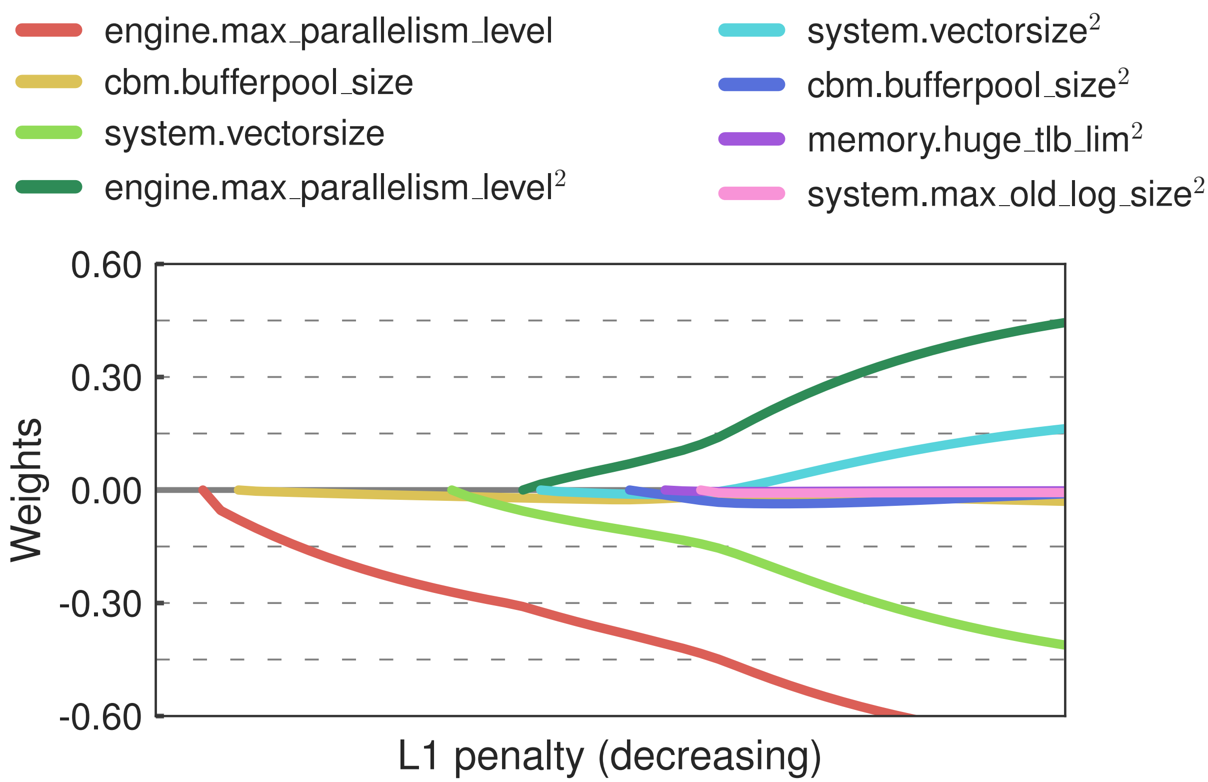


Figure A.3: Lasso Path (Vector)

and the latency is quadratic for all of the DBMSs (the quadratic term for Postgres' knob is not shown but is the 13th to enter the regression).

A.2 Efficacy Comparison

This section is an extension of Section 3.4.6, where we provide the DBMS configurations generated by OtterTune, the DBA, the tuning script, and Amazon AWS that were used in the evaluation. For the configurations generated by OtterTune, the tables display only the 10 most impactful knobs, which are ordered by importance. For all other configurations, the knobs are presented in lexicographical order.

Table A.1: Efficacy Comparison – DBA Configuration (MySQL)

innodb_buffer_pool_dump_at_shutdown	1
innodb_buffer_pool_load_at_startup	1
innodb_buffer_pool_size	12 G
innodb_doublewrite	0
innodb_flush_log_at_trx_commit	0
innodb_flush_method	O_DIRECT
innodb_log_file_size	1 G
skip_performance_schema	–

Table A.2: Efficacy Comparison – OtterTune Configuration (MySQL)

innodb_buffer_pool_size	8.8 G
innodb_thread_sleep_delay	0
innodb_flush_method	O_DIRECT
innodb_log_file_size	1.3 G
innodb_thread_concurrency	0
innodb_max_dirty_pages_pct_lwm	0
innodb_read_ahead_threshold	56
innodb_adaptive_max_sleep_delay	150000
innodb_buffer_pool_instances	8
thread_cache_size	9

Table A.3: Efficacy Comparison – Amazon RDS Configuration (MySQL)

innodb_buffer_pool_size	10.9 G
innodb_flush_method	O_DIRECT
innodb_log_file_size	128 M
key_buffer_size	16 M
max_binlog_size	128 M
read_buffer_size	256 k
read_rnd_buffer_size	512 M
table_open_cache_instances	16
thread_cache_size	20

Table A.4: Efficacy Comparison – Tuning Script Configuration (MySQL)

innodb_buffer_pool_instances	4
innodb_buffer_pool_size	4 G
query_cache_limit	2 G
query_cache_size	2 G
query_cache_type	1

Table A.5: Efficacy Comparison – DBA Configuration (Postgres)

bgwriter_lru_maxpages	1000
bgwriter_lru_multiplier	4
checkpoint_completion_target	0.9
checkpoint_segments	32
checkpoint_timeout	60 min
cpu_tuple_cost	0.03
effective_cache_size	10 G
from_collapse_limit	20
join_collapse_limit	20
maintenance_work_mem	1 G
random_page_cost	1
shared_buffers	2 G
wal_buffers	32 M
work_mem	150 M

Table A.6: Efficacy Comparison – OtterTune Configuration (Postgres)

shared_buffers	4 G
checkpoint_segments	540
effective_cache_size	18 G
bgwriter_lru_maxpages	1000
bgwriter_delay	213 ms
checkpoint_completion_target	0.8
deadlock_timeout	6s
default_statistics_target	78
effective_io_concurrency	3
checkpoint_timeout	1h

Table A.7: Efficacy Comparison – Amazon RDS Configuration (Postgres)

checkpoint_completion_target	0.9
checkpoint_segments	16
effective_cache_size	7.2 G
maintenance_work_mem	243 M
max_stack_depth	6 M
shared_buffers	3.6 G
wal_buffers	16 M

Table A.8: Efficacy Comparison – Tuning Script Configuration (Postgres)

checkpoint_completion_target	0.9
checkpoint_segments	64
default_statistics_target	100
effective_cache_size	23.3 G
maintenance_work_mem	1.9 G
shared_buffers	7.8 G
wal_buffers	16 M
work_mem	40 M

Appendix B

Tuning in the Real World

This section provides the knob configurations generated by the tuning algorithms from our experiments in Chapter 4 for Oracle.

Table B.1: Tuning Knobs Selected by DBA – GPR (10 knobs)

`_enable_numa_support`	FALSE
`_pga_max_size`	3.9 G
`_smm_max_size`	1.4 M
`_smm_px_max_size`	3.5 M
`db_32k_cache_size`	12.1 G
`db_cache_size`	27.9 G
`db_file_multiblock_read_count`	82
`log_buffer`	2 M
`optimizer_adaptive_plans`	FALSE
`optimizer_features_enable`	12.1.0.2
`shared_pool_size`	1.6 G

Table B.2: Tuning Knobs Selected by DBA – DNN (10 knobs)

_enable_numa_support	FALSE
_pga_max_size	5.2 G
_smm_max_size	1.4 M
_smm_px_max_size	3.5 M
db_32k_cache_size	15 G
db_cache_size	20.6 G
db_file_multiblock_read_count	104
log_buffer	1.6 M
optimizer_adaptive_plans	FALSE
optimizer_features_enable	12.2.0.1
shared_pool_size	5 G

Table B.3: Tuning Knobs Selected by DBA – DDPG (10 knobs)

_enable_numa_support	FALSE
_pga_max_size	4.4 G
_smm_max_size	1.4 M
_smm_px_max_size	3.5 M
db_32k_cache_size	7.7 G
db_cache_size	19.2 G
db_file_multiblock_read_count	98
log_buffer	11 M
optimizer_adaptive_plans	FALSE
optimizer_features_enable	12.1.0.2
shared_pool_size	2.8 G

Table B.4: Tuning Knobs Selected by DBA – DDPG++ (10 knobs)

_enable_numa_support	FALSE
_pga_max_size	6.7 G
_smm_max_size	1.4 M
_smm_px_max_size	3.5 M
db_32k_cache_size	15 G
db_cache_size	27.8 G
db_file_multiblock_read_count	57
log_buffer	20 M
optimizer_adaptive_plans	FALSE
optimizer_features_enable	12.2.0.1
shared_pool_size	4.1 G

Table B.5: Tuning Knobs Selected by DBA – LHS (10 knobs)

_enable_numa_support	FALSE
_pga_max_size	222 M
_smm_max_size	111 k
_smm_px_max_size	3.5 M
db_32k_cache_size	14.8 G
db_cache_size	16 G
db_file_multiblock_read_count	14
log_buffer	9 M
optimizer_adaptive_plans	TRUE
optimizer_features_enable	11.2.0.4
shared_pool_size	3.4 G

Table B.6: Tuning Knobs Selected by DBA – GPR (20 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	TRUE
_pga_max_size	100 M
_smm_max_size	50 k
_smm_px_max_size	3.5 M
db_32k_cache_size	2.5 G
db_cache_size	20.8 G
db_file_multiblock_read_count	39
db_writer_processes	7
disk_asynch_io	TRUE
filesystemio_options	none
ioseektim	5
iotfrspeed	20393
large_pool_size	128 M
log_buffer	6 M
optimizer_adaptive_plans	FALSE
optimizer_adaptive_statistics	TRUE
optimizer_dynamic_sampling	2
optimizer_features_enable	12.1.0.1
shared_pool_size	2.1 G

Table B.7: Tuning Knobs Selected by DBA – DNN (20 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	TRUE
_pga_max_size	1.1 G
_smm_max_size	564 k
_smm_px_max_size	3.5 M
db_32k_cache_size	12.5 G
db_cache_size	29.5 G
db_file_multiblock_read_count	88
db_writer_processes	5
disk_asynch_io	TRUE
filesystemio_options	setall
ioseektim	8
iotfrspeed	119624
large_pool_size	384 M
log_buffer	1.6 M
optimizer_adaptive_plans	TRUE
optimizer_adaptive_statistics	FALSE
optimizer_dynamic_sampling	3
optimizer_features_enable	12.1.0.1
shared_pool_size	4.6 G

Table B.8: Tuning Knobs Selected by DBA – DDPG (20 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	FALSE
_pga_max_size	2.8 G
_smm_max_size	1.4 M
_smm_px_max_size	3.5 M
db_32k_cache_size	12.7 G
db_cache_size	11.7 G
db_file_multiblock_read_count	74
db_writer_processes	1
disk_asynch_io	TRUE
filesystemio_options	setall
ioseektim	9
iotfrspeed	69125
large_pool_size	192 M
log_buffer	18 M
optimizer_adaptive_plans	FALSE
optimizer_adaptive_statistics	TRUE
optimizer_dynamic_sampling	4
optimizer_features_enable	12.1.0.1
shared_pool_size	4 G

Table B.9: Tuning Knobs Selected by DBA – DDPG++ (20 knobs)

`_enable_numa_support`	FALSE
`_optimizer_use_feedback`	TRUE
`_pga_max_size`	113 M
`_smm_max_size`	56 k
`_smm_px_max_size`	3.5 M
`db_32k_cache_size`	15 G
`db_cache_size`	29.5 G
`db_file_multiblock_read_count`	40
`db_writer_processes`	2
`disk_asynch_io`	TRUE
`filesystemio_options`	setall
`ioseektim`	10
`iotfrspeed`	190000
`large_pool_size`	384 M
`log_buffer`	1.6 M
`optimizer_adaptive_plans`	FALSE
`optimizer_adaptive_statistics`	TRUE
`optimizer_dynamic_sampling`	2
`optimizer_features_enable`	12.1.0.1
`shared_pool_size`	5 G

Table B.10: Tuning Knobs Selected by DBA – LHS (20 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	TRUE
_pga_max_size	4.4 G
_smm_max_size	1.4 M
_smm_px_max_size	3.5 M
db_32k_cache_size	12.9 G
db_cache_size	23.2 G
db_file_multiblock_read_count	25
db_writer_processes	1
disk_asynch_io	TRUE
filesystemio_options	setall
ioseektim	7
iotfrspeed	55847
large_pool_size	384 M
log_buffer	6 M
optimizer_adaptive_plans	TRUE
optimizer_adaptive_statistics	FALSE
optimizer_dynamic_sampling	6
optimizer_features_enable	12.2.0.1
shared_pool_size	4.5 G

Table B.11: Tuning Knobs Selected by DBA – GPR (40 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	TRUE
_pga_max_size	845 M
_smm_max_size	422 k
_smm_px_max_size	4.1 M
_unnest_subquery	TRUE
approx_for_aggregation	FALSE
approx_for_count_distinct	FALSE
approx_for_percentile	NONE
cursor_invalidation	IMMEDIATE
cursor_sharing	EXACT
db_32k_cache_size	10.6 G
db_big_table_cache_percent_target	0
db_cache_size	10.5 G
db_file_multiblock_read_count	71
db_keep_cache_size	0
db_recycle_cache_size	0
db_writer_processes	1
disk_asynch_io	TRUE
filesystemio_options	setall
hs_autoregister	TRUE
ioseektim	10
iotfrspeed	4096
java_jit_enabled	TRUE
java_pool_size	64 M
large_pool_size	64 M
log_archive_max_processes	6
log_buffer	1.6 M
optimizer_adaptive_plans	TRUE
optimizer_adaptive_statistics	FALSE
optimizer_dynamic_sampling	3
optimizer_features_enable	12.1.0.1
optimizer_mode	CHOOSE
pga_aggregate_target	8.1 G
plsql_optimize_level	3
result_cache_max_result	14
session_cached_cursors	68
shared_pool_reserved_size	80 M
shared_pool_size	1024 M
workarea_size_policy	AUTO

Table B.12: Tuning Knobs Selected by DBA – DNN (40 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	TRUE
_pga_max_size	2.2 G
_smm_max_size	518 k
_smm_px_max_size	1.3 M
_unnest_subquery	FALSE
approx_for_aggregation	FALSE
approx_for_count_distinct	FALSE
approx_for_percentile	NONE
cursor_invalidation	IMMEDIATE
cursor_sharing	EXACT
db_32k_cache_size	15 G
db_big_table_cache_percent_target	0
db_cache_size	12.1 G
db_file_multiblock_read_count	8
db_keep_cache_size	384 M
db_recycle_cache_size	64 M
db_writer_processes	1
disk_asynch_io	TRUE
filesystemio_options	setall
hs_autoregister	FALSE
ioseektim	1
iotfrspeed	57433
java_jit_enabled	TRUE
java_pool_size	384 M
large_pool_size	128 M
log_archive_max_processes	1
log_buffer	12 M
optimizer_adaptive_plans	FALSE
optimizer_adaptive_statistics	FALSE
optimizer_dynamic_sampling	2
optimizer_features_enable	12.2.0.1
optimizer_mode	CHOOSE
pga_aggregate_target	2.5 G
plsql_optimize_level	0
result_cache_max_result	8
session_cached_cursors	91
shared_pool_reserved_size	25 M
shared_pool_size	1.8 G
workarea_size_policy	AUTO

Table B.13: Tuning Knobs Selected by DBA – DDPG (40 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	TRUE
_pga_max_size	2.2 G
_smm_max_size	1022 k
_smm_px_max_size	2.5 M
_unnest_subquery	FALSE
approx_for_aggregation	FALSE
approx_for_count_distinct	FALSE
approx_for_percentile	PERCENTILE_DISC
cursor_invalidation	DEFERRED
cursor_sharing	EXACT
db_32k_cache_size	7.9 G
db_big_table_cache_percent_target	45
db_cache_size	16.2 G
db_file_multiblock_read_count	139
db_keep_cache_size	320 M
db_recycle_cache_size	320 M
db_writer_processes	7
disk_asynch_io	TRUE
filesystemio_options	none
hs_autoregister	TRUE
ioseektim	5
iotfrspeed	76574
java_jit_enabled	TRUE
java_pool_size	576 M
large_pool_size	192 M
log_archive_max_processes	12
log_buffer	10 M
optimizer_adaptive_plans	FALSE
optimizer_adaptive_statistics	FALSE
optimizer_dynamic_sampling	5
optimizer_features_enable	12.1.0.1
optimizer_mode	FIRST_ROWS_1000
pga_aggregate_target	5.0 G
plsql_optimize_level	2
result_cache_max_result	20
session_cached_cursors	97
shared_pool_reserved_size	178 M
shared_pool_size	3.3 G
workarea_size_policy	AUTO

Table B.14: Tuning Knobs Selected by DBA – DDPG++ (40 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	TRUE
_pga_max_size	100 M
_smm_max_size	50 k
_smm_px_max_size	4.6 M
_unnest_subquery	FALSE
approx_for_aggregation	FALSE
approx_for_count_distinct	FALSE
approx_for_percentile	NONE
cursor_invalidation	DEFERRED
cursor_sharing	FORCE
db_32k_cache_size	14.4 G
db_big_table_cache_percent_target	1
db_cache_size	2.9 G
db_file_multiblock_read_count	249
db_keep_cache_size	0
db_recycle_cache_size	128 M
db_writer_processes	1
disk_asynch_io	TRUE
filesystemio_options	asynch
hs_autoregister	FALSE
ioseektim	2
iotfrspeed	44791
java_jit_enabled	FALSE
java_pool_size	960 M
large_pool_size	128 M
log_archive_max_processes	1
log_buffer	20 M
optimizer_adaptive_plans	TRUE
optimizer_adaptive_statistics	FALSE
optimizer_dynamic_sampling	2
optimizer_features_enable	12.2.0.1
optimizer_mode	CHOOSE
pga_aggregate_target	9.2 G
plsql_optimize_level	3
result_cache_max_result	0
session_cached_cursors	194
shared_pool_reserved_size	45 M
shared_pool_size	1024 M
workarea_size_policy	AUTO

Table B.15: Tuning Knobs Selected by DBA – LHS (40 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	FALSE
_pga_max_size	2.1 G
_smm_max_size	788 k
_smm_px_max_size	1.9 M
_unnest_subquery	TRUE
approx_for_aggregation	TRUE
approx_for_count_distinct	TRUE
approx_for_percentile	PERCENTILE_DISC DETERMINISTIC
cursor_invalidation	DEFERRED
cursor_sharing	EXACT
db_32k_cache_size	4.2 G
db_big_table_cache_percent_target	21
db_cache_size	7.4 G
db_file_multiblock_read_count	75
db_keep_cache_size	64 M
db_recycle_cache_size	192 M
db_writer_processes	8
disk_asynch_io	FALSE
filesystemio_options	none
hs_autoregister	TRUE
ioseektim	1
iotfrspeed	63615
java_jit_enabled	FALSE
java_pool_size	384 M
large_pool_size	192 M
log_archive_max_processes	12
log_buffer	1.6 M
optimizer_adaptive_plans	TRUE
optimizer_adaptive_statistics	FALSE
optimizer_dynamic_sampling	4
optimizer_features_enable	11.2.0.4
optimizer_mode	CHOOSE
pga_aggregate_target	3.8 G
plsql_optimize_level	1
result_cache_max_result	23
session_cached_cursors	25
shared_pool_reserved_size	184 M
shared_pool_size	4.5 G
workarea_size_policy	MANUAL

Table B.16: Tuning Knobs Ranked by OtterTune – GPR (10 knobs)

`_enable_numa_support`	FALSE
`_smm_max_size`	100 k
`db_32k_cache_size`	8.4 G
`db_cache_size`	13.4 G
`db_file_multiblock_read_count`	75
`db_keep_cache_size`	192 M
`disk_asynch_io`	TRUE
`java_pool_size`	256 M
`optimizer_dynamic_sampling`	5
`optimizer_features_enable`	12.1.0.2
`session_cached_cursors`	152

Table B.17: Tuning Knobs Ranked by OtterTune – DNN (10 knobs)

`_enable_numa_support`	FALSE
`_smm_max_size`	100 k
`db_32k_cache_size`	13.5 G
`db_cache_size`	6.8 G
`db_file_multiblock_read_count`	21
`db_keep_cache_size`	128 M
`disk_asynch_io`	TRUE
`java_pool_size`	192 M
`optimizer_dynamic_sampling`	2
`optimizer_features_enable`	12.2.0.1
`session_cached_cursors`	200

Table B.18: Tuning Knobs Ranked by OtterTune – DDPG (10 knobs)

`_enable_numa_support`	FALSE
`_smm_max_size`	100 k
`db_32k_cache_size`	12.5 G
`db_cache_size`	23.2 G
`db_file_multiblock_read_count`	88
`db_keep_cache_size`	384 M
`disk_asynch_io`	TRUE
`java_pool_size`	640 M
`optimizer_dynamic_sampling`	8
`optimizer_features_enable`	12.1.0.2
`session_cached_cursors`	0

Table B.19: Tuning Knobs Ranked by OtterTune – DDPG++ (10 knobs)

`_enable_numa_support`	FALSE
`_smm_max_size`	100 k
`db_32k_cache_size`	9.9 G
`db_cache_size`	6.7 G
`db_file_multiblock_read_count`	241
`db_keep_cache_size`	0
`disk_asynch_io`	TRUE
`java_pool_size`	1024 M
`optimizer_dynamic_sampling`	4
`optimizer_features_enable`	12.2.0.1
`session_cached_cursors`	19

Table B.20: Tuning Knobs Ranked by OtterTune – LHS (10 knobs)

`_enable_numa_support`	FALSE
`_smm_max_size`	100 k
`db_32k_cache_size`	12.4 G
`db_cache_size`	20.1 G
`db_file_multiblock_read_count`	98
`db_keep_cache_size`	128 M
`disk_asynch_io`	TRUE
`java_pool_size`	896 M
`optimizer_dynamic_sampling`	5
`optimizer_features_enable`	11.2.0.4
`session_cached_cursors`	184

Table B.21: Tuning Knobs Ranked by OtterTune – GPR (20 knobs)

_enable_numa_support	FALSE
_smm_max_size	100 k
_unnest_subquery	FALSE
cursor_invalidation	DEFERRED
db_32k_cache_size	15 G
db_big_table_cache_percent_target	9
db_cache_size	8.9 G
db_file_multiblock_read_count	244
db_keep_cache_size	448 M
db_writer_processes	2
disk_asynch_io	FALSE
ioseektim	7
java_pool_size	576 M
large_pool_size	128 M
log_archive_max_processes	10
log_buffer	14 M
optimizer_dynamic_sampling	3
optimizer_features_enable	12.1.0.1
optimizer_mode	CHOOSE
plsql_optimize_level	3
session_cached_cursors	98

Table B.22: Tuning Knobs Ranked by OtterTune – DNN (20 knobs)

_enable_numa_support	FALSE
_smm_max_size	100 k
_unnest_subquery	FALSE
cursor_invalidation	DEFERRED
db_32k_cache_size	12.5 G
db_big_table_cache_percent_target	35
db_cache_size	25.6 G
db_file_multiblock_read_count	256
db_keep_cache_size	256 M
db_writer_processes	2
disk_asynch_io	TRUE
ioseektim	4
java_pool_size	128 M
large_pool_size	128 M
log_archive_max_processes	4
log_buffer	20 M
optimizer_dynamic_sampling	2
optimizer_features_enable	12.2.0.1
optimizer_mode	CHOOSE
plsql_optimize_level	0
session_cached_cursors	0

Table B.23: Tuning Knobs Ranked by OtterTune – DDPG (20 knobs)

`_enable_numa_support`	FALSE
`_smm_max_size`	100 k
`_unnest_subquery`	TRUE
`cursor_invalidation`	DEFERRED
`db_32k_cache_size`	7 G
`db_big_table_cache_percent_target`	41
`db_cache_size`	12.1 G
`db_file_multiblock_read_count`	86
`db_keep_cache_size`	320 M
`db_writer_processes`	6
`disk_asynch_io`	TRUE
`ioseektim`	6
`java_pool_size`	448 M
`large_pool_size`	64 M
`log_archive_max_processes`	12
`log_buffer`	9 M
`optimizer_dynamic_sampling`	6
`optimizer_features_enable`	11.2.0.4
`optimizer_mode`	FIRST_ROWS_1000
`plsql_optimize_level`	1
`session_cached_cursors`	115

Table B.24: Tuning Knobs Ranked by OtterTune – DDPG++ (20 knobs)

_enable_numa_support	FALSE
_smm_max_size	100 k
_unnest_subquery	FALSE
cursor_invalidation	IMMEDIATE
db_32k_cache_size	14.6 G
db_big_table_cache_percent_target	0
db_cache_size	4.9 G
db_file_multiblock_read_count	236
db_keep_cache_size	512 M
db_writer_processes	1
disk_asynch_io	FALSE
ioseektim	10
java_pool_size	64 M
large_pool_size	128 M
log_archive_max_processes	30
log_buffer	1.6 M
optimizer_dynamic_sampling	2
optimizer_features_enable	12.2.0.1
optimizer_mode	CHOOSE
plsql_optimize_level	3
session_cached_cursors	0

Table B.25: Tuning Knobs Ranked by OtterTune – LHS (20 knobs)

_enable_numa_support	FALSE
_smm_max_size	100 k
_unnest_subquery	FALSE
cursor_invalidation	DEFERRED
db_32k_cache_size	5.6 G
db_big_table_cache_percent_target	7
db_cache_size	12.1 G
db_file_multiblock_read_count	116
db_keep_cache_size	256 M
db_writer_processes	5
disk_asynch_io	FALSE
ioseektim	7
java_pool_size	128 M
large_pool_size	128 M
log_archive_max_processes	26
log_buffer	17 M
optimizer_dynamic_sampling	5
optimizer_features_enable	12.1.0.1
optimizer_mode	FIRST_ROWS_1000
plsql_optimize_level	3
session_cached_cursors	30

Table B.26: Adaptability to Different Workloads – GPR (20 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	FALSE
_pga_max_size	4.6 G
_smm_max_size	1.4 M
_smm_px_max_size	3.5 M
db_32k_cache_size	13 G
db_cache_size	5.9 G
db_file_multiblock_read_count	10
db_writer_processes	2
disk_asynch_io	TRUE
filesystemio_options	asynch
ioseektim	5
iotfrspeed	4096
large_pool_size	128 M
log_buffer	1.6 M
optimizer_adaptive_plans	TRUE
optimizer_adaptive_statistics	FALSE
optimizer_dynamic_sampling	4
optimizer_features_enable	12.2.0.1
shared_pool_size	2.3 G

Table B.27: Adaptability to Different Workloads – DNN (20 knobs)

`_enable_numa_support`	FALSE
`_optimizer_use_feedback`	FALSE
`_pga_max_size`	4.4 G
`_smm_max_size`	1.4 M
`_smm_px_max_size`	3.5 M
`db_32k_cache_size`	14.5 G
`db_cache_size`	4.6 G
`db_file_multiblock_read_count`	83
`db_writer_processes`	9
`disk_asynch_io`	TRUE
`filesystemio_options`	none
`ioseektim`	2
`iotfrspeed`	71402
`large_pool_size`	192 M
`log_buffer`	15 M
`optimizer_adaptive_plans`	FALSE
`optimizer_adaptive_statistics`	TRUE
`optimizer_dynamic_sampling`	7
`optimizer_features_enable`	12.1.0.1
`shared_pool_size`	4.6 G

Table B.28: Adaptability to Different Workloads – DDPG (20 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	FALSE
_pga_max_size	1.8 G
_smm_max_size	902 k
_smm_px_max_size	3.5 M
db_32k_cache_size	15 G
db_cache_size	6.1 G
db_file_multiblock_read_count	8
db_writer_processes	5
disk_asynch_io	TRUE
filesystemio_options	asynch
ioseektim	10
iotfrspeed	21309
large_pool_size	192 M
log_buffer	1.6 M
optimizer_adaptive_plans	TRUE
optimizer_adaptive_statistics	FALSE
optimizer_dynamic_sampling	6
optimizer_features_enable	12.2.0.1
shared_pool_size	4.4 G

Table B.29: Adaptability to Different Workloads – DDPG++ (20 knobs)

_enable_numa_support	FALSE
_optimizer_use_feedback	FALSE
_pga_max_size	3.6 G
_smm_max_size	1.4 M
_smm_px_max_size	3.5 M
db_32k_cache_size	11.4 G
db_cache_size	24.5 G
db_file_multiblock_read_count	97
db_writer_processes	5
disk_asynch_io	FALSE
filesystemio_options	none
ioseektim	9
iotfrspeed	111939
large_pool_size	256 M
log_buffer	11 M
optimizer_adaptive_plans	FALSE
optimizer_adaptive_statistics	FALSE
optimizer_dynamic_sampling	4
optimizer_features_enable	11.2.0.4
shared_pool_size	2.9 G

Bibliography

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

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

[3] OLTPBenchmark.com. http://oltpbenchmark.com.

[4] OtterTune. https://ottertune.cs.cmu.edu.

[5] PostgreSQL Configuration Wizard. http://pgfoundry.org/projects/pgtune/.

[6] RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[7] scikit-learn Documentation – Factor Analysis. http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.FactorAnalysis.html.

[8] scikit-learn Documentation – KMeans. http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[9] Société Générale. https://www.societegenerale.com.

[10] S. Agrawal, S. Chaudhuri, A. Das, and V. Narasayya. Automating layout of relational databases. In *ICDE*, pages 607–618, 2003.

[11] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, 2004.

[12] J. C. Barrett, D. G. Clayton, P. Concannon, B. Akolkar, J. D. Cooper, H. A. Erlich, C. Julier, G. Morahan, J. Nerup, C. Nierras, et al. Genome-wide association study and meta-analysis find that over 40 loci affect risk of type 1 diabetes. *Nature genetics*, 41(6):703–707, 2009.

[13] P. Belknap, B. Dageville, K. Dias, and K. Yagoub. Self-tuning for SQL performance in Oracle Database 11g. In *ICDE*, pages 1694–1700, 2009.

[14] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The journal of machine learning research*, 13:281–305, Feb. 2012.

[15] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. Jagadish, et al. The asilomar report on database research. *SIGMOD record*, 27(4):74–80, 1998.

[16] A. Bietti, A. Agarwal, and J. Langford. A contextual bandit bake-off. *arXiv preprint arXiv:1802.04064*, 2018.

[17] P. Boncz, T. Neumann, and O. Erling. *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*. 2014.

[18] D. Bouneffouf and I. Rish. A survey on practical applications of multi-armed and contextual bandits. *arXiv preprint arXiv:1904.10040*, 2019.

[19] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[20] E. A. Brewer. High-level optimization via automated statistical modeling. In *PPOPP*, pages 80–91, 1995.

[21] K. P. Brown, M. J. Carey, and M. Livny. Goal-oriented buffer management revisited. In *SIGMOD*, pages 353–364, 1996.

[22] Z. Cao, G. Kuenning, and E. Zadok. Carver: Finding important parameters for storage system tuning. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 43–57, 2020.

[23] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury advanced series in statistics and decision sciences. Duxbury Press, 2002.

[24] S. Ceri, S. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Trans. Softw. Eng.*, 9(4):487–504, 1983.

[25] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing SQL workloads. In *SIGMOD*, pages 488–499, 2002.

[26] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. *SIGMOD Rec.*, 27(2):367–378, 1998.

[27] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *VLDB*, pages 3–14, 2007.

[28] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *VLDB*, pages 1–10, 2000.

[29] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, et al. The mixed workload ch-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, pages 1–6, 2011.

[30] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.

[31] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-drive approach to database replication and partitioning. In *VLDB*, 2010.

[32] B. Dageville and M. Zait. Sql memory management in oracle9i. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 962–973, 2002.

[33] E. Danna and L. Perron. Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs. In *Principles and Practice of Constraint Programming*, volume 2833, pages 817–821, 2003.

[34] B. Debnath, D. Lilja, and M. Mokbel. SARD: A statistical approach for ranking database tuning parameters. In *ICDEW*, pages 11–18, 2008.

[35] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in oracle. In *CIdR*, 2005.

[36] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: an extensible testbed for benchmarking relational databases. In *VLDB*, pages 277–288, 2013.

[37] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTuned. *VLDB*, 2:1246–1257, August 2009.

[38] D. Dworin. Data science revealed: A data-driven glimpse into the burgeoning new field. Dec. 2011.

[39] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *The Annals of Statistics*, 32(2):407–499, 2004.

[40] K. Engeleiter, J. Beresniewicz, and C. Gervasio. Maximizing database performance: Performance tuning with db time. https://www.oracle.com/technetwork/oem/db-mgmt/s317294-db-perf-tuning-with-db-time-181631.pdf, 2010.

[41] D. G. Feitelson and M. Naaman. Self-tuning systems. *IEEE Softw.*, 16(2):52–60, Mar. 1999.

[42] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 13(1):91–128, 1988.

[43] F. Focacci, F. Laburthe, and A. Lodi. *Handbook of Metaheuristics*, chapter Local Search and Constraint Programming. Springer, 2003.

[44] M. R. Frank, E. Omiecinski, and S. B. Navathe. Adaptive and automated index selection in RDBMS. In *EDBT*, pages 277–292, 1992.

[45] D. G. Benoit. Automatic diagnosis of performance problems in database management systems. In *ICAC*, pages 326–327, 2005.

[46] L. Galanis, S. Buranawatanachoke, R. Colle, B. Dageville, K. Dias, J. Klein, S. Papadomanolakis, L. L. Tan, V. Venkataramani, Y. Wang, et al. Oracle database replay. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1159–1170, 2008.

[47] M. Hammer and A. Chan. Index selection in a self-adaptive data base management system. In *SIGMOD*, pages 1–8, 1976.

[48] M. Hammer and B. Niamir. A heuristic approach to attribute partitioning. In *SIGMOD*, pages 93–101, 1979.

[49] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[50] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2001.

[51] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, pages 261–272, 2011.

[52] C. R. Hicks and K. V. Turner. *Fundamental Concepts in the Design of Experiments*. Oxford University Press, 5 edition, Mar. 1997.

[53] W. W. Hsu, A. J. Smith, and H. C. Young. Characteristics of production database workloads and the tpc benchmarks. *IBM Systems Journal*, 40(3):781–802, 2001.

[54] M. Y. L. Ip, L. V. Saxton, and V. V. Raghavan. On the selection of an optimal set of indexes. *IEEE Trans. Softw. Eng.*, 9(2):135–143, 1983.

[55] A. Jain, M. Murty, and P. Flynn. Data clustering: A review. volume 31, pages 264–323, 1999.

[56] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[57] K. Kanellis, R. Alagappan, and S. Venkataraman. Too many knobs to tune? towards faster database tuning by pre-selecting important knobs. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[58] A. Krause and C. S. Ong. Contextual gaussian process bandit optimization. In *NIPS*, pages 2447–2455, 2011.

[59] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on Read-Optimized databases using Multi-Core CPUs. *VLDB*, 5:61–72, September 2011.

[60] S. Kumar. Oracle Database 10g: The self-managing database, Nov. 2003. White Paper.

[61] M. Kunjir and S. Babu. Black or white? how to develop an autotuner for memory-based analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1667–1683, 2020.

[62] E. Kwan, S. Lightstone, A. Storm, and L. Wu. Automatic configuration for IBM DB2 universal database. Technical report, IBM, jan 2002.

[63] E. Kwan, S. Lightstone, A. Storm, and L. Wu. Automatic configuration for IBM DB2 universal database. Technical report, IBM, jan 2002.

[64] D. Laney. 3-D data management: Controlling data volume, velocity and variety. Feb. 2001.

[65] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. volume 12, pages 2118–2130. VLDB Endowment, 2019.

[66] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670, 2010.

[67] M. Linster. Best practices for becoming an exceptional postgres dba. `http://www.enterprisedb.com/best-practices-becoming-exceptional-postgres-dba`, Aug. 2014.

[68] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. *SIGMETRICS Perform. Eval. Rev.*, 15(1):69–77, 1987.

[69] T. Lu, D. Pál, and M. Pál. Contextual multi-armed bandits. In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics*, pages 485–492. JMLR Workshop and Conference Proceedings, 2010.

[70] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the*

2018 International Conference on Management of Data, SIGMOD '18, pages 631–645, 2018.

[71] K. Mahadik, Q. Wu, S. Li, and A. Sabne. Fast distributed bandits for online recommendation systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–13, 2020.

[72] M. Majzoubi, C. Zhang, R. Chari, A. Krishnamurthy, J. Langford, and A. Slivkins. Efficient contextual bandits with continuous actions. *arXiv preprint arXiv:2006.06040*, 2020.

[73] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *MASCOTS*, pages 239–248, 2005.

[74] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, SIGMOD, pages 1137–1148, 2011.

[75] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, 2004.

[76] N. Park, W. Xiao, K. Choi, and D. J. Lilja. A statistical evaluation of the impact of parameter selection on storage system benchmarks. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, volume 6, 2011.

[77] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *VLDB*, 5:85–96, October 2011.

[78] D. T. Pham, S. S. Dimov, and C. D. Nguyen. Selection of k in k-means clustering. In *IMechE*, volume 219, 2005.

[79] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz. Parameter space noise for exploration. In *ICLR*, 2018.

[80] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.

[81] C. E. Rasmussen and C. K. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[82] A. Rosenberg. Improving query performance in data warehouses. *Business Intelligence Journal*, 11, Jan. 2006.

[83] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, 2010.

[84] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal*, 7(1):48–66, 1998.

[85] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang. Mrtuner: a toolkit to enable holistic optimization for mapreduce jobs. *Proceedings of the VLDB Endowment*, 7(13):1319–1330, 2014.

[86] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, pages 953–966, 2008.

[87] N. Srinivas, A. Krause, S. Kakade, and M. Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the 27th International Conference on Machine Learning*, 2010.

[88] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[89] M. Stonebraker, S. Madden, and P. Dubey. Intel "big data" science and technology center vision and execution plan. *SIGMOD Rec.*, 42(1):44–49, May 2013.

[90] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *VLDB*, pages 1081–1092, 2006.

[91] C. Sugar. *Techniques for clustering and classification with applications to medical problems*. PhD thesis, Stanford University, 1998.

[92] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer. Using probabilistic reasoning to automate software tuning. SIGMETRICS, pages 404–405, 2004.

[93] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang. ibtune: Individualized buffer tuning for large-scale cloud databases. *Proceedings of the VLDB Endowment*, 12(10):1221–1234, 2019.

[94] A. Tewari and S. A. Murphy. From ads to interventions: Contextual bandits in mobile health. In *Mobile Health*, pages 495–517. Springer, 2017.

[95] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`, June 2007.

[96] The Transaction Processing Council. TPC-H Benchmark (Revision 2.16.0). `http://www.tpc.org/tpch/spec/tpch2.16.0.pdf`, December 2013.

[97] W. Tian, P. Martin, and W. Powley. Techniques for automatically sizing multiple buffer pools in DB2. In *CASCON*, pages 294–302, 2003.

[98] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58:267–288, 1996.

[99] R. Tibshirani, G. Walther, and T. Hastie. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 63:411–423, 2001.

[100] R. J. Tibshirani, A. Rinaldo, R. Tibshirani, and L. Wasserman. Uniform asymptotic inference and the bootstrap after model selection. *arXiv preprint arXiv:1506.06266*, 2015.

[101] R. J. Tibshirani, J. Taylor, R. Lockhart, and R. Tibshirani. Exact post-selection inference for sequential regression procedures. *arXiv preprint arXiv:1401.3889*, 2014.

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

[103] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM*

International Conference on Management of Data, SIGMOD '17, pages 1009–1024, 2017.

[104] G. Weikum, C. Hasse, A. Mönkeberg, and P. Zabback. The COMFORT automatic tuning project. *Information Systems*, 19(5):381–432, July 1994.

[105] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *WWW*, pages 287–296, 2004.

[106] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's sql performance analyzer. *IEEE Data Engineering Bulletin*, 31(1), 2008.

[107] B. Zhang. `https://github.com/bohanjason/ottertune`.

[108] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, pages 265–276, 2014.

[109] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 415–432, 2019.

[110] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350, 2017.

[111] D. C. Zilio. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems*. PhD thesis, University of Toronto, 1998.

[112] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.