FINAL REPORT
AUTOMATIC TUNING OF DBMS USING MACHINE LEARNING
CMU 10-701: MACHINE LEARNING (FALL 2014)

Joy Arulraj Ram Raghunathan
{jarulraj, rraghuna}

Abstract

Modern database systems are highly configurable and must support a wide variety of workloads. However, tuning these systems is challenging. Configuration assistants like Microsoft's AutoAdmin [1] allow administrators to use statistics and suggestions from the database system to guide the physical design of the database. However, these methods still require experienced administrators and prior knowledge about the workload. We seek to apply machine learning methods to automate database system tuning with minimal user input and knowledge by matching workloads to representative benchmarks. In this midway report, we discuss the our progress on the workload mapping problem.

1. Introduction

1.1. Questions

We address these two questions in this project. First, we plan to *map* a given workload comprised of a set of SQL transactions to a standard database benchmark workload. This problem is independent of the underlying DBMS or hardware configuration. This will allow us to use prior knowledge about the standard benchmark gained from previous DBMS deployments.

We collect features that characterize the SQL workload as well as DBMS statistics. It will be interesting to see if feature extraction yields insights about the most influential features. We then use unsupervised techniques like *clustering* for mapping the workloads. Performance analysis of the resulting classifier is done via cross-validation.

Second, we plan to *estimate* the DBMS performance given a DBMS configuration, hardware setup and SQL workload. This will be done with supervised techniques like *Gaussian process regression*. As we expect a lot of features to be present in the input, we may need to make use of a feature extraction algorithm like principal component analysis (PCA) first. Performance analysis of the estimator will also be done using cross-validation. We describe our progress on solving the first problem in this report.

1.2. Minimum goals

The minimum goals are : (a) to create a workload mapper that maps an arbitrary SQL workload to a well-known standard benchmark, and (b) to estimate the performance of a DBMS given a workload and configuration pair.

We have already achieved the first goal on our dataset. We plan to generalize it to more sophisticated datasets as our stretch goal. We have also setup the infrastructure required for achieving the second goal.

1.3. Stretch goals

As a real-world workload can exhibit some aspects of different standard benchmarks or can exhibit different characteristics across time (e.g. read-mostly during day and write-mostly during night), mapping the entire workload onto a single benchmark may not be an entirely accurate characterization. To help characterize the workload more accurately, we will consider mapping parts of a workload onto different benchmarks using techniques like *multi-label prediction*.

In this report, we first describe how we generate the dataset in Section 2. We then focus on the feature extraction problem in Section 3. Finally, we sketch our initial evaluation results in Section 4.

2. Data Set

The dataset required for this project would ideally comprise of a collection of real world SQL workloads, DBMS configurations and performance metrics. Collecting and curating such a dataset is itself an interesting problem. However, in this project, we first want to experiment with a smaller dataset to better understand the features relevant for our learning problem. Therefore, we chose

to generate the dataset using OLTP-Bench [5], an extensible DBMS benchmarking framework. We use the SQL workloads of standard benchmarks already available in OLTPBench.

We generate more synthetic variants of these workloads for training and testing purposes. As we mentioned earlier, while this synthetic data will not be representative of real-world workloads, we feel it is a good starting point for evaluating viability of the approach outlined above. We extract several features from the workload such as types of database queries, distribution of query types, table access patterns, etc. using a workload analyzer. We hook this analyzer into Postgres [10] to collect features from the DBMS.

The key reasons for why we use this framework to generate the dataset are the following:

- It supports several relational DBMSs through the JDBC interface including Postgres, MySQL, and Oracle.
- It allows us to control the workload mixture in a benchmark. For instance, we can adjust the percent of read and update transactions in the YCSB [3] benchmark to generate different variants of the workload.
- It supports user-defined configuration of the rate at which the transaction requests are submitted to the DBMS. This allows us to emulate different world workloads with varying degrees of concurrency.
- The framework exposes statistics that are complementary to the internal statistics of the DBMS [4]. We extract features from these statistics.

We implemented a dataset generator that runs different benchmarks supported by OLTP-Bench on a Postgres DBMS. After every workload execution, we collect statistics from the DBMS as well as from the testbed. We alter the workload mixture in all the benchmarks to generate different variants and emulate real world workloads. The key characteristics of the benchmarks that we use for generating the dataset are presented in Table 1.

3. Feature Extraction

We collect 3 types of features from both the DBMS as well as the benchmarking framework.

3.1. Features from OLTP-Bench

After executing the benchmark, we obtain statistics about the latency and throughput delivered by the DBMS. This includes both temporal performance metrics as well as aggregate metrics. We then record the size of the workload also known as the *scalefactor*. The *isolation level* of the DBMS correlates strongly with performance metrics Stricter isolation levels like "serializable level" correlate with lower performance because the DBMS needs to maintain the constraints regarding the visibility of effects of concurrent transactions.

We also record the type of DBMS used. Although currently we focus only on Postgres, we anticipate this tool to be useful for other DBMSs as well. We also record the expected label i.e. the benchmark name. This is used for evaluating the accuracy of our classification algorithms.

3.2. Static parameters from Postgres

Static parameters are features that do not vary over every execution. This primarily includes the configuration parameters of the DBMS and the hardware setup. For example, these are some of the static parameters that we use as features:

- Size of shared memory buffers: This impacts the performance of memory-intensive queries significantly.
- Background writer delay: The background writer issues writes of dirty shared buffers to disk. This increases the net overall I/O load but allows server processes to avoid waiting for writes to finish.
- Vacumm cost delay: The vacumm process performs garbage collection. Very short delays can impact the DBMS performance.
- WAL level: The type of write-ahead logging performed - minimal, archive, or hot standby - affects the logging overhead.
- *fsync*: Durability requirements of data.
- Sequential page cost: Used in the cost model of the DBMS' planner.
- Hardware features: CPU cache sizes, DRAM size, disk size, cache latency, DRAM latency and disk latency.

Overall, these metrics significantly impact the performance of the DBMS. A non-expert user might not be able to configure these parameters to obtain good performance. Our tuning tool can help such users by automatically identifying a good DBMS configuration for a given workload.

3.3. Dynamic parameters from Postgres

We also collect dynamic parameters from the DBMS during feature extraction. To do this, we implemented a Postgres driver that queries the DBMS's internal catalog

Benchmarks	Tables	Columns	Pr. Keys	Indexes	Fr. Keys	Txn. Types	# of Joins	Application domain	Attributes
AuctionMark	16	125	16	14	41	10	10	Online Auctions	Non-deterministic heavy transactions
Epinions	5	21	2	10	0	9	3	Social Networking	Joins over many-to-many relationships
SEATS	10	189	9	5	12	6	6	Online Airline Ticketing	Secondary indices queries foreign-key joins
TATP	4	51	4	5	3	7	1	Caller Location App	Short, read-mostly non-conflicting transactions
TPC-C	9	92	8	3	24	5	2	Order Processing	Write-heavy transactions
Twitter	5	18	5	4	0	5	0	Social Networking	Client-side joins on graph data
Wikipedia	12	122	12	40	0	5	2	Online Encyclopedia	Complex transactions large data, skew
YCSB	1	11	1	0	0	6	0	NoSQL store	Key-value queries

Table 1: Key characteristics of the benchmarks used in our evaluation. "Pr. key" denotes primary key and "Fr. key" denotes foreign key.

tables like pg_stat_database, pg_statio_user_indexes, pg_stat_activity and pg_stat_user_table to obtain useful workload parameters. For instance, these are some of the dynamic parameters that we use as features :

- Number of transactions in this database that have been committed or rolled back.
- Number of disk blocks read in this database.
- Number of times disk blocks were found already in the DBMS's buffer cache.
- Number of rows returned, fetched, inserted, updated or deleted by queries in this database.
- Number of index and cache blocks hit.
- Status of different storage backends of the DBMS.
- Size of the tables and indexes in the DBMS.
- Number of sequential scans and index scans performed by the workload.

We normalize the relevant metrics by the number of transactions executed. Before the start of an execution, we reset all the statistics using our DBMS driver. Overall, this gives us a nice set of features about the workload.

After collecting all these features, we transform them to a metric space and normalize them to generate a feature matrix and a label matrix. This is used by the classification algorithms that we describe in the next section.

4. Evaluation

We address two problems in our experiments :

- Mapping a SQL workload to a benchmark class.
- Estimating the performance of the DBMS using the features and the class model.

We note that we leave the problem of using the performance estimator to find an optimal DBMS configuration for a given workload for future work. The estimator allows us to obtain inexpensive estimates without actually running the workload on the DBMS under a specific configuration.

We first describe the experimental setup and information about the dataset in Section 4.1. Then, we present the results for the workload mapping/classification problem in Section 5 and for the performance estimation problem in Section 6. Finally, we discuss the conclusions and future work in Section 7.

4.1. Experimental Setup

All our experiments are performed on the machine described in Table 2. We deploy Linux kernel 3.2.0 on the machine and disable logical processor support ("Hyper-Threading"). We use the Scikit [7] framework in Python 3 for evaluating different machine learning algorithms. Our auto-tuning framework currently only supports Postgres 9.5. We plan to extend it to support other database management systems like MySQL in future work.

The dataset consists of 10 benchmark classes. These include all the benchmarks described in Table 1. We consider 4 variants of the YCSB key-value store benchmark. These variants map to different read-write mixtures and are listed below :

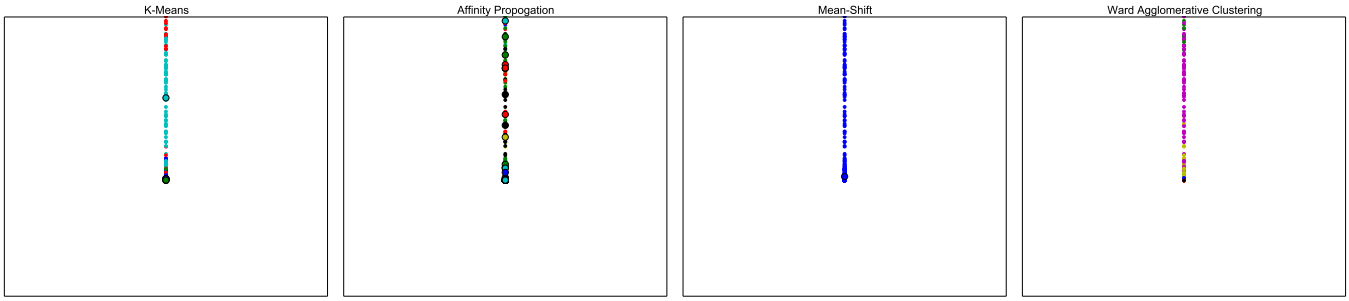


Figure 1: Clusters found by the clustering algorithms.

Algorithm	Homogeneity	Completeness	V-Measure	# of clusters	Silhouette Coefficient
K-Means	0.607	0.592	0.600	10	0.106
Affinity-Propagation	0.654	0.317	0.427	88	0.082
Mean-Shift	0.012	0.368	0.024	2	0.517
Ward Agglomerative Clustering	0.589	0.635	0.611	10	0.097

Figure 2: Performance metrics of clustering algorithms.

Attribute	Value
CPU	8 cores (Intel i7-3770 @ 3.7 GHz)
DRAM	16 GiB
L2 cache	256 KiB
L3 cache	8 MiB

Table 2: Experimental setup.

- Read-only : 100% reads 0% writes
- Read-heavy : 80% reads 20% writes
- Balanced : 50% reads 50% writes
- Write-heavy : 20% reads 80% writes

We added these variants to stress-test the classifiers and estimators as these variants can overlap with other benchmark classes along some dimensions in high-dimensional space.

For each DBMS run, we first generate a configuration by picking values for 12 key configuration parameters listed in Table 3. We then select a random benchmark and alter the transaction mix of the benchmark i.e. the proportion of different types of transactions in the specific benchmark. After running the benchmark for atleast more than 1 minute, we halt the system. We then collect the dynamic features from the DBMS and the benchmarking framework. We also collect the static features from the specific DBMS configuration. These features and metrics constitute one sample in the dataset. Each sample consists of 272 features and our dataset consists of 1000 such samples.

5. Classification

We addressed the problem of mapping a workload to a standard benchmark class using two techniques.

5.1. Clustering

Clustering was our first approach as an unsupervised algorithm seemed to best fit the data at hand. We evaluated the following clustering algorithms in Scikit.

- **K-Means** : This algorithm clusters the samples into groups of similar variance by minimizing the within-cluster sum-of-squares. The number of clusters needs to be specified. The algorithms scales well to a large number of samples.
  Given a set of n samples X, the algorithm splits them into K disjoint clusters, wherein each cluster is defined by the mean μ_k of the samples in the cluster i.e. the centroids of the cluster. The algorithm minimizes the within-cluster sum-of-squares criterion:

$$\sum_{i=0}^{n} \min_{\mu_k \in C} (\|x_k - \mu_i\|^2)$$

- **Affinity Propagation**: This algorithm creates clusters by passing messages between pairs of samples till it reaches convergence. The clusters are described using a small number of exemplars that are most representative of the dataset. The messages indicate the suitability of one sample to be the exemplar of the other and this gets updated over time for the entire dataset.

Feature name	Feature meaning	Default value	Mutated value set
shared_buffers	Number of shared memory buffers used by the database server	128 MB	[4 MB, 32 MB, 128 MB, 512 MB]
bgwrite_delay	Delay between rounds for the background writer	100 ms	[100 ms, 1000 ms, 10000 ms]
wal_level	Type of logging	minimal	[minimal, archive, hot_standby, logical]
fsync	Forced synchronization to disk	on	[on, off]
synchronous_commit	Whether transaction commit will wait for WAL records to be flushed	on	[on, off, local, remote_write]
wal_buffers	Number of disk-page buffers in shared memory for WAL	-1	[-1, 4 MB, 32 MB, 128 MB]
wal_writer_delay	Specifies the delay between rounds for the WAL writer	200 ms	[100 ms, 200 ms, 1000 ms, 10000 ms]
commit_delay	Time delay between writing a commit record to the WAL buffer	0 μs	[0 μs, 1000 μs, 1000 μs, 10000 μs]
track_activities	Collect query/index statistics	on	[on, off]
log_planner_stats	Log planner statistics	off	[on, off]
debug_print_plan	Print plans in log	off	[on, off]
autovacuum	Enable autovacuum subprocess	on	[on, off]

Table 3: Configuration attributes that we mutate in each DBMS run.

For a pair of samples i and k, the evidence that sample k should be the exemplar for sample i is defined by:

$$r(i,k) \leftarrow s(i,k) - max[a(i,j) + s(i,j) \forall j \neq k]$$

Here, $s(i,k)$ is a similarity metric and availability $a(i,k)$ is the accumulated evidence that sample i should choose sample k as its exemplar. Thus, the exemplars chosen are similar enough to many samples and are chosen by many samples to be representative of themselves.

- **Mean-Shift** : This algorithm tries to identify blobs in a smooth density of samples. It works by first identifying candidates for centroids and then filtering them to eliminate near-duplicates.
  For a candidate centroid x_i in iteration t, the algorithm updates the candidate effectively to be the mean of the samples within its neighborhood:

$$x_i^{t+1} = x_i^t + \frac{\sum_{x_j \in N(x_i)} K(x_j - x_i) x_j}{\sum_{x_j \in N(x_i)} K(x_j - x_i)}$$

  Here, $N(x_i)$ depicts the neighborhood of samples within a given distance around x_i and the additive term is basically the mean shift vector computed for each centroid that points towards a region of the maximum increase in the density of points.

- **Agglomerative Clustering** : This algorithm is a type of hierarchical clustering algorithms that build nested clusters by merging or splitting them successively. The cluster hierarchy is represented as a tree, wherein the root is the unique cluster that gathers all the samples and the leaves are the clusters with only one sample. This particular algorithm uses a bottom up approach. Each samples starts in its own cluster, and over time the clusters are successively merged together based on a linkage criteria. We use the *ward* criteria that minimizes the sum of squared differences within all clusters that is effectively a variance-minimizing approach.

The clusters found by each algorithm in high-dimensional space are shown in Figure 1. We computed standard clustering metrics for each algorithm including the following :

- **Homogeneity**: Given a ground truth, this metric computes if all the clusters contain only data points which are members of a single class.
- **Completeness**: Given a ground truth, this metric computes if all the data points that are members of a given class are elements of the same cluster.
- **V-measure**: This metric is the harmonic mean between homogeneity and completeness [9].

$$v = 2 * \frac{(homogeneity * completeness)}{(homogeneity + completeness)}$$

- **Silhouette coefficient**: This metric is computed using the mean intra-cluster distance a and the mean nearest-cluster distance b for each sample. It is defined for each sample as $(b-a)/max(a,b)$.

These results are presented in Figure 2. We observe that the clustering algorithms work reasonably well with our dataset especially the K-Means and Ward Agglomerative Clustering algorithms. Both these algorithms require number of clusters as a parameter. However, this is not a restriction for our problem as we know the number of benchmarks - and hence the number of clusters - that we have. Algorithms like Affinity-Propagation and Mean-Shift give very high and very low estimates for the number of clusters in the dataset. As we required higher classification accuracy and better intuition about the classifier, we also experimented with SVM classifiers and

decision trees. We finally decided to use decision trees as they met both our accuracy and intuition requirements.

5.2. Decision Trees

Decision trees are non-parametric supervised learning algorithms used for both classification and regression. It creates a model for predicting the class of a sample by learning simple decision rules inferred from the sample features[6]. These trees can be interpreted easily and visualised. We however need to be careful not to create overly complex trees that do not generalise the data well i.e. we should avoid overfitting.

Given a set of training samples $x_i \in R^n$, $i = 1, ..., l$ and the corresponding label vector $y \in R^l$, the decision tree partitions the space recursively so that samples with the same labels are grouped together. We denote the data at a node m in the decision tree as Q. For every candidate split $\theta = (j, t_m)$ that consists of a feature j and a feature-specific threshold t_m, the tree partitions the data into two subsets $Q_{left}(\theta)$ and $Q_{right}(\theta)$, wherein :

$$Q_{left}(\theta) = (x, y)|x_j <= t_m$$

$$Q_{right}(\theta) = Q \setminus Q_{left}(\theta)$$

At each node m, we compute an impurity measure that we denote by H.

$$G(Q, \theta) = \frac{n_{left}}{N_m} H(Q_{left}(\theta)) + \frac{n_{right}}{N_m} H(Q_{right}(\theta))$$

We search for parameters that minimize H:

$$\theta^* = \text{argmin}_\theta \, G(Q, \theta)$$

We then do the same process recursively for subsets $Q_{left}(\theta^*)$ and $Q_{right}(\theta^*)$ until we reach the maximum allowable depth or $N_m < \min_{samples}$ or $N_m = 1$. For the classification problem where the target is in $[0, K-1]$ for node m, representing a region R_m with N_m observations, the proportion of class k observations in node m is given by :

$$p_{mk} = 1/N_m \sum_{x_i \in R_m} I(y_i = k)$$

We use the *Gini measure* as the impurity measure:

$$H(X_m) = \sum_k p_{mk}(1 - p_{mk})$$

The algorithm is an optimized version of the CART (Classification and Regression Trees) algorithm[2]. It basically constructs binary trees using the feature and threshold that yield the largest information gain at each node. The features need not be categorical as the algorithm dynamically defines a discrete attribute based on numerical variables that partitions the continuous attribute value into a discrete set of intervals. It does not compute rule sets unlike the C4.5 algorithm[8].

A sample decision tree with maximum depth throttled to 4 is shown in Figure 3. We obtain 76% accuracy with this short tree and derive these interesting observations :

- Wikipedia benchmark performs a lot of index scans.
- SEATS benchmark involves fetching several rows per transaction.
- Epinions benchmark, on the other hand, involves returning several rows per transaction.
- Auctionmark benchmark has less locality of reference i.e. the number of disk block cache hits is low.
- Twitter benchmark also fetches several rows per transaction. However, it does not perform any sequential scans.

The accuracy increases quickly to 91% with a maximum depth of 8. The impact of maximum tree depth on the accuracy of the decision tree classifier is presented in Figure 4. Using three-way cross validation, we verified that the tree is able to classify the dataset accurately without overfitting. The impact of maximum number of leaf nodes on the accuracy of the decision tree classifier is presented in Figure 5. We observe that these parameters affect both precision and recall equally. The accuracy increases more steeply with increasing tree depth as expected.

The per-class accuracy metrics of the decision tree is shown in Table 4. The precision is the ratio $tp/(tp+fp)$ where tp is the number of true positives and fp the number of false positives. It is intuitively the ability of the classifier not to label as positive a sample that is negative. The recall is the ratio $tp/(tp+fn)$ where tp is the number of true positives and fn the number of false negatives. It is intuitively the ability of the classifier to find all the positive samples. The F1 score is a weighted average of the precision and recall and is defined as:

$$F1 = 2 * \frac{(precision * recall)}{(precision + recall)}$$

The support is the number of occurrences of each class in the true label vector y_true.

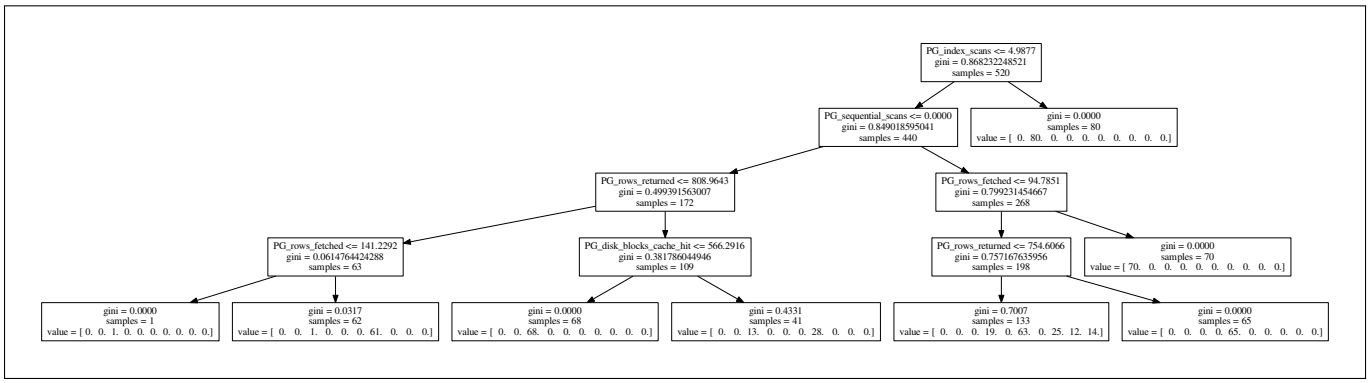


Figure 3: Decision tree with max depth set to 4.

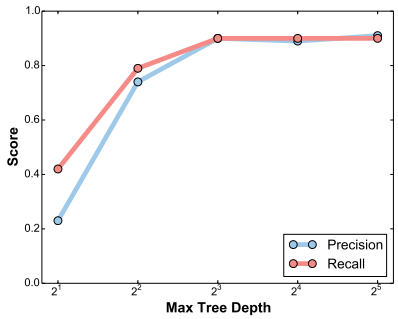


Figure 4: Impact of max depth on the accuracy of the decision tree.

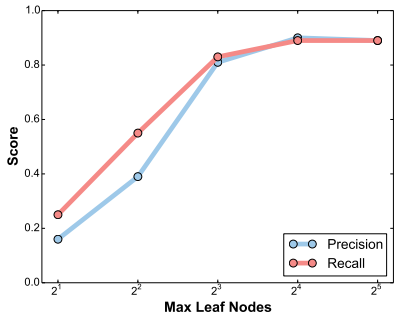


Figure 5: Impact of max leaf nodes on the accuracy of the decision tree.

Class	Precision	Recall	F1-score	Support
0.0	1.00	0.00	1.00	84
1.0	0.99	1.00	0.99	74
2.0	0.98	0.99	0.98	81
3.0	0.54	0.39	0.45	18
4.0	1.00	1.00	1.00	69
5.0	1.00	0.99	0.99	70
6.0	1.00	0.95	0.97	60
7.0	0.41	0.95	0.57	19
8.0	0.00	0.00	0.00	17
9.0	0.56	0.52	0.54	27
Avg / Total	0.90	0.91	0.90	519

Table 4: Per-class accuracy metrics of the decision tree with max depth = 8.

6. Estimation

We estimate expected performance of each benchmark on a given database configuration for the second phase of our solution. In doing so, we hope to approximately estimate the performance of the target workload through the estimation for the benchmark it maps to. As such, we train two estimators for each benchmark: one for database throughput and one for database latency.

We chose to use Gaussian Process Regression, a supervised learning method, for this task as it works well for estimating regressions with no prior knowledge about distribution. It also gives a probabilistic estimation, allowing for further insight about bounds and probability of exceeding them. The Gaussian Process Regression models the data with the function

$$G(X) = f(X)^{\mathrm{T}}\beta + Z(X)$$

where $f(X)^{\mathrm{T}}\beta$ is a linear regression model and $Z(X)$ is a Gaussian process with zero mean. X is the input, represented as a vector of feature values. In this model, we try to find the "best linear unbiased prediction" (BLUP) of the process given the training data. That is, we try to find the best function for $\hat{G}(X) = G(X|\text{training data})$. Under the model, it can be shown that $\hat{G}(X) = a(X)^{\mathrm{T}}y$ where $a(X)$ is the product of the weights with the feature values. From this model, we can find the we can find the BLUP by solving the optimization problem

$$a(X)^* = \arg\min_{a(X)} \mathbb{E}[(G(X) - a(X)^T y)^2]$$

subject to the constraint $\mathbb{E}[G(X) - a(X)^T y] = 0$. Gaussian Processes are amenable to kernels, although they are

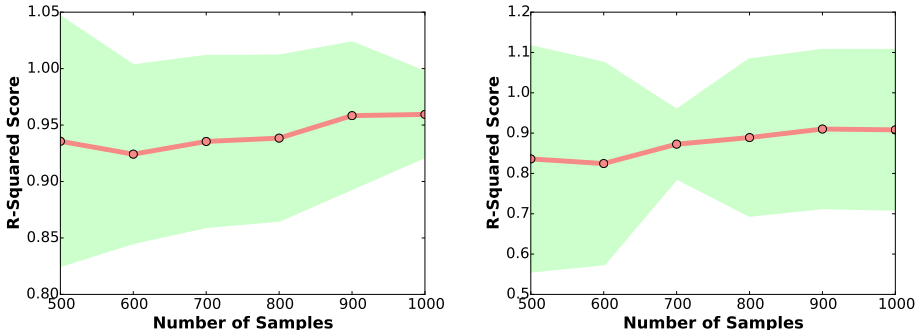


Figure 6: Per-benchmark gaussian processes to estimate (a) latency and (b) throughput

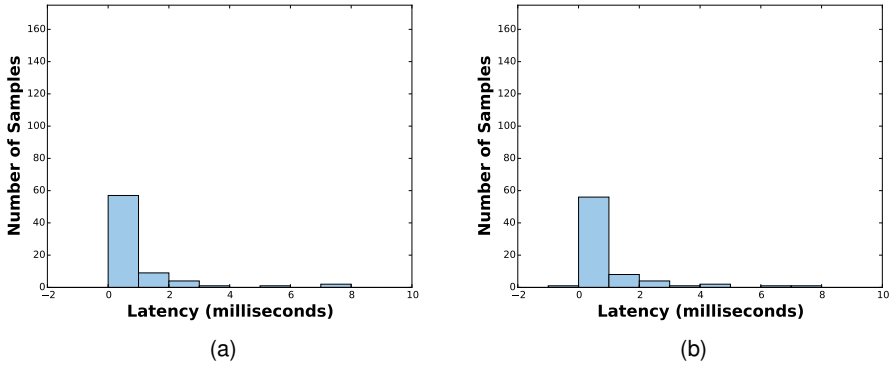
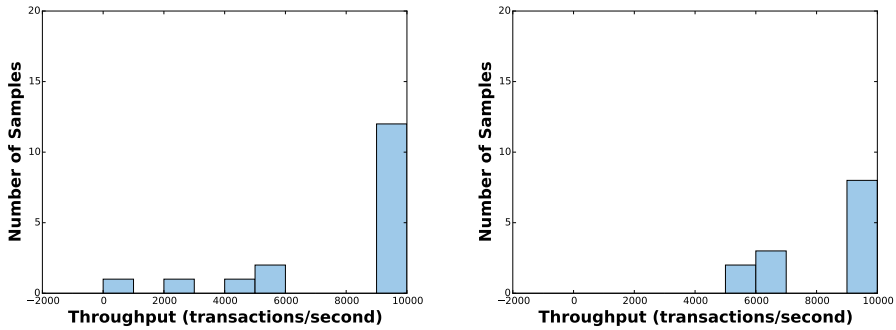


Figure 7: (a) Actual and (b) predicted latency distribution for the Wikipedia benchmark

Feature Name	Feature Meaning
PG_Cache_Hits	Number of buffer hits in this table
PG_Index_Hits	Number of buffer hits in all indexes on this table
PG_Index_scans	Number of index scans initiated on this table
PG_rows_inserted	Number of rows inserted by queries in this database
PG_rows_returned	Number of rows returned by queries in this database
PG_rows_updated	Number of rows updated by queries in this database
PG_sequential_scans	Number of sequential scans initiated on this table
PG_transactions_committed	Number of transactions in this database that have been committed
PG_transactions_rolled_back	Number of transactions in this database that have been rolled back
autovacuum	Number of times the autovacuum daemon vacuumed this table
commit_delay	Time delay before a transaction attempts to flush the WAL buffer
fsync	Enable making sure that updates are physically written to disk
shared_buffers	Amount of memory the database server uses for shared memory buffers
track_activities	Enable the collection of information on the executing commands
wal_buffers	Amount of shared memory used for WAL data
wal_level	Determine how much information is written to the WAL
wal_writer_delay	Specify the delay between activity rounds for the WAL writer

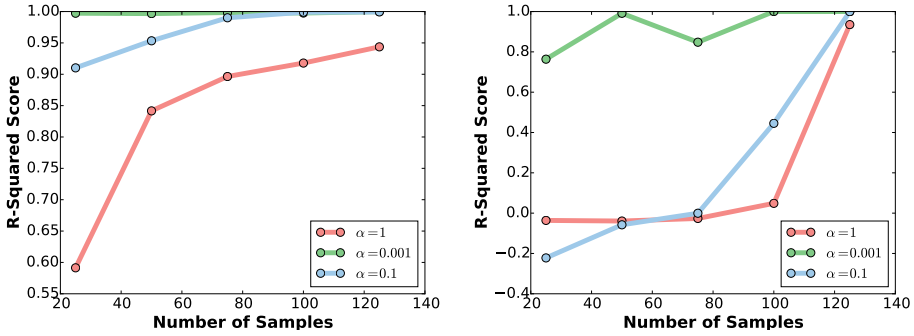
Table 5: Some of the influential features for throughput estimation.

used as correlations in this model. We use the absolute exponential autocorrelation model in our experiments.



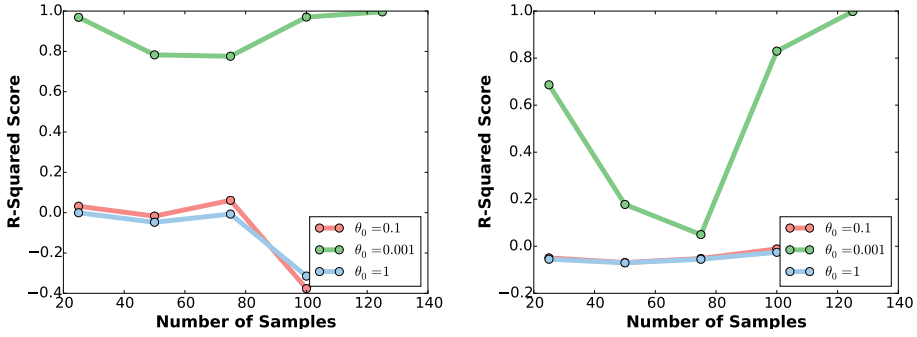
(a) (b)

Figure 8: (a) Actual and (b) predicted throughput distribution for the balanced YCSB benchmark



(a) (b)

Figure 9: Performance of Lasso for various α values for (a) latency and (b) throughput on the Wikipedia benchmark



(a) (b)

Figure 10: Performance of Gaussian Process Regression with absolute exponential autocorrelation for various θ_0 values for (a) latency and (b) throughput on the Wikipedia benchmark

Feature Name	Feature Meaning
PG_index_scans	Number of index scans initiated on this table
PG_sequential_scans	Number of sequential scans initiated on this table
fsync	Enable making sure that updates are physically written to disk
synchronous_commit	Whether transaction commit will wait for WAL records to be flushed

Table 6: Influential features for latency estimation.

This covariance model calculates autocorrelation of a vector X as

$$\exp\left(\sum_{i=1}^{n} -\theta_0|\Delta X_i|\right)$$

where θ_0 is a the autocorrelation parameter and ΔX is the component-wise distances at which the correlation should be calculated. We experimented with various values of θ_0 and settled on $\theta_0 = 0.001$ as it exhibited the best performance, as seen in Figure 10.

One way to measure the performance of a regression model is the R^2 score defined as

$$R^2 \equiv 1 - \frac{SS_{res}}{SS_{tot}}$$

In this definition, $SS_{res} = \sum_i (y_i - f_i)^2$ where y_i is the true value and f_i is the predicted value. This term is also known as the residual sum of squares. The other term in the definition is $SS_{tot} = \sum_i (y_i - \bar{y})^2$ where y_i is the true value and $\bar{y}$ is the mean of all the true values. This is proportional to the variance of the true values. From this definition of the R^2 score, we see that the maximum value is 1 and that it indicates that the model perfectly predicted the observed data. A score less than 1 indicates that the model was not perfect and a score of zero indicates that the model is no better at predicting the values than the mean of the true values. Figure 6 shows the median R^2 score across the estimators for all benchmarks as a red line. The shaded green region indicates a single standard deviation spread. We infer from the figures that we can achieve a very good estimation with as few as 600 samples. In addition, the estimators become collectively better as more data is provided, as evidenced by the shrinking green region as number of samples increases.

One example of a good performance estimator is the latency estimator for the Wikipedia benchmark. This estimator has a high R^2 score of 0.9734. Looking at the actual and predicted latency values in Figure 7, we find that the predicted distribution mimics the actual one very closely, with only a slight dip into negative latency predictions. We believe this estimator can be even stronger if we encode constraints for possible predictions, such as positive values only.

One example of a poorer performance estimator is the throughput estimator for the balanced YCSB benchmark. This estimator has a lower R^2 score of 0.7090. Looking at the actual and predicted throughput values in Figure 8, we see that the estimator fails to handle the holes in throughput seen in the actual values. We believe these isolated throughput values are due to environmental factors that are not reflected in the feature set such as load on the machine and other processes' resource usages. A more "global" set of features across the whole system rather than just pertaining to the database may help train better estimators.

To gain more insight about what parameters affected performance the most, we performed Lasso Regression to determine the most influential features. Lasso regression encourages sparse coefficients in its solution, so it is a good method for finding the most important features for a particular problem. It works by optimizing the objective function

$$\min_w \frac{1}{2n_{\text{samples}}} \|Xw - y\|_2^2 + \alpha \|w\|_1$$

which is a simple least squares linear regression with the lasso constraint added under weight α. A higher value for α encourages sparser solutions while a lower value relaxes this constraint. We experimented with different values of alpha and settled on using $\alpha = 0.001$ as it exhibited the best performance, as seen in Figure 9.

These features are summarized in Table 5 and Table 6. Confirming general intuition, throughput seems to be greatly affected by all operation types along with options that add delays (e.g. bgwriter_delay, commit_delay) or constrain how relaxed the database can be about transaction execution (e.g. fsync, synchronous_commit). However, it is interesting to note that latency seems primarily affected only by operations that take a long time to complete and not the type of operations or hard delays

introduced into the system.

In a practical implementation of our estimator, we will not be able to collect performance statistics before estimating performance for different configurations. As such, we eliminated the features pertaining to throughput and latency from consideration. However, these features are available in the training data. We believe that a better learner will use graphical models to infer the missing throughput and performance information for the test data before estimation. We will explore this technique in the future.

7. Conclusion

DBMS performance tuning is a niche skill that requires much experience and experimentation to get correct. However, machine learning techniques can help automate this process by using prior knowledge to estimate performance without having to go through a time-consuming experiment. To achieve this goal, we have taken a two-step approach where we first use map a workload to a well-studied benchmark. Armed with this information, we then use a regression estimator to estimate the performance of the benchmark under a given environment.

Using decision trees to map the workloads, we find that we can effectively classify workloads using a few defining characteristics. For performance estimation, we trained estimators using Gaussian Process Regression that show very accurate estimation of database throughput and latency. In addition, these highly accurate estimators can be trained with as few as 600 samples!

While promising, this work only uses synthetic data generated from OLTPBench. We hope to continue this work using a larger dataset that includes mixtures of benchmarks along with real-world workloads. In addition, we hope to extend our research to include performance estimation across different hardware profiles and DBMS's.

References

[1] Sanjay Agrawal, Surajit Chaudhuri, Abhinandan Das, and Vivek Narasayya. Automating layout of relational databases. In *In Proceedings of 19th International Conference on Data Engineering*, pages 607–618, 2003.

[2] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[3] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. *SoCC*, 2010.

[4] PG developers. Postgres statistics collector. http://www.postgresql.org/docs/9.1/static/monitoring-stats.html, 2014.

[5] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. http://oltpbenchmark.com, 2014.

[6] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer New York Inc., New York, NY, USA, 2001.

[7] 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.

[8] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[9] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*, pages 410–420, 2007.

[10] Michael Stonebraker and Greg Kemnitz. The postgres next-generation database management system, 1991.