Automated Database Workload Characterization, Mapping, and Tuning through Machine Learning

Madeline MacDonald
University of Utah

UUCS-18-007

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

10 December 2018

Abstract

Database management systems (DBMS) have numerous configuration settings, such as buffer or memory sizes, that have an impact on the database's performance. These settings require tuning to optimize performance, and the optimal values for each setting are highly dependent on the server's workload and hardware. The process of tuning a DBMS requires a highly skilled database administrator (DBA) to manually test different configurations, but this is an long and expensive process. Furthermore, the final tuning configuration is specific to the workload, DBMS version, and hardware, and if any of those factors change the tuning process must be redone.

OtterTune, a project by the Database Group at Carnegie Mellon University, solves this tuning problem by using machine learning techniques to automatically tune databases. OtterTune observes a new database workload, isolates the most important system metrics from that workload, and then maps the new workload to a similar previously tuned workload. It learns from previous tuning results for the similar workload and generates a new configuration recommendation automatically.

In this work, analyze OtterTune's architecture and approach to automating database tuning, specifically focusing on the workload characterization and mapping techniques. We then implement a lightweight version of OtterTune that automatically tunes PostgreSQL 9.6 databases. We utilize open source OtterTune code from Carnegie Mellon in this solution to ensure that the results are comparable. Finally, we test our implementation of OtterTune on a selected set of database benchmarks and analyze our results.

AUTOMATED DATABASE WORKLOAD CHARACTERIZATION, MAPPING, AND TUNING THROUGH MACHINE LEARNING

by

Madeline MacDonald

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Bachelor of Computer Science

School of Computing
The University of Utah
December 2018

Feifei Li	H. James de St. Germain
[illegible]	
signature	*signature*
12/10/2018	
date	*date*
Ross Whitaker	
signature	
date	

ABSTRACT

Database management systems (DBMS) have numerous configuration settings, such as buffer or memory sizes, that have an impact on the database's performance. These settings require tuning to optimize performance, and the optimal values for each setting are highly dependent on the server's workload and hardware. The process of tuning a DBMS requires a highly skilled database administrator (DBA) to manually test different configurations, but this is an long and expensive process. Furthermore, the final tuning configuration is specific to the workload, DBMS version, and hardware, and if any of those factors change the tuning process must be redone.

OtterTune, a project by the Database Group at Carnegie Mellon University, solves this tuning problem by using machine learning techniques to automatically tune databases. OtterTune observes a new database workload, isolates the most important system metrics from that workload, and then maps the new workload to a similar previously tuned workload. It learns from previous tuning results for the similar workload and generates a new configuration recommendation automatically.

In this work, analyze OtterTune's architecture and approach to automating database tuning, specifically focusing on the workload characterization and mapping techniques. We then implement a lightweight version of OtterTune that automatically tunes PostgreSQL 9.6 databases. We utilize open source OtterTune code from Carnegie Mellon in this solution to ensure that the results are comparable. Finally, we test our implementation of OtterTune on a selected set of database benchmarks and analyze our results.

CONTENTS

ABSTRACT . . . . **ii**

CHAPTERS

1. INTRODUCTION **1**

2. BACKGROUND AND RELATED WORK **3**

2.1 Motivation 3
2.2 Related work 3

3. OTTERTUNE **6**

3.1 Overview 6
3.2 Workload Statistics Collection 7
3.2.1 Workload Characterization Methods 7
3.2.2 Factor Analysis 8
3.2.3 K-Means Clustering 8
3.2.4 Workload Mapping 8
3.2.5 Similarity Score Calculation 9
3.2.6 Tuning 10

4. METHODS **11**

4.1 Data Collection 11
4.1.1 Challenges 11
4.1.2 Approach 12
4.2 The OtterTune Architecture 13

5. RESULTS **15**

5.1 Mapping Evaluation 15
5.2 Tuning Evaluation 17
5.3 Conclusions from Evaluations 19

6. FUTURE WORK **20**

6.1 Factorization Machines 20

7. CONCLUSIONS **22**
Bibliography 23

CHAPTER 1

INTRODUCTION

In modern business, efficiently managing data is vital to success. Having a database that quickly allows access to inventory records, or that efficiently handles large volumes of order placement, has a huge impact on profit. To make sure that enterprise databases are up to the task, it's necessary to configure each database to be specialized for the hardware it runs on, the data it stores, and the ways in which it's used to read or write data. Each database system has dozens of setting parameters that can be changed to increase throughput or to reduce latency, but their optimum values are dependent largely on the nature of the database workload.

When analyzing a database workload, the issue isn't trying to gather more data, but rather learning how make sense of the data we have. Modern database management systems (DBMS) expose huge amounts of information about database workloads, everything from the queries themselves to table schema information to system metrics, but knowing what information is relevant and what to ignore poses an ongoing question. The process of determining which parameters are relevant, analyzing the data values for that subset of parameters, and classifying database workloads based on those values, is referred to as workload characterization.

Selecting relevant metrics to gather statistics on is central to workload characterization, and recent work has shown that machine learning models have a huge potential to improve metric selection and characterization. This thesis focuses on the work done by the Database Group at Carnegie Melon University, and their project OtterTune, which uses a multi-stage workload characterization and mapping algorithm to generate system knob tuning recommendations [10]. Under OtterTune's structure, a database workload is observed while running, and its runtime metrics are collected and sent to a server to be processed. This full set of runtime metrics is then analyzed and pruned until all redundant and uninformative metrics are removed, and the remaining metrics are used to characterize the workload. This system allows workloads to be characterized without making any assumptions about the workload type, and even allows for previously characterized workloads to be re-characterized more accurately as new data comes in. Because this method can generalize across different workload types and different DBMS's, it's a powerful new tool for gaining insight into database usage. Ultimately, these workload insights are useful because they enable automated tuning for DBMS systems, and allow previous tuning configurations to be reused and learned from. The OtterTune architecture learns from previous tuning configurations so that it can automatically generate configurations that will optimize latency or throughput for a workload.

This thesis focuses on implementing a simplified version of OtterTune, and evaluating OtterTune's performance on mapping and tuning workloads. Because the results of mapping and tuning workloads are highly dependent on server hardware, database versions, and workload settings it isn't feasible to recreate all of OtterTune's findings. Instead, this project takes a narrower scope of analyzing tuning performance, including using a smaller training data set and focusing on tuning only PostgreSQL 9.6 databases. Experimental evaluations of this simplified version of OtterTune show up to 7x increase in throughput when compared to default configurations.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this section I will explain the motivation for database tuning, the challenges inherent in database tuning, and previous work in the field.

2.1 Motivation

Modern database systems ship with default configurations that are meant to meet the broad needs of many users, but that aren't specialized for any one task. In generalizing settings for quick start up out of the box most settings are passable, but performance won't be anywhere near its full potential. For example, default configuration settings in PostrgreSQL don't make use of parallel query execution, or allocate the full amount of memory available for use on the hardware [1]. By making simple changes to database configurations users can see incredible latency reductions or throughput increases, but knowing how and where to make those simple changes is challenging and intimidating to many users. Effectively tuning a database requires an understanding of how the mechanics of a database work, knowing details of your hardware configuration, and knowing what type of work your database will be doing [12]. Even if a user knows all of those details, many knobs interact with each other, which requires a thorough understanding of the relationships between the settings. Furthermore, with each new database version release the number of database knobs increases [10] which makes it even more complicated to keep up.

2.2 Related work

Workload characterization has been an ongoing research problem for a long time, which means that there have been a variety of different attempts to improve on it. Overall, there are two main approaches to workload characterization: Analysis at the logical level of queries and database schemas, and analysis of internal runtime metrics provided by the DBMS.

The first approach, characterization of workloads from queries and database schemas, relies on collecting statistics about the language used in queries, the way users interact with tables and views, and the differences in how transactions are used between workloads [11]. This can give a rather comprehensive view of how users interact with the database in the workload, and provide insight into the impact of design choices on workloads, making analysis on the logical level a practical choice in some cases. That said, this approach focuses purely on the commands issued and the database structure, and not on query behavior at run time. There are many applications of workload mapping that rely on runtime performance, such as distributed database systems and client-server architectures, and analysis at a logical level doesn't account for those needs [3].

The second approach, analyzing runtime metrics from observed workloads during execution, accounts for the needs of more complicated system architectures by giving insight into the behavior of the workload during execution. Most previous work in this area follows a well established procedure for mapping. First, based off of knowledge of the system and expected workloads, a set of runtime metrics is selected that is believed to be able to characterize a workload. This is usually done by a DBA, or another expert who uses personal knowledge of the workload to select metrics for characterization. Second, data is collected during an observation period as the workload runs. Finally, the collected metric values are analyzed to characterize the workload[3].

The major downside to this approach in practice, is that as systems and software evolve, the set of metric parameters selected isn't guaranteed to generalize or remain relevant. Over time the work done to select a set of parameters must be repeated, which takes expertise about the system and is inaccessible for many users. Workload characterization approaches that require specially selected parameter sets cannot usually be generalized across different types of systems or workloads, and may not even remain optimal on the same system as software updates over time.

CHAPTER 3

OTTERTUNE

One of the most recent, and most promising, approaches to workload characterization comes from the OtterTune project from Carnegie Mellon University [10], lead by Dana Van Aken and Andy Pavlo. Their work is a departure from the previous methods, in that it relies on analyzing runtime metrics but doesn't use a predefined set of metrics to characterize workloads. Instead, OtterTune collects and stores data about workloads it observes, and uses innovative machine learning models to prune away irrelevant metrics until a small subset of metrics remain that characterize the workload. This approach is incredibly powerful, because it allows the same characterization process to be applied to a wide variety of workloads without requiring supervised metric selection.
OtterTune uses workload characterization as part of a larger process of automating DBMS system knob tuning, so it only requires system information that comes from runtime metrics and ignores information about query structures or higher level database organization.

3.1 Overview

Ottertune works through an iterative tuning process in conjunction with a large repository of existing database tuning results to automatically generate new knob configurations. Ottertune collects system runtime metrics, observes the workload for five minutes, then collects metrics again. These before and after metric files are uploaded to the tuning server where they are processed and compared with all previous workloads in the repository to find the closest match. Using the large amounts of tuning data from the repository for the mapped workload, a Gaussian Process Regression model is constructed to optimize throughput, and from that model a new DBMS configuration file is created with recommended settings.

3.2 Workload Statistics Collection

In OtterTune statistics collection for workloads is done using a controller that runs on the database server for the target database. When the controller is run it's given a config file containing the type of DBMS to target, connection info for the target database, and connection info for the OtterTune server. OtterTune uses a modular architecture that allows each type of database to collect statistics differently, which allows for easy testing and implementation.

The stats collector first collects the dbms metrics prior to running the workload, then collects knob configuration data. It then waits for a set time period for the workload to run before collecting metrics again. Finally a summary file is compiled with information about the database type and version and the start and end observation time. Four files are uploaded to OtterTune: A summary of the data collection, the knob configuration, metrics before the workload, and metrics after the workload.

3.2.1 Workload Characterization Methods

Next the server determines which metrics best describe the given workload. This happens in two main phases, factor analysis and K-means clustering. Both of these stages are meant to remove unimportant information and to isolate the most descriptive workload metrics. To start, OtterTune collects all previous runtime metric data for each workload previously seen and puts the information into a matrix X which represents a single workload, where each row represents a metric and each column is a vector representing a certain knob configuration.

$$X = \begin{array}{cc} \begin{array}{ccc} k_1 & \quad & k_i \end{array} & \\ \left[\begin{array}{ccc} x_{11} & \cdots & x_{1j} \\ \vdots & \ddots & \\ x_{i1} & & x_{ij} \end{array}\right] & \begin{array}{c} m_1 \\ \\ m_j \end{array} \end{array}$$

Figure 3.1. The input matrix to factor analysis which represents a single workload. Here each k is a vector representing a single knob configuration, and each m is a numerical metric. x_{ij} represents the value of metric m with configuration k.

3.2.2 Factor Analysis

OtterTune's source code implementation uses the SKlearn library for factor analysis [7]. It assumes that the data can be described as a linear transformation of lower dimensional latent factors, with some additional Gaussian noise. The output of the factor analysis step is another matrix U, which again has each row representing a numerical metric, however this time each column represents a factor found in the factor analysis step.

$$U = \begin{array}{c c} \begin{array}{c c c} f_1 & & f_i \end{array} & \\ \begin{bmatrix} u_{11} & \cdots & u_{1j} \\ \vdots & \ddots & \\ u_{i1} & & u_{ij} \end{bmatrix} & \begin{array}{c} m_1 \\ \\ m_j \end{array} \end{array}$$

Figure 3.2. The output matrix from factor analysis. Here each f is a factor found in the factor analysis phase, and each m is a numerical workload metric. u_{ij} represents the coefficient of metric m for the factor f.

3.2.3 K-Means Clustering

The next phase of pruning redundant metrics is K-Means clustering, which clusters closely related metrics together and selects one metric from each cluster. This phase starts by scatter-plotting data from matrix U. Each metric in U is plotted using the elements of its row (the coefficients to each of the factors) as coordinates. Two metrics will be close together if they have similar coefficients, which indicates redundancy and signals that these metrics can be pruned.

Following this initial scatter-plot and removal step, a heuristic using the gap statistic is used to choose an ideal number of clusters k. K-means clustering is applied, and then one metric from each cluster is retained while the rest are pruned. The final output from this step is a list of pruned metrics that best characterize the given workload.

3.2.4 Workload Mapping

Now that OtterTune has a list of the most relevant metrics to characterize a workload, these metrics can be used to calculate a similarity score between a new workload and existing workloads. When a new workload is received OtterTune looks at the workload summary file which tells which database type and version it is. Using this information OtterTune retrieves a set of N matrices, where N is the number of relevant metrics found in the workload characteristics step. Each matrix in this set has identical row and column labels, where each row is a workload that has been run on this DBMS version previously, and each column is a vector representing a certain knob configuration.

$$S = \left\{ \begin{array}{cc} \begin{array}{ccc} k_1 & \quad & k_i \end{array} & \\ \begin{bmatrix} x_{11} & \cdots & x_{1j} \\ \vdots & \ddots & \\ x_{i1} & & x_{ij} \end{bmatrix} & \begin{array}{c} w_1 \\ \\ w_j \end{array} \end{array}_{1}, \cdots, \begin{array}{cc} \begin{array}{ccc} k_1 & \quad & k_i \end{array} & \\ \begin{bmatrix} x_{11} & \cdots & x_{1j} \\ \vdots & \ddots & \\ x_{i1} & & x_{ij} \end{bmatrix} & \begin{array}{c} w_1 \\ \\ w_j \end{array} \end{array}_{n} \right\}$$

Figure 3.3. The set S of matrices for similarity score calculation. Here each k is a vector representing a knob configuration, and each w is a previously observed workload. x_{mij} is the value of metric m when executing workload i with knob configuration j.

Before a similarity score can be calculated, preprocessing must be done to make the data in S usable. First, there's the issue that each matrix in S may be very sparse, as not every workload is run on multiple knob configurations. To prevent this issue, a Gaussian Process Regression model is trained on each metric to predict the performance of that metric on a given knob configuration.

In addition, some of these metrics have much larger values than others, so to reduce issues from differing magnitudes OtterTune calculates deciles for each metric, bins the values according to their deciles, and then replaces the metric values with their bin numbers.

3.2.5 Similarity Score Calculation

Now that S is ready for workload mapping, each relevant metric from the target workload to be mapped is compared to each previous workload in S, and the Euclidean distance between them is calculated. Similarity scores for each workload are calculated by summing the distances across all previously seen instances of that workload. Because the goal in finding a similar workload is to minimize distance, the the workload with the lowest similarity score is selected as the most similar.

3.2.6 Tuning

Before OtterTune can recommend knob settings, it must choose which knobs to tune. PostgreSQL 9.6 contains 30 tunable knobs, but instead it uses an iterative approach to only select and tune the most impactful knobs on each tuning iteration. This allows for quicker convergence towards an optimal value. Ranking the DBMS knobs by impact is done using linear regression with scikit-learn's Lasso library [7]. Prior to ranking the knobs preprocessing steps are performed that use onehot encoding to encode enumerated knob values, and standardizes the knob values so that the results aren't skewed.

Finally, using previously observed tuning configurations from the mapped workload and previous tuning attempts from the target database, a Gaussian Process Regression model is trained to estimate the value of the tuning target objective metric under different knob configurations [10]. To find the recommended optimal configuration OtterTune creates an initialization set of starting points from previous configurations of the workload. These include top performing configurations from the current tuning session, and previously seen random configurations of knobs that were generated to create training data. Gradient descent finds a set of potential optimized configurations using the initialization set as starting points. The configuration in the potential set with the highest estimated improvement is selected as the final configuration recommendation. At configurations that have already been tested there is very little expected improvement upon the result value, since it's already been tested and running it again wouldn't change the results. In regions between known configurations expected improvement can increase depending on the training data. As tuning continues more unknown regions will be tested, reducing the expected improvement values. Each time, OtterTune will always select either an unknown configuration that it thinks has potential, or a well known configuration that it doesn't believe can be significantly improved upon.

CHAPTER 4

METHODS

My approach to this project can be broken down into three main phases: training data collection, implementing the OtterTune pipeline, and experimental evaluation. In this section I'll describe the methodology behind collecting training data and implementing the OtterTune system.

4.1 Data Collection

As with any machine learning project, OtterTune is only as good as its data. In order to work effectively OtterTune requires a robust repository workload metrics under a variety of randomly generated knob configurations. This requires building a data collection pipeline that generates a postgresql.conf file with random knob settings, starts a database server, runs the workload, collects metrics, then stops the server. This process must be repeated thousands of times to generate a large enough training set.

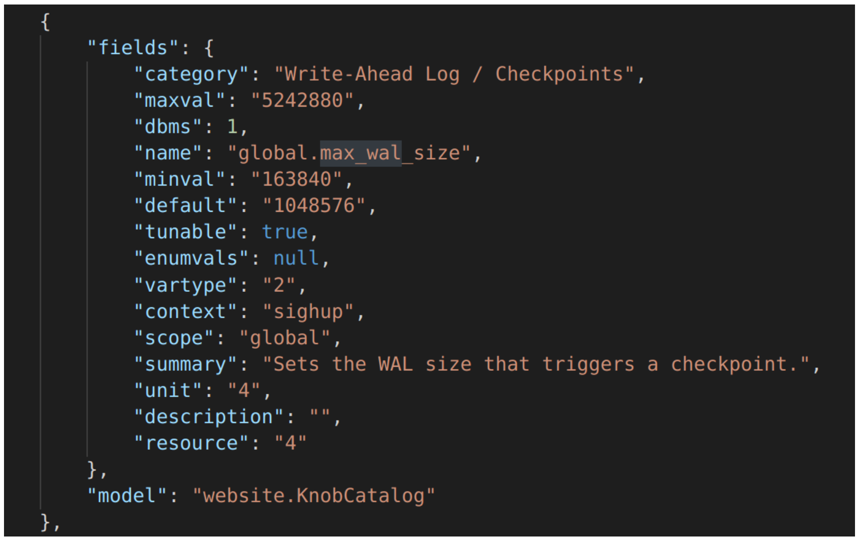
4.1.1 Challenges

Generating random knob configurations took a substantial amount of effort, because in order to be useful all knobs must be within a realistic range of values, while also testing unusual configurations and maintaining awareness of knob relationships. One major challenge is that there is a wide range of knob value types, including strings, integers, Boolean variables, enumerated variables, or real numbers. Additionally, knobs that manage memory may need to be incremented by 8kB at a time [1], while other memory knobs have no such limitations. Complicating the matter, many knobs are hardware dependent, and nowhere is there a concrete guide on what constitutes a reasonable range for all knobs. PostgreSQL offers a list of valid ranges for each knob, but these ranges are often unhelpful. For example, PostgreSQL has a *seq_page_cost* with a listed range between 0 and $1.79769e + 308$, but reasonable values in practice are between .1 and 1.5 [6].

Additionally, knobs cannot be tuned in isolation from one another. Many knobs interact, and so while both may have reasonable values individually the final impact is disastrous. One example of this is in the write-ahead logging configurations, where the valid ranges for *min_wal_size* and *max_wal_size* have substantial overlap. If both settings had values randomly selected from their reasonable ranges the max could be lower than the min, which would lead to issues [2]. A less obvious example of connected knobs is *maintenance_work_mem* and *autovacuum_max_workers*, or *work_mem* and *max_parallel_workers_per_gather*. In both cases, each worker is allocated the set amount of memory, which means that if both knobs are randomly assigned high values within their ranges memory is quickly used up. In the case of *work_mem* and *max_parallel_workers_per_gather*, a parallel query with 4 workers can use up to 5 times as much memory and other resources as a query with no parallel workers [1].

4.1.2 Approach

To keep track of all the requirements for each knob, a detailed knobs.json file as shown in **Figure 4.1** was used that contained metadata about each PostgreSQL knob such as data type, range, default value, what resource it manages, and if it's tunable. The base .json file was taken from the original OtterTune project files, so that all non-hardware dependent knobs would stay true to the original paper. From this base file I manually researched each tunable knob to narrow down the value range to only include reasonable numbers. This included configuring memory settings to fit the machines running the data collection tests, and evaluating knob interactions to minimize conflicting value ranges. A parser then uses this configuration data to generate a new postgresql.conf file, which is then uploaded to the database server.



```
{
    "fields": {
        "category": "Write-Ahead Log / Checkpoints",
        "maxval": "5242880",
        "dbms": 1,
        "name": "global.max_wal_size",
        "minval": "163840",
        "default": "1048576",
        "tunable": true,
        "enumvals": null,
        "vartype": "2",
        "context": "sighup",
        "scope": "global",
        "summary": "Sets the WAL size that triggers a checkpoint.",
        "unit": "4",
        "description": "",
        "resource": "4"
    },
    "model": "website.KnobCatalog"
},
```

Figure 4.1. Sample knob metadata entry from knobs.json

The database server is restarted to use the new configuration files, and oltp-bench [5] is used to execute a benchmark as a sample workload. Following the completion of the workload or the end of the five minute observation interval, whichever comes first, the metric data and knob configurations are gathered and uploaded to the processing server, along with a summary file with info about the workload execution. This process is repeated 500 times for each workload. For the purposes of this project I chose 8 different workloads to initialize my data repository with. To stay true to the approach of the original OtterTune paper I selected 5 variations of TPC-C [9] and 3 variations of smallbank, and used those workloads to populate the repository. I selected these benchmarks as they are both widely used, and can be easily configured to represent a variety of different workload characteristics.

4.2 The OtterTune Architecture

Following the construction of the data repository, I set up the full OtterTune pipeline to evaluate tuning performance on target workloads. This implementation had two main goals. First, since my data collection systems varied from the original OtterTune data collection formats, it was important to implement my own workload data parser so that I could ensure that the data repository was fully functional. To achieve the first goal I used sample OtterTune workload data from the Carnegie Mellon Database Group to ensure that my repository met the requirements for the later analysis steps. Second, this implementation needed to be lightweight, and utilize exiting OtterTune code [4] where possible to minimize the chance of programming errors skewing performance results. The existing OtterTune code is highly modular, so I broke down the different elements of the processing and tuning pipeline to isolate only the relevant code for tuning PostgreSQL 9.6. At the conclusion of this process I had an efficient and lightweight version of OtterTune that stayed true to the original approach while eliminating unnecessary code.

CHAPTER 5

RESULTS

The goal in implementing and testing OtterTune's mapping and tuning performance was to demonstrate a proof-of-concept for the characterization, mapping, and tuning approach taken in the original paper. To achieve this I collected data on three core tests. First, the system's workload mapping abilities are evaluated by calculating the probability of correctly mapping a workload across various sample sizes. Following the analysis of workload mapping performance I tuned an new variant of the TPC-C workload, and a completely unseen YCSB workload. The data suggests that impressive improvements in throughput are possible with a robust repository of previous workloads, but the tuning performance degrades sharply for unseen workloads.

5.1 Mapping Evaluation

To evaluate the accuracy of OtterTune's workload mapping system I used cross validation to calculate the chances that n randomly selected samples from a given workload would be accurately mapped to the same workload. For each sample size n between $1 - 10$, n samples were randomly selected and removed from each workload. Those n samples were then characterized and mapped back to the rest of the training data, and the selected mapping was recorded. This was repeated 15 times for each workload at each sample size,across all workloads in the repository. The result is percentage of trials in which a sample set of n tuning results from workload w was correctly mapped to other instances of workload w.

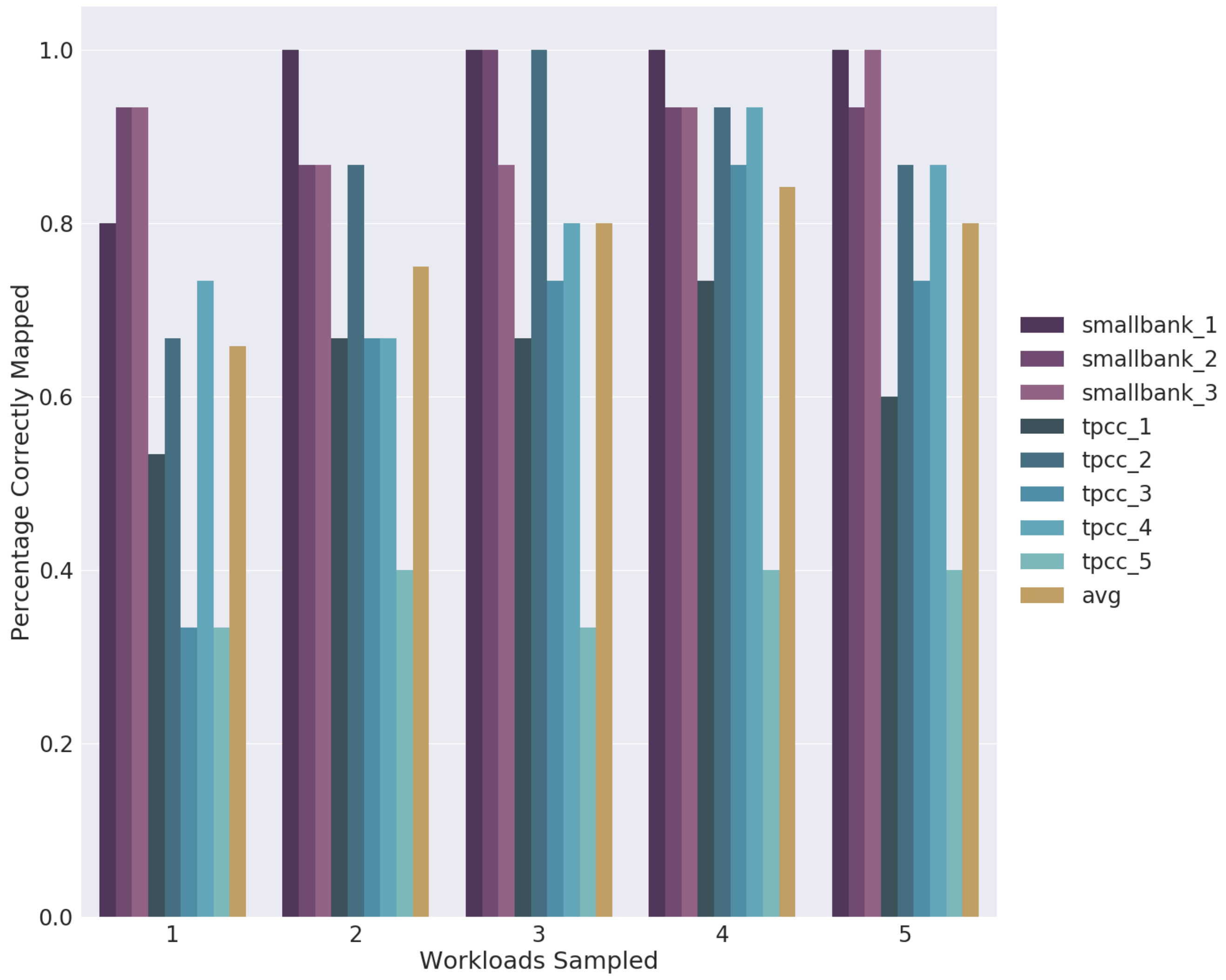


Figure 5.1. Workload mapping performance

The results of this mapping evaluation, shown in **Figure 5.1**, demonstrate that while the mapping accuracy is initially very good, with an average of 70% of workloads being correctly mapped, the mapping accuracy increases with sample size. The accuracy of the mapping results varies across workload type, with workloads that ran the smallbank benchmark generally mapping more accurately than TPC-C workloads. This is expected behavior, since there were more examples of TPC-C workloads in the training set there were often two workloads with very close similarity scores, whereas with only three smallbank examples in the training repository there were larger differences between each workload, which meant better distinctions between workload mappings. The mapping results for the tpcc_5 workload are of particular interest, because unlike errors in the other TPC-C workloads, this workload's mapping errors weren't caused by close differences in other TPC-C workloads. In fact, in every case where tpcc_5 was inaccurately mapped, it selected smallbank_1 as the most similar workload. Nothing substantially differentiates tpcc_5 from the other TPC-C workloads, it just seems that the TPC-C configurations for tpcc_5 lead to very similar workload metrics to smallbank_1. This is an interesting example of how OtterTune's decision to ignore logical features, such as schema or query structure, can lead to errors in workload mapping. An increase in the number of tpcc_5 workloads randomly sampled for mapping doesn't seem to have a significant effect on the mapping accuracy. This issue could become further magnified in an enterprise instance of OtterTune with a much larger variety of workloads to map to, which indicates a need to consider other logical features as well when mapping a workload.

Overall, the results from the mapping evaluation are encouraging. OtterTune heavily relies on accurately identifying similar workloads to learn from while tuning, and these results indicate that initial mappings from a single observation period are generally fairly accurate, and increase in accuracy as more workload tuning results are considered. This evaluation shows that there is potential room for improvement in mapping accuracy if logical features of the workload are considered in addition to system metrics, but shows that in the majority of cases OtterTune's mapping structure holds.

5.2 Tuning Evaluation

To evaluate the tuning performance of OtterTune two workloads were selected and tested. The first was a TPC-C workload executed on a database 3 times larger than the TPC-C database the training instances used. This workload represents an ideal case, where the workload type is already well represented in OtterTune's repository but the system metrics will vary due to the increased database size. The second workload selected was a YCSB workload with an equal balance of read and write queries. This workload was chosen because it's a departure from the data in the repository, and represents a completely unknown workload type.

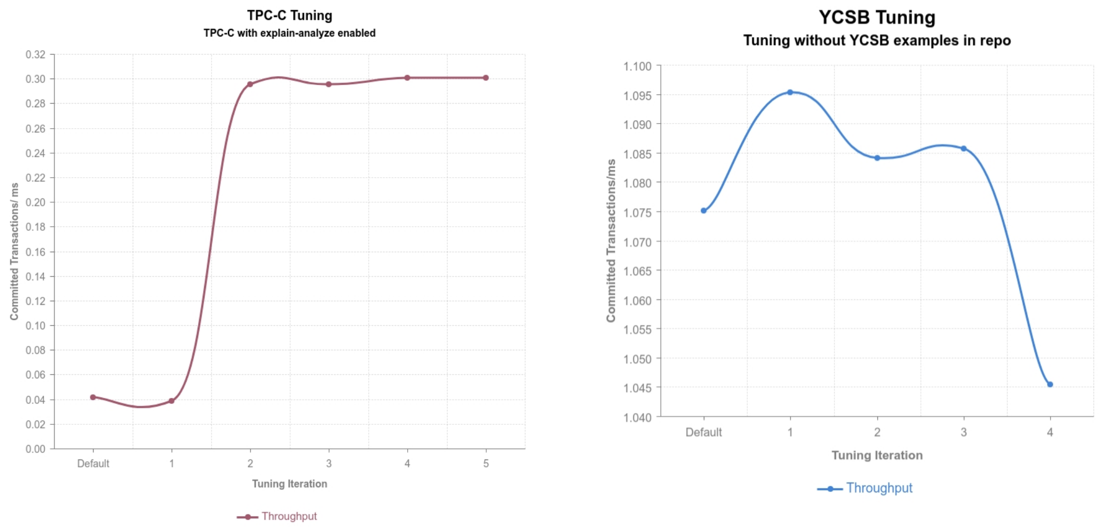


Figure 5.2. Throughput on target workloads during tuning session

As is evident in **Figure 5.2**, OtterTune's tuning performance varies widely between the two workloads. In the case of TPC-C, throughput increased by over 7x within three tuning iterations, and held steady at the higher performance rate. In stark contrast YCSB's throughput initially increased before dropping slightly, and ultimately showed no significant improvement over the default configuration. These two workloads powerfully demonstrate the impact of the tuning repository on recommended knob configurations. In the case of TPC-C, there were several thousand tuning results already stored in the repository, and so even though the tested TPC-C had a different number of simulated warehouses and terminals the workload itself was very well understood. Because of this wealth of data it only took two tuning iterations before throughput increased to over 7x its initial value with default knobs. For the YCSB workload, there was no tuning data in the repository that closely resembled the workload. This meant that it was impossible for OtterTune to accurately identify a workload to learn from, and without useful previously seen tuning data there was no way to improve performance. This demonstrates the importance of a large repository of training data for workloads, which is acknowledged in the original OtterTune paper.

5.3 Conclusions from Evaluations

The results from the mapping evaluations and the tuning experiments show that while OtterTune has incredible potential as a database tuning service, that potential is contingent on several factors. First, there must be enough training data for new workloads to be accurately mapped. The training data requirements are made more difficult by the fact that training data is only useful for one hardware configuration and DBMS version pair, and so building a general use tuning service would require a large range of workloads executed on huge numbers of machines, under many commonly used DBMS versions. These requirements make OtterTune perfectly suited for use in a large scale cloud computing environment, like AWS, Azure, or Google cloud, where there are a finite number of hardware configurations and a large amount of databases that could potentially need to be tuned.

The results of the mapping evaluation also show that there is potential for mapping errors when system metrics are similar, but the workloads are very different. In most cases this issue would be mitigated by a larger variety of workloads to choose from or by gathering more tuning data from the target workload as tuning is performed. However, in edge cases where the system metrics may closely align but the logical structure of the workload differs, inaccuracies in mapping would mean that a workload is tuned from data that isn't representative of its actual performance. This could potentially cause initial bad knob configuration recommendations and slow down tuning until enough data is gather to improve mapping accuracy. In some cases, more data might not improve mapping, and tuning could become impossible. These issues may be mitigated by including data about the query structure in the mapping process, to differentiate between very different workloads with similar system metrics.

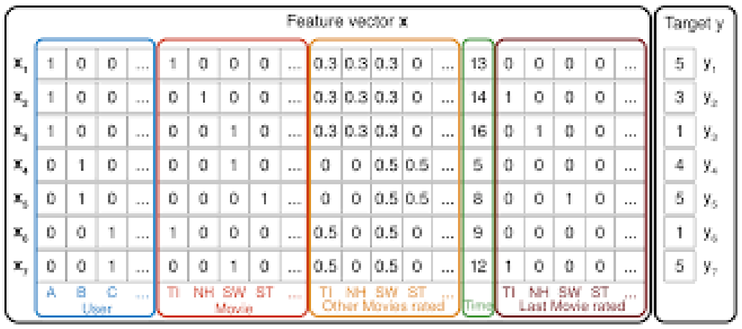
CHAPTER 6

FUTURE WORK

OtterTune takes an innovative approach to tuning in that it is the first major database tuner to learn from and reuse previous configurations, but it still is limited in that it only compares workloads across system metrics. As demonstrated in **Figure 5.1** there is potential for very different workloads to be inaccurately mapped if the logical structure of the database is ignored during tuning. One interesting future direction for automated database tuning would be to incorporate the logical structure of the database as a feature when mapping, or to include details of the query plan in mapping workloads. Plans for future work involving OtterTune include vectorizing query plans and using them as a feature in a Factorization Machine to generate more accurate workload mapping and tuning results.

6.1 Factorization Machines

Factorization Machines [8] are a supervised learning technique that is mixture of Matrix Factorization and Support Vector Machines. They are recognized as being highly efficient for classification and regression problems on large sparse datasets, and are commonly used for prediction of user behavior, specifically click prediction [8]. Factorization Machines can also be used to generate recommendations, such as generating movie recommendations based on data about other movies that a user has seen. A diagram of a Factorization Machine setup for movie recommendations is seen in **Figure 6.1**. Another appealing quality of Factorization Machines is that they have a linear complexity, and do not rely on support vectors like SVMs, which makes them an efficient way to make predictions [8].



	Feature vector x																			Target y	
x_1	1	0	0	...	1	0	0	0	...	0.3	0.3	0.3	0	...	13	0	0	0	0	...	5	y_1
x_2	1	0	0	...	0	1	0	0	...	0.3	0.3	0.3	0	...	14	1	0	0	0	...	3	y_2
x_3	1	0	0	...	0	0	1	0	...	0.3	0.3	0.3	0	...	16	0	1	0	0	...	1	y_3
x_4	0	1	0	...	0	0	1	0	...	0	0	0.5	0.5	...	5	0	0	0	0	...	4	y_4
x_5	0	1	0	...	0	0	0	1	...	0	0	0.5	0.5	...	8	0	0	1	0	...	5	y_5
x_6	0	0	1	...	1	0	0	0	...	0.5	0	0.5	0	...	9	0	0	0	0	...	1	y_6
x_7	0	0	1	...	0	0	1	0	...	0.5	0	0.5	0	...	12	1	0	0	0	...	5	y_7
	A	B	C	...	TI	NH	SW	ST	...	TI	NH	SW	ST	...	Time	TI	NH	SW	ST	...		
	User				Movie					Other Movies rated						Last Movie rated						

Figure 6.1. An example of a factorization machine for generating movie recommendations. [8]

Because Factorization machines perform well in user behavior prediction, we believe that they might have applications in predicting workload behavior. Specifically, we believe that there is potential for Factorization machines to be able to predict the performance of a workload under a given knob configuration, given information about previous workloads and their performance under different configurations. An example of how a Factorization Machine might be arranged for this purpose is given in **Figure 6.2**. Using a Factorization Machine in this way wouldn't solve the problem of generating new knob configurations, but it could serve as an additional tool for evaluating knob recommendations before they're tested. Additionally, by changing the inputs to the feature vector, Factorization Machines could be used to give better recommendations for workload mapping.

Feature Vector **x**				Target **y**
Workload Name	Knob Config	Query Plan	Metric Data	Goal Metric
tpcc_1	K_1	Q_1: <HashJoin, …>	M_1	.30
tpcc_1	K_2	Q_2	M_2	.09
….	K_n	Q_n	M_n	G_n
smallbank_1	K_{n+1}	Q_{n+1}	M_{n+1}	G_{n+1}

Figure 6.2. An example of a factorization machine for predicting performance of a workload under different knob configurations.

CHAPTER 7

CONCLUSIONS

In this thesis I presented an explanation and analysis of OtterTune's approach to automated database tuning, as well as the background and motivation for the OtterTune Project. I implemented a lightweight version of OtterTune using much of the same analysis code, but with a newly developed training data generator and parser. Evaluations of mapping workloads using this new tuner showed that workload mapping using the OtterTune approach is between 70-85% accurate, but that certain workloads had significantly lower accuracy of around 40% while others had near perfect accuracy. In general, workload mapping accuracy increased when more samples from a workload were used for mapping and characterization, but there are potential improvements to be made in mapping by considering logical features of the workload such as query plan or table schema. Evaluations of tuning performance on well known workloads showed significant increases in throughput, up to a 7x increase from the default knob configuration values in just three tuning iterations. For workloads that OtterTune hadn't previously seen, and that behaved differently from all training workloads, didn't see significant changes in performance during tuning. The concept of a factorization machine based approach to workload characterization and database tuning was introduced, which will be the focus of future work.

Bibliography

[1] *19.4. Resource Consumption*. URL: `https://www.postgresql.org/docs/9.6/runtime-config-resource.html`.

[2] *19.5. Write Ahead Log*. URL: `https://www.postgresql.org/docs/9.6/runtime-config-wal.html`.

[3] M. Calzarossa and G. Serazzi. "Workload characterization: a survey". In: 81 (Aug. 1993), pp. 1136–1150.

[4] cmu-db. *cmu-db/ottertune*. Nov. 2018. URL: `https://github.com/cmu-db/ottertune`.

[5] Djellel Eddine Difallah et al. "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases". In: *PVLDB* 7.4 (2013), pp. 277–288. URL: `http://www.cs.cmu.edu/~pavlo/static/papers/oltpbench-vldb.pdf`.

[6] EnterpriseDB. *3.1.3.1 Top Performance Related Parameters*. URL: `https://www.enterprisedb.com/docs/en/10.0/EPAS_Guide_v10/EDB_Postgres_Advanced_Server_Guide.1.24.html`.

[7] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[8] Steffen Rendle. "Factorization Machines". In: *Proceedings of the 2010 IEEE International Conference on Data Mining*. ICDM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 995–1000. ISBN: 978-0-7695-4256-0. DOI: `10.1109/ICDM.2010.127`. URL: `http://dx.doi.org/10.1109/ICDM.2010.127`.

[9] *TPC-C*. URL: `http://www.tpc.org/tpcc/default.asp`.

[10] Dana Van Aken et al. "Automatic Database Management System Tuning Through Large-scale Machine Learning". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. 2017, pp. 1009–1024. URL: `http://db.cs.cmu.edu/papers/2017/p1009-van-aken.pdf`.

[11] Philip Yu et al. "Workload characterization of relation database environments". In: 18 (May 1992), pp. 347–355.

[12] Yuqing Zhu et al. "BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning". In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: ACM, 2017, pp. 338–350. ISBN: 978-1-4503-5028-0. DOI: `10.1145/3127479.3128605`. URL: `http://doi.acm.org/10.1145/3127479.3128605`.