Automatic Database Management System Tuning Through Large-scale Machine Learning

Dana Van Aken
Carnegie Mellon University
dvanaken@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Geoffrey J. Gordon
Carnegie Mellon University
ggordon@cs.cmu.edu

Bohan Zhang
Peking University
bohan@pku.edu.cn

ABSTRACT

Database management system (DBMS) configuration tuning is an essential aspect of any data-intensive application effort. But this is historically a difficult task because DBMSs have hundreds of configuration "knobs" that control everything in the system, such as the amount of memory to use for caches and how often data is written to storage. The problem with these knobs is that they are not standardized (i.e., two DBMSs use a different name for the same knob), not independent (i.e., changing one knob can impact others), and not universal (i.e., what works for one application may be sub-optimal for another). Worse, information about the effects of the knobs typically comes only from (expensive) experience.

To overcome these challenges, we present an automated approach that leverages past experience and collects new information to tune DBMS configurations: we use a combination of supervised and unsupervised machine learning methods to (1) select the most impactful knobs, (2) map unseen database workloads to previous workloads from which we can transfer experience, and (3) recommend knob settings. We implemented our techniques in a new tool called OtterTune and tested it on three DBMSs. Our evaluation shows that OtterTune recommends configurations that are as good as or better than ones generated by existing tools or a human expert.

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 [35, 25]. DBMSs are the critical component of data-intensive ("Big Data") applications [46]. 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 [24]. 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 [22, 47, 36]. 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. As an example, the default MySQL configuration in 2016 assumes that it is deployed on a machine that only has 160 MB of RAM [1].

Given this, many organizations resort to hiring expensive experts to configure the system's knobs for the expected workload. But as databases and applications grow in both size and complexity, optimizing a DBMS to meet the needs of an application has surpassed the abilities of humans [11]. This is because the correct configuration of a DBMS is highly dependent on a number of factors that are beyond what humans can reason about.

Previous attempts at automatic DBMS configuration tools have certain deficiencies that make them inadequate for general purpose database applications. Many of these tuning tools were created by vendors, and thus they only support that particular company's DBMS [22, 33, 37]. The small number of tuning tools that do support multiple DBMSs still require manual steps, such as having the DBA (1) deploy a second copy of the database [24], (2) map dependencies between knobs [49], or (3) guide the training process [58]. All of these tools also examine each DBMS deployment independently and thus are unable to apply knowledge gained from previous tuning efforts. This is inefficient because each tuning effort can take a long time and use a lot of resources.

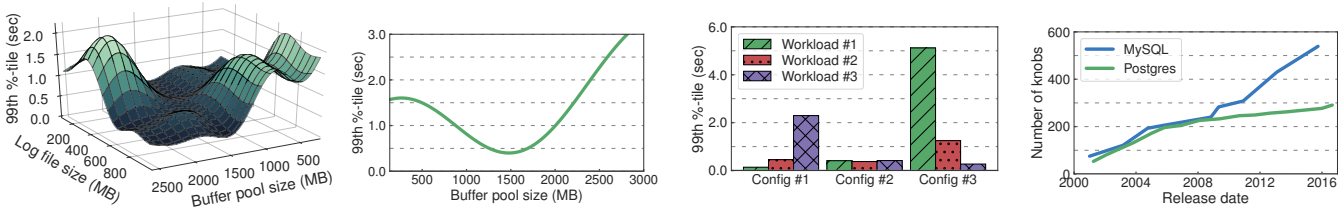
In this paper, we present a technique to reuse training data gathered from previous sessions to tune new DBMS deployments. The crux of our approach is to train machine learning (ML) models from measurements collected from these previous tunings, and use the models to (1) select the most important knobs, (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 experience reduces the amount of time and resources it takes to tune a DBMS for a new application. To evaluate our work, we implemented our techniques using Google TensorFlow [50] and Python's `scikit-learn` [39] in a tuning tool, called **OtterTune**, and performed experiments for two OLTP DBMSs (MySQL, Postgres) and one OLAP DBMS (Vector). Our results show that OtterTune produces a DBMS configuration for these workloads that achieves 58–94% lower latency compared to their default settings or configurations generated by other tuning advisors. We also show that OtterTune generates configurations in under 60 min that are within 94% of ones created by expert DBAs.

The remainder of this paper is organized as follows. Sect. 2 begins with a discussion of the challenges in database tuning. We then provide an overview of our approach in Sect. 3, followed by a description of our techniques for collecting DBMS metrics in Sect. 4,

SIGMOD'17, May 14 - 19, 2017, Chicago, IL, USA

ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3064029



(a) Dependencies **(b)** Continuous Settings **(c)** Non-Reusable Configurations **(d)** Tuning Complexity

Figure 1: Motivating Examples – Figs. 1a to 1c show performance measurements for the YCSB workload running on MySQL (v5.6) using different configuration settings. Fig. 1d shows the number of tunable knobs provided in MySQL and Postgres releases over time.

identifying the knobs that have the most impact in Sect. 5, and recommending settings in Sect. 6. In Sect. 7, we present our experimental evaluation. Lastly, we conclude with related work in Sect. 8.

2. MOTIVATION

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 [36].

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 [47]. Such a "trial-and-error" approach to DBMS tuning is tedious, expensive, and inefficient because (1) many of the knobs are not independent [24], (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.

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. We present the details of our operating environment for these experiments in Sect. 7.

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 [49]. 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 Fig. 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 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 Fig. 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). Fig. 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 Fig. 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 [43], and many DBAs spend nearly 25% of their time on tuning [21].

A better approach than examining each database application separately is to use an automated tool that leverages the knowledge gained from one application to assist in the tuning of others.

3. SYSTEM OVERVIEW

We now present our automatic database tuning tool that overcomes the problems that we described above. OtterTune is a tuning service that works with any DBMS. 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.

[1] MySQL Knob: `innodb_buffer_pool_size`

[2] MySQL Knob: `innodb_log_file_size`

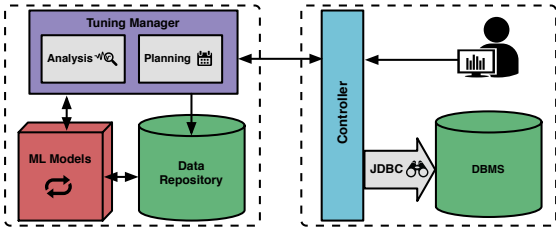


Figure 2: OtterTune Architecture – An overview of the components in the OtterTune system. The controller connects to the DBMS and collects information about the performance of the system. This information is then sent to the tuning manager where it is stored in its repository. It then builds models that are used to select an optimal configuration for the DBMS.

Fig. 2 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 information from the DBMS using a standard API (e.g., JDBC), installs new configurations, and collects performance measurements.

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 v9.3 data is separate from Postgres v9.4). This prevents OtterTune from tuning knobs 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.

3.1 Example

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). We defer the problem of automatically determining the hardware capabilities of a DBMS deployment to future work.

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 additional DBMS-specific internal metrics. Examples include MySQL's counters for pages read to or written from disk. Other DBMSs provide similar metrics, but OtterTune does not require any information from the DBA about their meaning, whether they are indicative of good or bad performance, or whether metrics with different names mean the same thing in different DBMSs. We discuss this metric collection in further detail in Sect. 4, along with our approach to rank the DBMS's knobs by their importance in Sect. 5.

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. As we discuss in Sect. 6, selecting this next configuration is the critical task in OtterTune. There are two distinct steps that take place after each observation period that determine what kind of configuration the system will recommend. In the first step, OtterTune tries to "understand" the target workload and map it to a workload for the same DBMS and hardware profile that it has seen (and tuned) in the past. Once the tuning manager has found the best match using the data that it has collected so far, it then starts the second step where it recommends a knob configuration that is specifically designed to improve the target objective for the current workload, DBMS, and hardware. 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.

3.2 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 [24].

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 [16] that the DBA can utilize for this purpose. As

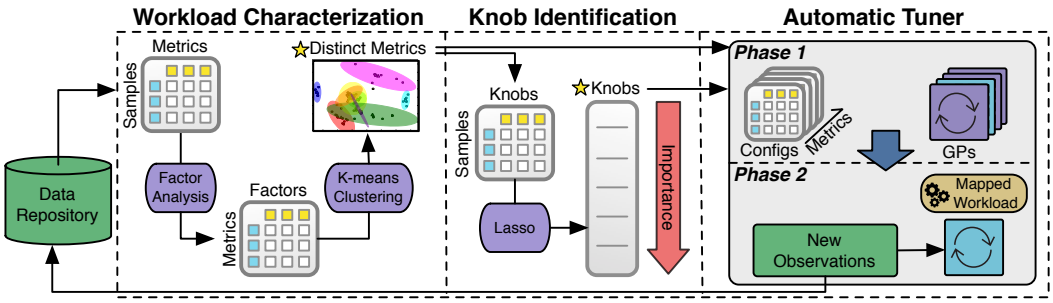


Figure 3: OtterTune Machine Learning Pipeline – This diagram shows the processing path of data in OtterTune. All previous observations reside in its repository. This data is first then passed into the **Workload Characterization** (Sect. 4) component that identifies the most distinguishing DBMS metrics. Next, the **Knob Identification** (Sect. 5) component generates a ranked list of the most important knobs. All of this information then fed into the **Automatic Tuner** (Sect. 6) component where it maps the target DBMS's workload to a previously seen workload and generates better configurations.

discussed in Appendix C, we plan to investigate how to apply these same techniques to tune the database's physical design.

In the sections that follow, we describe how OtterTune collects runtime metrics from the DBMS during a tuning session. It then creates models from this data that allow it to (1) select the most impactful knobs, (2) map previously unseen database workloads to known workloads, and (3) recommend knob settings. We start with discussing how to identify which of the metrics gathered by the tuning tool best characterize an application's workload. An overview of this entire process is shown in Fig. 3.

4. 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 [15], 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 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.

4.1 Statistics Collection

OtterTune's controller supports a modular architecture that enables it to perform the appropriate operations for different DBMSs to collect their runtime statistics. At the beginning of each observation period, the controller first resets all of the statistics for the target DBMS. It then retrieves the new statistics data at the end of the period. Since at this point OtterTune does not know which metrics are actually useful, it collects every numeric metric that the DBMS makes available and stores it as a key/value pair in its repository.

The main challenge in this collection process is how to represent metrics for sub-elements of the DBMS and database. Some systems, like MySQL, only report aggregate 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). 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.

4.2 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[3] and pages.[4] The two metrics are the same measurement just in different units, thus it is unnecessary to consider

[3] MySQL Metric: `innodb_data_read`
[4] MySQL Metric: `innodb_pages_read`

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[5] moves almost in unison with the metric that measures the number of blocks read from the buffer for indexes.[6]

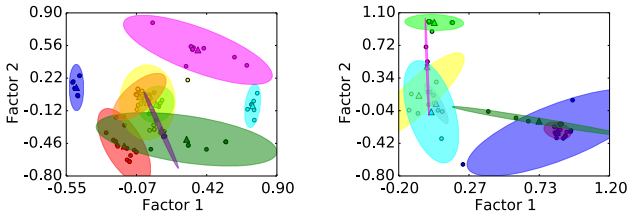
We use two well-studied techniques for this pruning. The first is a dimensionality reduction technique, called *factor analysis* (FA) [5], that transforms the (potentially) high dimensional DBMS metric data into lower dimensional data. We then use the second technique, called *k-means* [6], to cluster 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 [31, 30]. 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 [40] 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 compared this heuristic with other techniques [55, 48] 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 Sect. 7.

The visualization in Fig. 4 shows a two-dimensional projection of the scatter-plot and the metric clusters in MySQL and Postgres. In the MySQL clusters in Fig. 4a, 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[7], the amount of data read[8], and the time spent waiting for resources[9] are all grouped



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

Figure 4: 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.

into the same cluster. In Fig. 4b 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[10,11] and indexes.[12,13]

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.

5. 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* [54], 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.

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 ∼20 min and consumes ∼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 Sect. 6.

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.

[5] Postgres Metric: `pg_stat_database.tup_updated`
[6] Postgres Metric: `pg_statio_user_tables.idx_blks_hit`
[7] MySQL Metrics: `innodb_data_written`, `innodb_buffer_pool_write_requests`
[8] MySQL Metrics: `innodb_rows_read`, `bytes_sent`
[9] MySQL Metrics: `innodb_log_waits`, `innodb_row_lock_time_max`
[10] Postgres Metric: `pg_stat_bgwriter.buffers_clean`
[11] Postgres Metric: `pg_stat_bgwriter.maxwritten_clean`
[12] Postgres Metric: `pg_statio_user_tables.idx_blks_read`
[13] Postgres Metric: `pg_statio_user_indexes.idx_blks_read`

5.1 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 [14]. 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 [54, 26]. There is also both practical and theoretical work backing its effectiveness as a consistent feature selection algorithm [56, 57, 64, 9].

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* [29] 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.

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.

5.2 Dependencies

As we showed in Sect. 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.

5.3 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 [27, 20]. As we show in our evaluation in Sect. 7.3, this always produces better configurations than any static knob count.

6. 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.

6.1 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 Sect. 3). 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.

6.2 Step #2 – Configuration Recommendation

In the next step, OtterTune uses *Gaussian Process* (GP) regression [42] 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) [32, 45]. 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* [29] 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 Sects. 5.1 and 6.1), 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.

7. 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.

7.1 Workloads

For these experiments, we use workloads from the OLTP-Bench testbed that differ in complexity and system demands [3, 23]:

YCSB: The Yahoo! Cloud Serving Benchmark (YCSB) [18] 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 [51]. 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. 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 [52]. 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.

7.2 Training Data Collection

As discussed in Sect. 3, 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 [12]. 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.

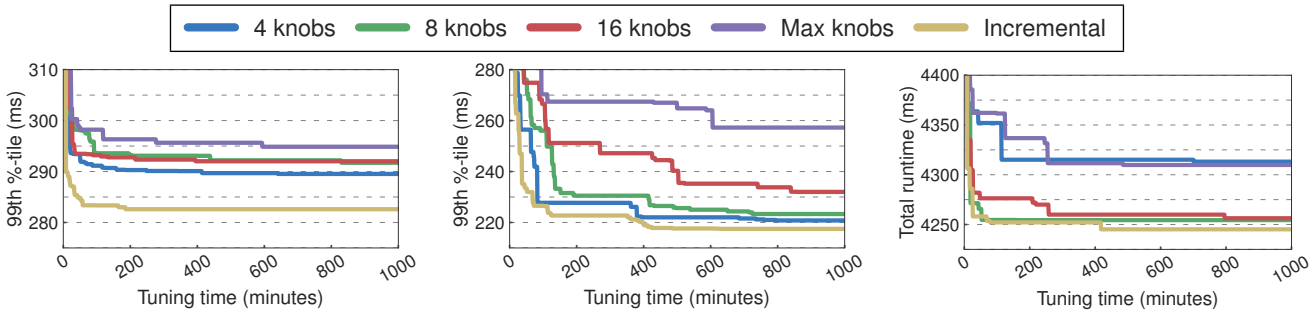
We executed a total of over 30k trials per DBMS using these different workload and knob configurations. Each of these trials is treated like an observation period in OtterTune, thus the system collects both the external metrics (i.e., throughput, latency) and internal metrics (e.g., pages read/written) 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.

7.3 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 Sect. 5.3 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 Sect. 5. 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.



(a) MySQL (TPC-C) **(b)** Postgres (TPC-C) **(c)** Vector (TPC-H)

Figure 5: 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.

MySQL: The results in Fig. 5a 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, (see Fig. 1a), 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 Fig. 5b 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.

Vector: As shown in Fig. 5c, 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.

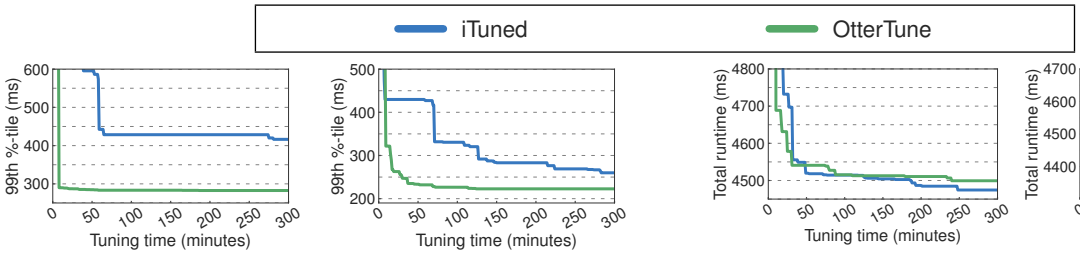
7.4 Tuning Evaluation

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 [24], 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. 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 Sect. 6.2.

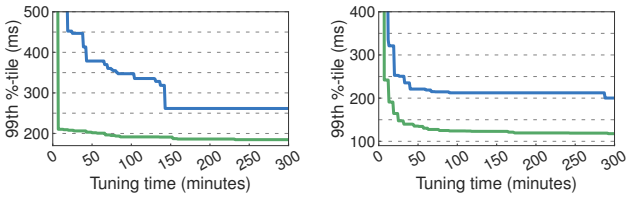
For this comparison, we use both the TPC-C and Wikipedia benchmarks for the OLTP DBMSs (MySQL and Postgres) and two variants of the TPC-H workload for the OLAP DBMS (Vector). OtterTune trains its GP models using the data from the most similar workload mixture determined in the last workload mapping stage. Both tuning tools use the incremental knob approach to decide how many knobs to tune during each observation period (see Sect. 5.3). 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 Fig. 6 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, Fig. 6a 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.



(a) MySQL **(b)** Postgres

Figure 6: Tuning Evaluation (TPC-C) – A comparison of the OLTP DBMSs for the TPC-C workload when using configurations generated by OtterTune and iTuned.



(a) MySQL **(b)** Postgres

Figure 7: Tuning Evaluation (Wikipedia) – A comparison of the OLTP DBMSs for the Wikipedia workload when using configurations generated by OtterTune and iTuned.

Wikipedia: We next compare the two tuning approaches on MySQL and Postgres using a more complex workload. Like with TPC-C, the results in Fig. 7 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. Fig. 8 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.

7.5 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 Sect. 6. 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 Sect. 6.1.
- **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 Sect. 6.2.

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

Figure 8: Tuning Evaluation (TPC-H) – Performance measurements for Vector running two sub-sets of the TPC-H workload using configurations generated by OtterTune and iTuned.

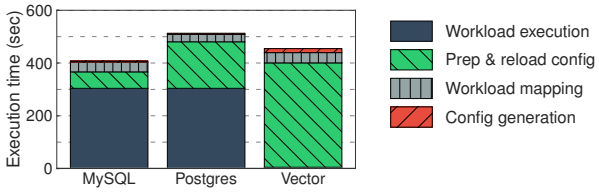


Figure 9: Execution Time Breakdown – The average amount of time that OtterTune spends in the parts of the system during an observation period.

The results in Fig. 9 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.

7.6 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.[14] 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 B.

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.,

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

OS kernel parameters). On the client instance, we provided them with a script to execute the workload for the 5 min observation period and a general log full of previously executed queries for that workload. The DBAs were permitted to restart the DBMS and/or the workload as many times as they wanted.

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

MySQL: Our first DBA is the premiere MySQL tuning and optimization expert from Lithuania with over 15 years of experience and also works at a well-known Internet company. They finished 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.

Fig. 10 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 Sect. 7.3 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 Fig. 10 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 Sect. 3.2, 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` [4]) is less sophisticated than the MySQL one in that it only uses pre-programmed rules that generate knob configurations for the target hardware and does not consider the DBMS's metrics. We found, however, that using the Postgres tuning tool was easier because it was based on the amount of RAM available in the system and some high-level characteristics about the target workload (e.g., OLTP vs. OLAP). It took 30 seconds to generate the configuration and we never had to restart the DBMS. It changed a total of eight knobs.

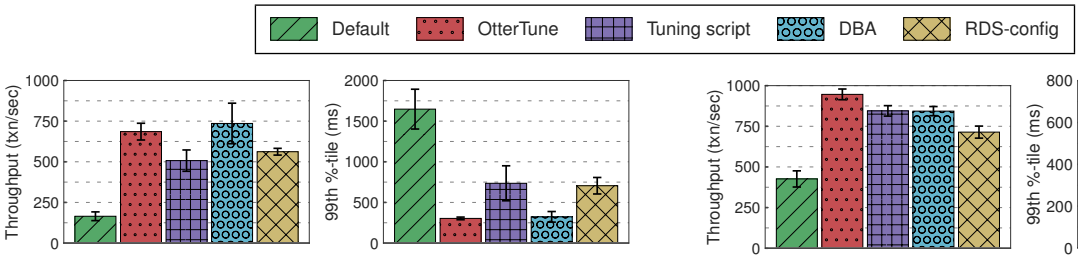
The latency measurements in Fig. 11b show that the configurations generated by OtterTune, the tuning tool, the DBA, and RDS all achieve similar improvements for TPC-C over Postgres' default settings. This is likely because of the overhead of network round-trips between the OLTP-Bench client and the DBMS. But the throughput measurements in Fig. 11 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. We leave this problem as a goal for future work.

8. RELATED WORK

Much of the previous work on automatic database tuning focused on choosing the best logical or physical design of a database [16, 65], such as selecting indexes [28, 17, 59], partitioning schemes [8, 38, 41, 19], or materialized views [7]. A physical design is the



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

Figure 10: Efficacy Comparison (MySQL) – Throughput and latency measurements for the TPC-C benchmark using the (1) default configuration, (2) OtterTune configuration, (3) tuning script configuration, (4) Lithuanian DBA configuration, and (5) Amazon RDS configuration.

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

Figure 11: 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.

configuration of the data that determines how queries will access it, whereas a knob configuration affects the DBMS's internals.

Others have looked at tuning a subset of DBMS knobs that have the most impact on performance [49, 21, 24]. Unlike physical database design tools, configuration tools cannot use the built-in cost models of DBMSs' query optimizers. Such models generate estimates based 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 [44]. As such, they are not able to properly capture bottlenecks in the same way that the DBMS's metrics can when executing concurrent queries or transactions [63].

In the 2000s, IBM released the DB2 Performance Wizard tool that asks the DBA questions about their application (e.g., whether the workload is OLTP or OLAP) and then provides knob settings based on their answers [34]. 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 heuristics to determine how to allocate the DBMS's memory [47, 53].

Oracle developed a similar internal monitoring system to identify bottlenecks due to misconfigurations in their DBMS's internal components [22, 33]. It then provides the DBA 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 [62, 10]. This approach has also been used with Microsoft's SQL Server [37]. But for both DBMSs, using this tool is still a manual process: the DBA provides the knob settings that they want to change and then the tool executes experiments with and without applying that change. The DBA 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 [13, 24, 60]. 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 [61]. This continues until the DBA either halts the process or the tool recognizes that additional performance gains from running more experiments are unlikely.

The COMFORT tool uses this on-line feedback approach to solve tuning issues like load control for locking [60]. It uses a technique from control theory that can adjust a single knob up or down at a time, but cannot uncover dependencies between multiple knobs.

Other work for BerkeleyDB uses *influence diagrams* to model probabilistic dependencies between configuration knobs [49]. This approach uses the knobs' conditional independences to infer expected outcomes of a particular DBMS configuration. The problem, however, is that these diagrams must be created manually by a domain expert and thus they only tune four knobs at a time.

The DBSherlock tool helps a DBA diagnose problems by comparing regions in the DBMS's performance time-series data where the system was slow with regions where it behaved normally [63].

The iTuned tool is the closest work that is related to our method [24]. It 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 explore the solution space and converge to a near-optimal configuration. The initial model is trained from data gathered from executing a set of experiments that were selected using an adaptive sampling technique. The iTuned tool can take up to seven hours to tune the DBMSs, whereas our results in Sect. 7.4 show that OtterTune achieves this in less than 60 min.

An alternative technique is to use linear and quadratic regression models to map knobs to performance [58]. Others have looked at using hill-climbing techniques for this problem [61]. 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.

All of these feedback-driven tools must determine which configuration knobs are likely to improve the DBMS's performance and their value ranges. This allows the tool to execute the minimum number of experiments that provide an approximate sampling of the entire search space [24]. The SARD tool generates a relative ranking of a DBMS's knobs based on their impact on performance using a technique called the *Plackett-Burman design* [21]. Others have developed statistical techniques for inferring from these experiments how to discretize the potential values for knobs [49].

9. CONCLUSION

We presented a technique for tuning DBMS knob configurations by reusing training data gathered from previous tuning sessions. Our approach uses a combination of supervised and unsupervised machine learning methods to (1) select the most impactful knobs, (2) map previously unseen database workloads to known workloads, and (3) recommend knob settings. Our results show that OtterTune produces configurations that achieve up to 94% lower latency compared to their default settings or configurations generated by other tuning advisors. We also show that OtterTune generates configurations in under 60 min that are comparable to ones created by human experts.

10. REFERENCES

[1] MySQL – InnoDB startup options and system variables. http://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html.

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

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

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

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

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

[7] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.

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

[9] 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.

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

[11] 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.

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

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

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

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

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

[17] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. In *VLDB*, pages 146–155, 1997.

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

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

[20] 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.

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

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

[23] 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.

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

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

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

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

[28] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for olap. In *ICDE*, pages 208–219, 1997.

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

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

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

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

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

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

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

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

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

[38] 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.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

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

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

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

[44] 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.

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

[46] 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.

[47] 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.

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

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

[50] M. A. *et al.* TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*, abs/1603.04467, 2016.

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

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

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

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

[55] 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.

[56] 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.

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

[58] 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.

[59] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 advisor: an optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000.

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

[61] 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.

[62] 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.

[63] D. Y. Yoon, N. Niu, and B. Mozafari. DBSherlock: a performance diagnostic tool for transactional databases. In *SIGMOD*, pages 1599–1614, 2016.

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

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

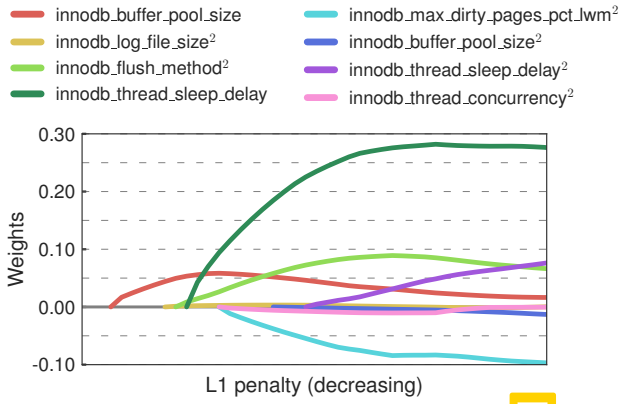


Figure 12: Lasso Path (MySQL)

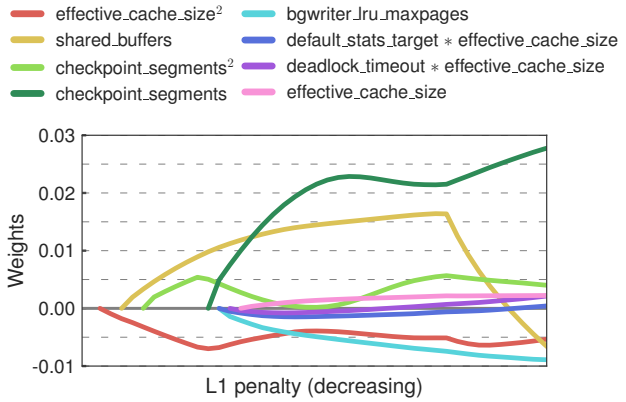


Figure 13: Lasso Path (Postgres)

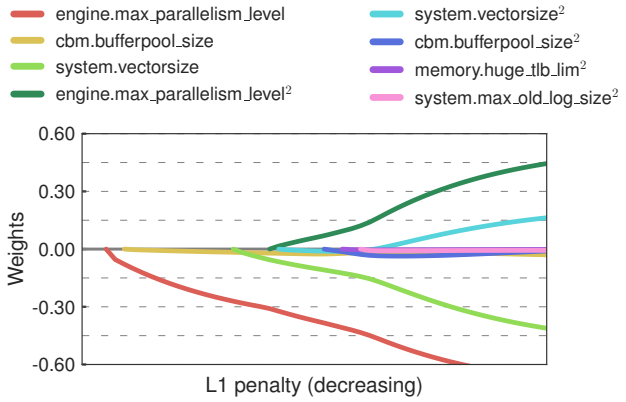


Figure 14: Lasso Path (Vector)

APPENDIX

A. IDENTIFYING IMPORTANT KNOBS

This section extends the discussion of the Lasso path algorithm presented in Sect. 5.1. The results in Figs. 12 to 14 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 features 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 Sect. 5.2, 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, Fig. 13 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. Figs. 12 to 14 show that, as expected, the relationship between the buffer pool size knob 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).

B. EFFICACY COMPARISON

This section is an extension of Sect. 7.6, where we provide the DBMS configurations generated by OtterTune, the DBA, the tuning script, and Amazon AWS that were used in the evaluation. Tables 1 and 2 show the configurations for the TPC-C workload running on MySQL and Postgres, respectively. 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.

C. FUTURE WORK

There are important problems that remain unsolved in this line of work. Foremost is that we want to enable OtterTune to automatically detect the hardware capabilities of the target DBMS. This is tricky because it must be able to do so with only remote access to the DBMS's host machine. This restriction is necessary because of the prevalence of database-as-a-service deployments where such access is not available. One approach might be for OtterTune to execute a microbenchmark before it begins the workload mapping step that would stress different resources in the DBMS separately.

Another problem that we plan to explore is how to adapt the auto-tuning techniques described in this paper to optimize the physical design of a database. In particular, we are interested in leveraging data from previous tunings to speed up the process of tuning a new application. Similar to tuning DBMS configurations, tuning the physical design is becoming increasingly complex to the point where techniques that are able to reduce the complexity of the problem are becoming more and more necessary.

D. ACKNOWLEDGEMENTS

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

(a) 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

(b) 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

(c) 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

(d) 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 2: DBMS Configurations (Postgres) – The best configurations for the TPC-C workload running on Postgres generated by (a) OtterTune, (b) the DBA, (c) the tuning script, and (d) Amazon RDS.

(a) 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

(b) 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	–

(c) 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

(d) 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 1: DBMS Configurations (MySQL) – The best configurations for the TPC-C workload running on MySQL generated by (a) OtterTune, (b) the DBA, (c) the tuning script, and (d) Amazon RDS.