Using Regression to Estimate Query Resource Consumption

Baron Schwartz, VividCortex

Database administrators often are interested in the resources a statement or query consumes. An example is CPU: we would like to know how much CPU time a query uses. Query resource consumption is not possible to measure directly in many cases, and not always desirable even if possible. This paper discusses how a variation of linear regression can reveal relationships between metrics such as query execution time and arbitrary metrics inside or outside of the RDBMS.

1 Problem Statement and Approach

We will first explain the specific problem we solve with the techniques presented in this paper, and then explain the method in its more general form. We measure statistics such as response time and throughput for each class of query that is executed in a relational database management system (RDBMS; hereafter called a database server). Queries are classified by removing literals and normalizing whitespace. We use the term "query" to denote any SQL program or statement.

The measurements for each query are accumulated per class, and the sum is sampled once per second. The result is a collection of time-series metrics for each class of query. At the same time, we observe metrics such as user and system CPU time consumed from the server process itself. Our technique lets us divide aggregate metrics such as user CPU time into portions, which we can blame on the classes of queries.

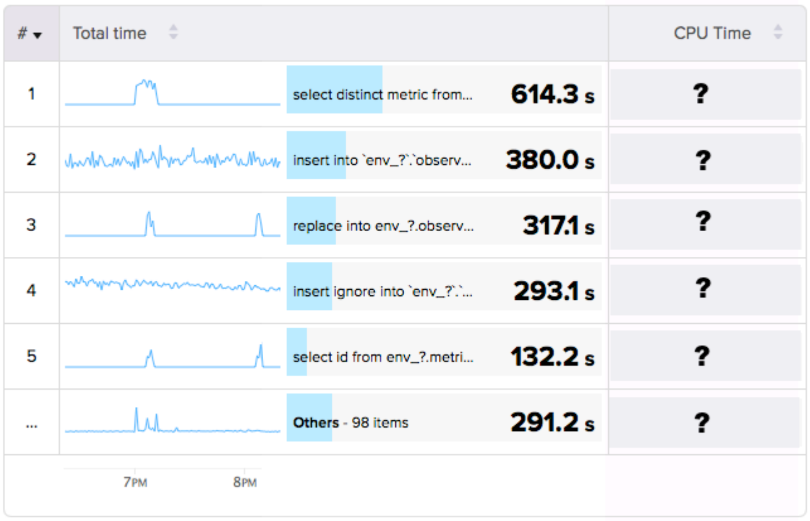
The effect is to produce approximate per-query-class metrics of user CPU time or other metrics. In this paper we will focus exclusively on query wall-clock time and user CPU time. Figure 1 on the following page shows a user interface concept from our application, which might help explain the goal. The technique amounts to disaggregation of metrics. Among its advantages:

- We reduce the amount of data we measure from the system under observation.
- We can perform high-dimensional analysis on low-dimensional data.
- We can perform ad-hoc analysis without needing to identify metrics of interest a priori.
- We can compute granular metrics that are not otherwise available. RDBMSs and similar systems offer limited per-query metrics. The MySQL database server, for example, does not expose per-query CPU time metrics, nor would it be practical to build into the database server every possible metric of interest.
- We can compute metrics that are not possible to instrument inside the database server. For example, we know from studying database server internals that much of a query's CPU work is deferred and performed lazily in the background, often combined with work from many other queries. We believe that in some cases regression is more accurate than anything that could be instrumented inside the database server.

2 Sample Data and Code

We have made available sample data and code for readers of this paper to examine and reproduce our results. Both are available at VividCortex's GitHub repository named wlr.

The sample data consists of two CSV files. A header row labels the columns. Each row contains a sample of data from a MySQL database server. Each column contains metrics about a single class of queries. The dimension is in microseconds of elapsed response time for all queries of that class observed within the observation interval for the sample. The last column in the file is an exception. Instead of a query class's metrics, it contains the database server's accumulated user CPU time, in microseconds.



#	Total time			CPU Time
1		select distinct metric from...	614.3 s	?
2		insert into `env_?`.`observ...	380.0 s	?
3		replace into env_?.observ...	317.1 s	?
4		insert ignore into `env_?`.`...	293.1 s	?
5		select id from env_?.metri...	132.2 s	?
...		**Others** - 98 items	291.2 s	?

Figure 1. The goal is to compute the CPU column in this user interface

The two sample files are contain one day's worth of data for the server we chose to measure. Samples were taken at 60-second intervals, so each row represents one minute of accumulated statistics. The first sample data file contains the first 12 hours of the day, and the second one the latter half of the day.

We chose the provided sample data to encourage readers to experiment and compare results from other methods. The sample is relatively clean and free of strange effects such as dramatically shifting server behavior. It is, however, a real dataset from a production database server, running a mixed workload of various types of read and write operations combined with periodic batch jobs.

Sample code is provided in the code directory, in Google's Go programming language. To compile and execute the code, readers should install Go, which is available for all common operating systems. You can execute the code as follows:

```
go run regress.go sample-data/001.csv
```

The program will read the CSV file and "train" itself by performing the regression technique described in this paper, printing lines of output in the following form:

```
TRAIN name xValue yValue slope contrib
TRAIN q.2e09446e39e26d42 1859 1.4673e+08 3.8417 7141.7
TRAIN q.42107779fcd075ef 12065 1.4673e+08 3.8417 46350
```

The output shows the query class, the value of the query class during the given time interval, the amount of server CPU time during this interval, the slope computed at this iteration, and the query class's computed contribution to the server CPU time. After the regression is complete, a prediction phase begins, during which the program will reread the CSV file and attempt to predict the CPU time column from the columns of per-query-class elapsed time. Output during this phase resembles the following:

```
PREDICT actual predicted
PREDICT 1.4673e+08 1.342e+08
PREDICT 1.5952e+08 1.5274e+08
```

As the headings indicate, the columns are the actual and predicted values of the server's CPU time, in microseconds. Finally, the program will print out the results of the regression, as well as descriptive statistics about the quality of the prediction. We will discuss these in detail later. If you execute the program with two arguments, it will train on the first file and use the second file to test the model's predictive power.

3 Requirements

In developing this technique, we sought to fulfill the following requirements, which we were not able to meet otherwise:

- Low memory and CPU consumption on large datasets. We must be able to compute results for datasets consisting of millions of samples and hundreds of thousands of classes of queries, which may be sparse or dense.
- Ability to compute results for entire datasets, not just one or a few classes of queries.
- Ability to compute on-demand, without any precomputation or preparation, over arbitrary datasets.
- Reasonably accurate results, supporting a way for humans to gauge the quality.
- Simple implementation.

Several of these requirements made many known techniques unsatisfactory. We now restate our problem and examine existing solutions.

4 Restatement of Problem; Review of Prior Art

If we were ignorant of regression, we might state the general problem intuitively as follows: given many samples of several independent (*x*) variables $x_1...x_n$ and a single dependent (*y*) variable, compute coefficients $c_1...c_n$ such that the equation $y = c_1x_1+...+c_nx_n$ describes the system's observed behavior as closely as possible.

In domain-specific terms, each *x*-variable is a query class's execution time, and *y* is the observed CPU time or other metric of interest. We now compute a coefficient for each class of queries. When multiplied by each sample of the query class's accumulated execution time, we obtain a value for that query class's CPU time. When summed together, these should approximate as closely as possible the server's observed CPU time.

This naive description is a simplification of ordinary least-squares linear regression, which fits a hyperplane through a multidimensional sample space while minimizing the sum of squared errors. Each dimension of the hyperplane is described by a slope, and there is an added intercept term. The simplest case of a hyperplane with only one *x*-variable is a line described by the equation $y = ax + b$. For example, the result of a simple least-squares linear regression against the points in Figure 2 is the equation $y = 4.032x + 31.602$, which is plotted in the figure.

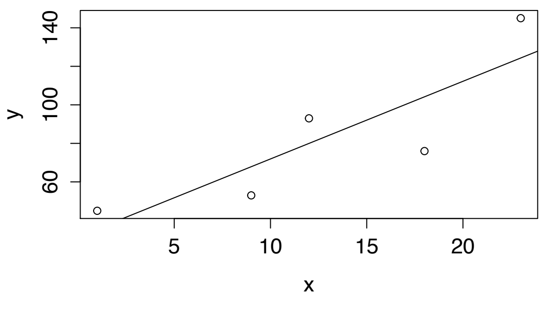


Figure 2: An example of simple least-squares linear regression

Linear regression is the *de facto* technique for solving the problem we defined. However, we found it was more general than we needed, and as a result too complex, too inefficient, and not accurate enough for our purposes. When there are multiple *x*-variables (multiple linear regression), it examines not only the correlation between the *x*-variables and *y*-variable, but also that of each pair of *x*-variables. The number of correlations is $O(n^2)$ in the number of *x*-variables. In words, multiple linear regression is computationally expensive with a large number of variables (classes of queries, in our application). But the physical reality of queries executing in a database server is that each query consumes CPU time independently, so the correlations of each pair of *x*-variables may not need to be computed.

Multiple linear regression also tends to produce poor results on large numbers of *x*-variables, because *x*-variables with large contributions to the *y*-variable are more statistically significant than minority contributors. Conventional wisdom holds that the few most significant *x*-variables usually explain most of the results, and including all *x*-variables in the computation produces a "worse fit" than discarding all but the top few. This ignores the fact that each query class does consume some amount of CPU, and we want to know that quantity. It is also common for some classes of queries to consume resources more or less constantly, and for occasional resource-intensive queries to dominate for a while. We do not want our results to be rendered useless by the inevitable outliers.

Before developing our technique, we spent several months evaluating prior art. Our goal was not only to familiarize ourselves with existing techniques, but also to generate a "golden" result that represents the state-of-the-art in quality, against which we could compare our results.

The variations of regression and other approaches that we tried produced roughly similar results. We achieved highly significant results with some of the techniques by standard statistical measures such as slope and intercept errors, the R^2 value, and Student's *T*-statistic. Most techniques produced better results if we followed the standard best practices such as including only the most significant *x*-variables.

5 Weighted Linear Regression

After studying prior art, we developed an intuitive sense that our use case is simpler and more specific, and that perhaps a different approach could meet our requirements of reasonable accuracy, efficiency, and reasonable results for insignificant *x*-variables. We worked from these requirements to develop a technique we call weighted linear regression.

We began with some simplifying assumptions, derived from key observations mentioned previously, and our knowledge of database server internals.

1. The *x*-variables all have the same dimensions and meaning: microseconds of time. As a result, we can reasonably add, subtract, divide, and multiply them together.

2. The coefficients of the *x*-variables are not likely to differ widely (by orders of magnitude), and are likely to be linearly related to metrics such as the server's CPU consumption. This is intuitively justified by observing that in many cases, queries within a database server spend much of their time executing CPU instructions. It seems reasonable to suggest that a query that runs twice as long as another one will be likely to execute about twice the CPU instructions, provided utilization is not so high that queue time becomes a significantly nonlinear effect. (This is a simplification that turns out to work well in many cases. In servers that are I/O bound, weighted linear regression has proven to be a good means of understanding how much I/O is caused by various classes of queries.)

3. In most cases a database server that experiences no demand from queries will eventually quiesce (perhaps after finishing some deferred work) and consume no CPU time. This is equivalent to saying that queries are directly or indirectly responsible for the server's entire CPU time consumption.

These assumptions led us to question whether the full complexity and cross-checking of multiple linear regression are really needed, and whether the usual regression technique will try to compute relationships between the wrong things. To motivate some intuition about this, it may be helpful to visualize some data from our sample dataset. We will plot two *x*-variables from our 001.csv dataset; the first variable is one of the larger contributors to overall query time, and the second is a minority contributor.

Figure 3 plots each of these variables against the user CPU time consumed by the server. The plots make it clear that the first variable is related to the server's CPU consumption, which we should expect because the first query class should be responsible for most of the CPU. Figure 4 is a plot of the server's CPU time. The second variable appears negatively correlated with CPU, which is unphysical. Again appealing to intuition, this helps illustrate why variables that physically must contribute to CPU time are sometimes omitted from ordinary least-squares regression because they produce "worse" results overall. A human examining such a scatterplot visually might think there is a negative relationship between the variable and CPU time—that is, the longer the query runs, the less CPU it consumes.

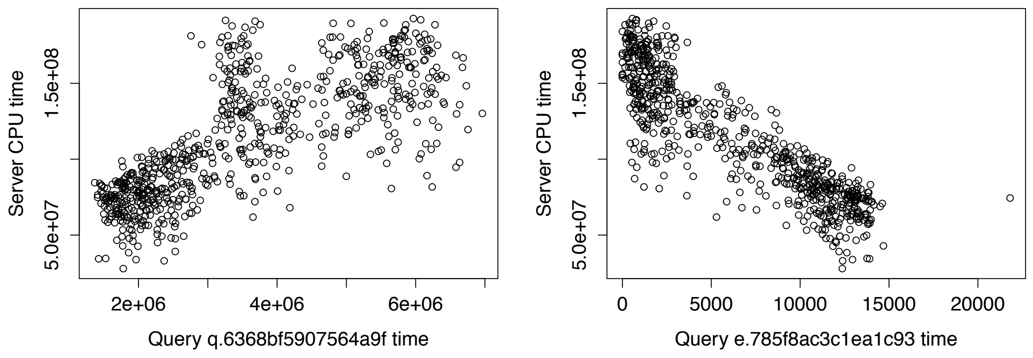


Figure 3: Query class time versus database server CPU time

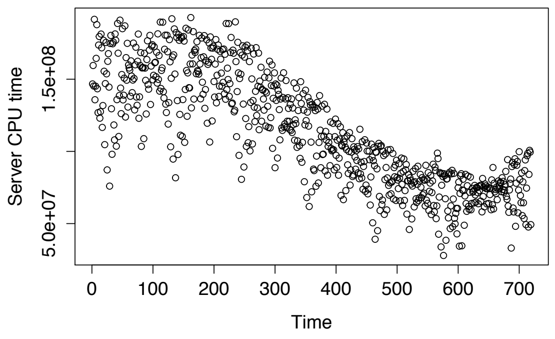


Figure 4: Database server CPU time

This is not the case, as we will now show. The apparent negative correlation between the second variable and CPU time is just an unfortunate coincidence due to when each class of queries is present. What if, instead of evaluating the correlations of the *x*- and *y*-variables, as well as the Cartesian products of all the *x*-variables and the *y*-variable, we define the algorithm roughly as follows?

For each sample of *y*- and *x*-variables:

1. Assume that the sum of the x-variables is wholly responsible for the y-variable.
2. Divide the latter by the former to obtain a ratio, whose meaning and dimension is quantity of *y*-variable per quantity of *x*-variable.
3. Multiply each *x*-variable by the ratio to obtain an estimate of the *y*-variable that should be blamed on it for this sample. Call this estimate a *z*-variable. Note that each *x*-variable now has a corresponding *z*-variable.

This leads to a transformation of the problem: instead of performing a single regression of many *x*-variables against the sole *y*-variable, we can perform multiple independent simple (single-variable) linear regressions of each pair of *(x, z)* variables to determine the best-fit slope and intercept for each class of queries.

Let us revisit the data used for Figure 3, now plotting each *x*-variable against its *z*-variable—its computed contribution to the *y*-variable instead of the entire *y*-variable. The result is Figure 5. The change in method makes it obvious that each query class's execution time has a positive relationship to the portion of CPU we assigned it.

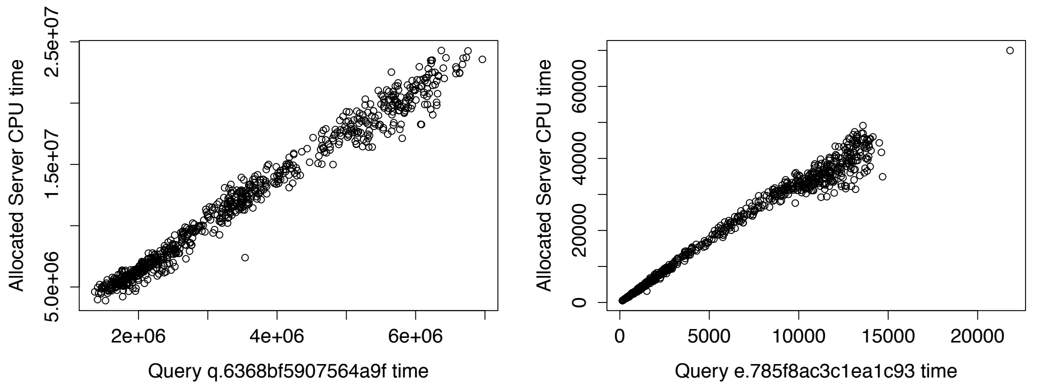


Figure 5: Query class execution time versus weighted database server CPU time

We performed this and other types of analysis, both numerical and visual, on a variety of sample datasets from real systems. Most of the techniques we tried, and their irregularities and unexplained effects, are so minor and would take so long to explain that it is not of interest here. We will summarize by saying that we needed several more modifications to avoid unphysical results and handle more cases:

1. Non-positive slopes are disallowed. A query that runs longer should consume more CPU cycles, not fewer.

2. Positive intercepts are allowed. Queries may have a relatively fixed startup cost, such as query parsing and planning, and then an incremental cost that is proportional to its execution time.

3. Query classes that are not present in an interval (the x-variable is 0) are ignored during that interval, because a query that does not execute does not have a startup cost.

4. Query classes for which we have too few samples to perform an ordinary regression (fewer than three samples) are handled with straightforward Cartesian geometry.

The result is an algorithm that is inspired by, but different from ordinary multiple linear regression. It is natural to express in a procedural programming language, but not expressible as a closed-form equation.

6 Evaluating Results

Several principles guided our assessment of our results and their quality. Our goal was to use two types of "gold standards" as references. First, we compared against the best results we achieved with the existing literature and tools available to us. Second, we measured how well our models predicted known quantities. To do this, we divided our sample datasets, trained our regression technique on one portion of them, and used the results to try to predict the held-back remainder. This approach is especially useful in guarding against over-fitting.

An important goal is not only to estimate the relationship between fine-grained measurements of query classes and aggregate metrics, but to avoid blindly assuming such a relationship exists. Statistical techniques and models are sometimes applied inappropriately, with the goal of merely producing a quantity, rather than asking whether the model is realistic at all. It is our goal to explicitly include that question in the process, so that the user of our software can determine if the assumed relationship exists, and its strength, before looking at the result.

Finally, we wanted to make the relationships and their strength and quality obvious to humans without advanced degrees in math and statistics. To achieve this, we often use visualizations such as scatterplots with a 1:1 aspect ratio. We believe that results are best judged both numerically and visually. The usual statistical measures are useful and important, but can be hard to interpret. They can also conceal important information that is obvious in simple 1:1 plots.

As a quick review, standard measurements for the quality of regression results focus on goodness of fit, standard error, and statistical significance. Goodness of fit is often measured by the correlation coefficient and the R^2 value; standard error through error terms and T-statistics for the slope and intercept. We also use mean absolute percentage error (MAPE) as a descriptive statistic. We do this by training our model on a dataset, and then using the result to predict either the training dataset or another dataset from the same system. For each sample in the dataset, we compute the actual error in the prediction, then take the mean of these. This is a metric that some people may find more intuitive (e.g. it indicates the likely relative size of the errors in the regression model).

Let us examine our algorithm's results on our 001.csv dataset, and compare it to ordinary least-squares multiple regression with some of the usual descriptive statistics. There are too many x-variables to examine all of them, but we will show the two we have been using as examples. We use the following R commands:

```
res <- lm(user_us ~ ., data=001.csv)
summary(res)
```

The (partial) results are as follows:

```
# some lines omitted
                     Estimate Std. Error t value Pr(>|t|)
(Intercept)         2.566e+07  3.641e+06   7.048 5.43e-12 ***
q.6368bf5907564a9f  1.895e+00  5.520e-01   3.434 0.000640 ***
e.785f8ac3c1ea1c93 -1.816e+02  1.882e+02  -0.965 0.334913

Residual standard error: 2199000 on 554 degrees of freedom
Multiple R-squared:  0.9976,    Adjusted R-squared:  0.9969
F-statistic:  1416 on 164 and 554 DF,  p-value: < 2.2e-16
```

A simple description of the result is that the model found a combination of coefficients that explains 99.76% of the *y*-variable (user_us). This is, academically, a very good fit. However, examining individual coefficients shows that many of them are strongly negative, which is unphysical. Adding constraints such as disallowing negative coefficients and enforcing a zero intercept make the model more physically realistic, but worsen the various quality-of-fit metrics such as the R^2 value.

In general, many existing methods were able to produce a similarly good fit, usually by discarding all but the most significant variables. However, all of the models and results were either physically unrealistic or did not meet our requirements such as computing a contribution for all query classes.

Given that we cannot find a "gold standard" result that is directly comparable to our algorithm's results, we needed to find methods of describing how good our algorithm's results are. We chose the following for their understandability and utility. Some of these may not be intuitive until we demonstrate them:

- Examine the usual descriptive statistics about each of the per-*x*-variable regressions.
- Examine scatterplots of each *x*-variable versus its *z*-variable, as shown earlier.
- Run the algorithm, and then, for each sample of the *x*-variables, predict the *y*-variable and measure the MAPE relative to its actual value.
- Ditto; then perform a linear regression of the predicted versus actual values of the *y*-variable. A perfect result would have an R^2 value of 1, a slope of 1, and an intercept of 0.
- Ditto; then plot the residuals as time series and histograms.
- Ditto the previous three points, but train on a portion of the dataset and predict against another portion.

The combination of these exercises proved helpful for finding strange or unexplained behavior, and for showing how susceptible our algorithm is to errors such as over-fitting the model to a specific set of inputs. Let us now see examples of the above. First, we run our algorithm against the test dataset, and examine the variables mentioned previously. Since we are performing multiple independent regressions for each *x*-variable, we can examine each of these with the usual statistics, especially the R^2 and *T*-statistic. The results are in Table 1 on the following page.

How should we interpret this? Let's look at the first query class. We measured 719 time intervals that contained this query class, and after regressing its execution time against the CPU time we attributed to it, the regression produced an R2 value of 98%. The slope of the fitted line is 3.65, which means that for every second this query class executes, we expect the database server to use 3.65 seconds of CPU time. That is, the query class can use more CPU time than its wall-clock time.

Query Class	Samples	R^2	Slope	T Value	Intercept	T Value
q.6368bf5907564a9f	719	0.98	3.65	0.0054	-8.345e+05	0.98
e.785f8ac3c1ea1c93	711	0.98	3.092	0.0053	993.4	2.12

Table 1: Descriptive statistics for two per-query-class weighted regressions

Is this realistic? Yes, for several reasons. One is that queries often cause background work inside the database server that happens later and is not easy to attribute to a specific query other than by techniques such as this regression. Another is that we are measuring from the time that the database server receives the entire query, until it sends the first byte back across the network to the client application. Some queries will run longer than this, so in some cases we will underestimate the query's execution time. Finally, we are probably not measuring all queries that execute on this system, so we may be blaming CPU workload on queries when it is really caused by activity we cannot measure (although that is likely to be a small effect; otherwise our regression algorithm would produce poor quality results, which we do see occasionally).

The *T*-statistic for the slope is very small, which is good. The intercept is negative, with a high *T*-statistic; as we mentioned previously, we disregard negative intercepts. In practice, intercepts have proven to influence results very little overall, even when they are large and/or have a large error term and thus *T*-statistic. A future version of our algorithm will probably force intercepts to be 0.

The only part of this result that we can compare directly to that obtained from R with ordinary multiple linear regression is the slope, which corresponds to the coefficient of the variable. The results are quite different. This can be attributed to ordinary multiple linear regression's unphysical fitting of the model to the data, resulting in negative slopes among other things.

Visually, it is quite obvious that our results are reasonable; please refer again to Figure 5, which plots the query class's execution times versus the CPU time attributed to them. In case it is confusing what our algorithm actually does, it essentially fits a straight line through these points, variable by variable. The slope and intercept of that line are shown in Table 1.

The full listing of all variables and their results is obtainable by executing our sample code; nearly all of them have very good results. This is a much better result than ordinary unconstrained regression, and matches our intuitive sense of how queries should influence CPU time in the database server.

The next quality check is to use the samples of query class execution time to try to predict the corresponding CPU usage. We know what the correct answer is, and we judge our results by how closely we match them with the mean absolute percent error (MAPE). In the test dataset, our MAPE is 5.9%. When we "train" our algorithm on the 001.csv dataset and then try to use the result to predict the 002.csv dataset (the two datasets were taken sequentially from a single database server), the result is also a MAPE of 5.9%.

Our next quantitative check of the results is to record the predicted and actual values of CPU and perform a simple linear regression, using the actual CPU as the *x*-variable and the predicted CPU as the *y*-variable. If the results are perfect, every combination of points in the resulting scatterplot will fall exactly along a line through the origin with slope 1, and the R^2 will be 1. Results on our test dataset are good: when training and predicting on the same dataset, we achieve a slope of 0.97, and the R^2 value is 0.96. When training on 001.csv and predicting on 002.csv, our slope is 1.00 and the R^2 is 0.91. This is far easier to assess visually than numerically, which leads us to our next quality control technique. The plots in Figure 6 show the actual versus predicted CPU samples for the two cases. The diagonal lines indicate where a perfect match would lie.

These plots are essentially residuals on a diagonal line. We prefer this instead of simply plotting residuals, because it becomes hard to assess the relative magnitude of the residuals, especially since most plotting libraries will set the *y* axis range relative to the magnitudes. Plotting actual versus predicted values and setting the aspect ratio to 1:1 makes the values and their relative magnitudes immediately obvious to the eye; there is no need even to label the axes.

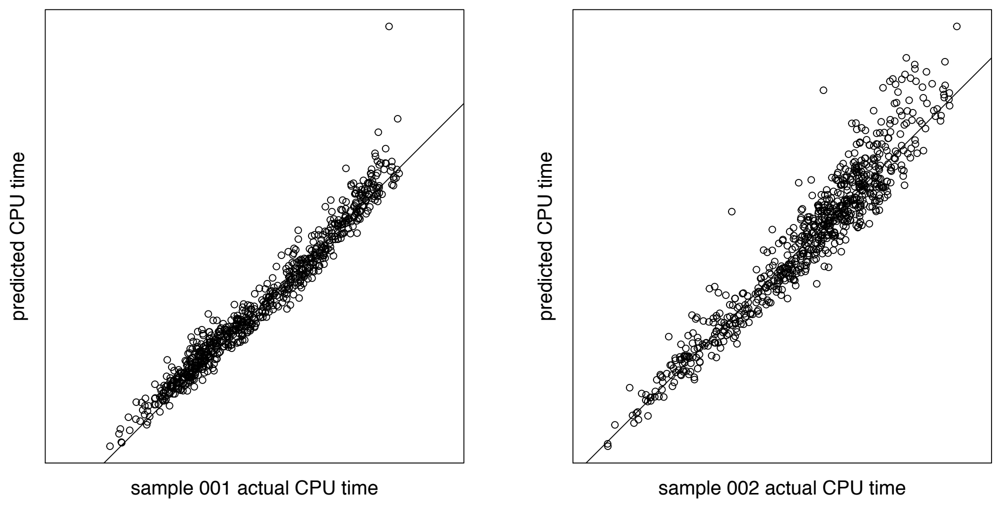


Figure 6: Actual versus predicted database server CPU time

Of course, the plot also helps humans to instantly see outlying points such as the lone outlier near the top of the first plot. These plots help illustrate the results and their quality: quite good in the first case; more dispersed in the second case as expected (and as shown by the lower R^2), but still generally good. They give additional context to numbers such as the MAPE we showed earlier: although both cases have the same MAPE, it is clear that MAPE does not fully characterize the results.

These plots omit the time dimension, so we plot the residuals as a time series as well. However, residuals themselves are difficult to put into context: is a residual of 198.3 bad? What about a residual of 1.5811e+08? It is helpful to display the relative residuals; thus we plot them as percentages, as shown in Figure 7.

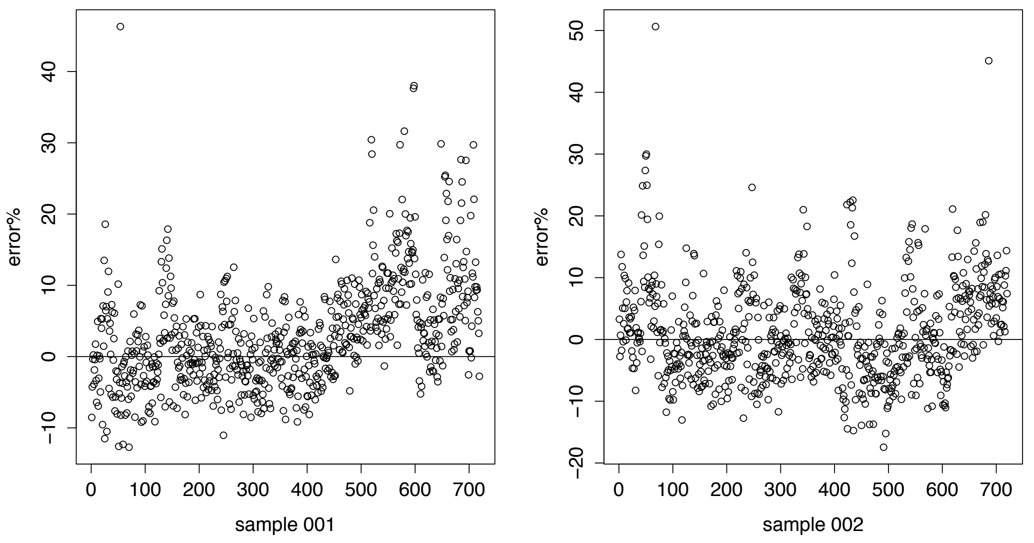


Figure 7: Residual errors as percentages

Finally, because we would like to know how the errors are distributed, we plot these residual percentages as histograms in Figure 8.

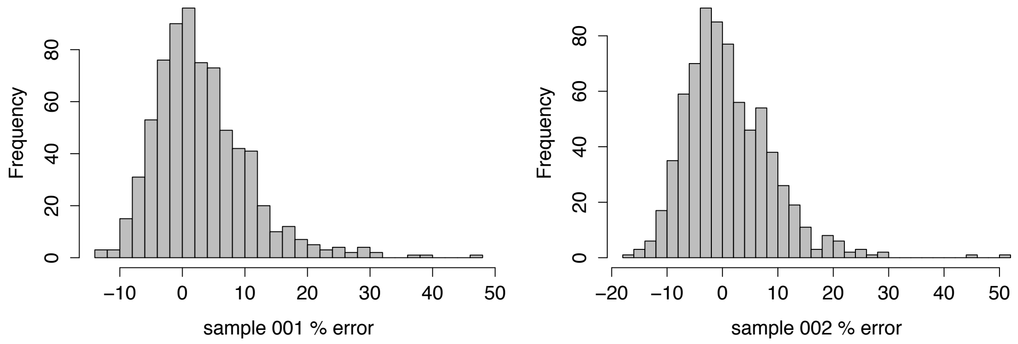


Figure 8: Histograms of percentage residual error

7 Shortcomings of the Method

We do not pretend that our algorithm is perfect, despite being pleased with the correspondence between our model and the underlying physical reality of how database servers execute queries, as well as the quality of the results themselves. There are certainly shortcomings, as well as areas where it should not be applied as-is.

The first shortcoming is that sometimes the results show things that humans know to be wrong. This is not present in the sample dataset we have used, nor can we discuss specific cases we have found due to confidentiality of the information. But it is clear after using the technique for some time that it is possible to fool the regression. Here are some cases we have found:

- Some types of queries that execute for a long time but do no work, such as `SLEEP()` queries, can be blamed for consuming CPU time. This typically happens only if these queries execute consistently. If not, then the algorithm will find evidence that contradicts the assumption that queries all consume about the same amount of CPU per unit of time, and those queries' regressions will show very poor R^2 and other values, which can serve as a hint to the human that the results are not to be trusted.

- When performing the regression against y-variables other than CPU time, it is important to think about whether there is truly a relationship between query class execution time and that variable. Sometimes the answer is not obvious, and inspecting the results is the only way to know for sure. Sometimes there is no relationship, a weak relationship, or a non-linear relationship. A case in point is the metric of rows sorted in the MySQL database server. Depending on the query execution plan, queries that run a long time might do so because of sorting many rows—but in other cases, the long time might be due to other work, and sorting rows is a small and fixed portion of the execution time.

- The model assumes that all the resource consumption is accounted for by what is measured, while this might not be the case. For example, some queries might be executing in the server in ways that are not measured. This can skew the results and/or produce worse fits.

- Due to the way we measure query execution time in our software, queries can be counted as complete while they are still executing. (We measure time to first byte.) This can skew the true query execution time and the results of the regression.

- Particularly noisy or chaotic server behavior, or changes in server behavior during the observation interval (such as a change in server settings) sometimes creates poor results.

Some of these problems are due to specifics of how we measure query execution time in our software; others are more general considerations. As usual, garbage in produces garbage out.

8 Playing Devil's Advocate

In addition to the shortcomings noted in the previous section, it is fairly easy to imagine cases where the technique might not work, or to propose other ways to get the same results. We wish to mention a few that have occurred to us, and address them briefly.

One might ask why go through all this work, only to get approximate results? Why not just measure directly the variables of interest, such as CPU per query? We would respond that it is infeasible and undesirable to measure, transmit, store, and analyze the Cartesian product of all of the dimensions of interest in the systems we manage, for a variety of reasons including performance overhead and cost. Secondly, many things of interest are not instrumented in the required way, and either cannot be so instrumented or would give misleading results. CPU per query is an example of this, as mentioned, because some queries cause the server to perform work indirectly, which can only be discovered through techniques such as ours. Additionally, once this regression technique is implemented and ad-hoc, interactive inspection of the relationship between arbitrary groups of metrics is at one's fingertips, it becomes easy to appreciate how many interesting things can be discovered that no one would think to instrument a priori.

What about variables where there is no relationship, or a nonlinear relationship? This is indeed a concern, but the answer is easy to find: simply apply the algorithm and inspect the results. They will reveal if this is the case, as it sometimes is. In fact, there's an even easier way: since we're regressing portions of overall query time against portions of overall CPU time, why not do the reverse—examine the correlation between total query time and total CPU time? This is easy to plot, and reveals whether there's a relationship, as shown in Figure 9.

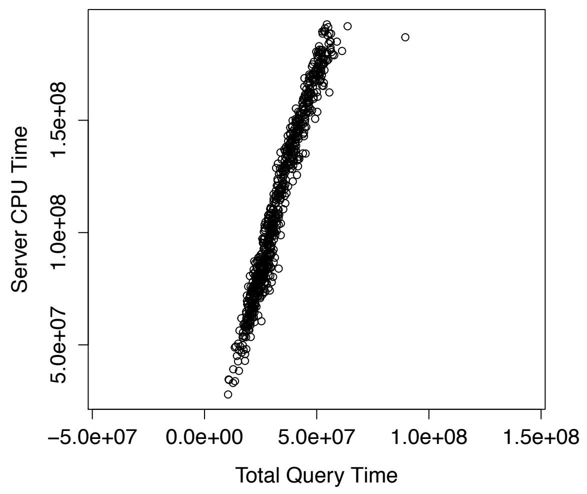
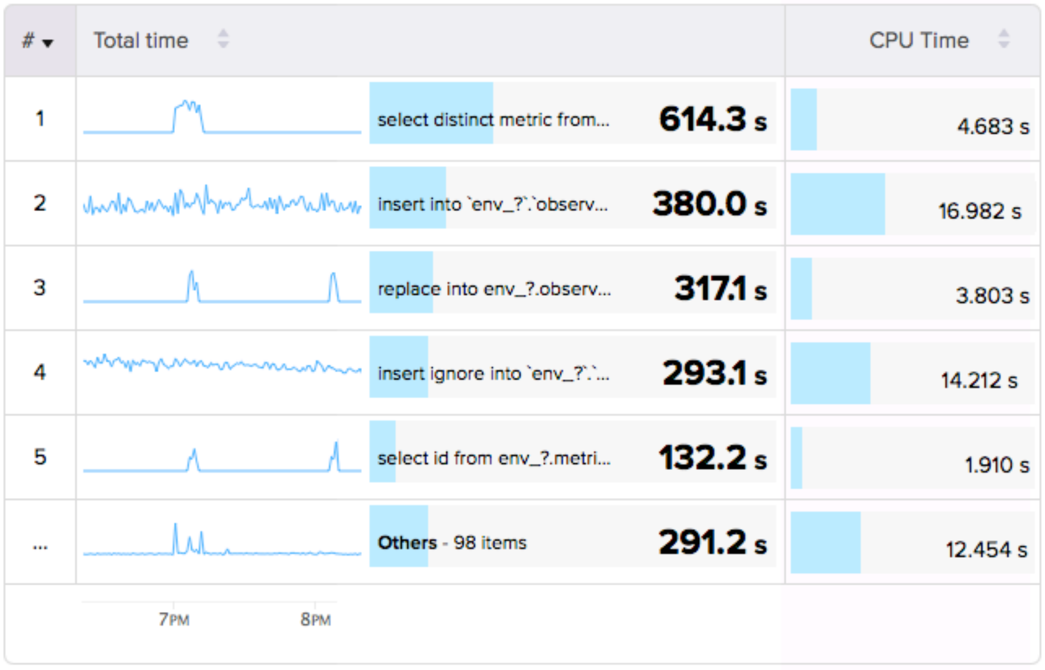


Figure 9: Total query time versus total CPU time, showing a strong relationship

An objection that specifically applies to the examples given in this paper is that query wall-clock time and CPU time are known to be non-linearly related due to queueing at higher CPU utilization, or indeed any work that is performed off-CPU, such as waiting for I/O. Again, this is certainly the case, but in conditions where that is not true, the approach yields good results, and in all cases it makes the validity of the results visually obvious. It is worth mentioning again that we developed this technique to examine relationships between arbitrary types of metrics. In addition to query wall-clock time and server CPU time, we use it ad-hoc for metrics such as query error rates and network packet retransmits.

Finally, one might ask if the technique simply shows what was already obvious—for example, if we assume that most queries consume about the same amount of CPU time per second of execution time, why not just blame the longest-running query classes for the CPU consumption on the server? This is a good intuition, but it turns out not to be true, as shown in Figure 10, which is the same as Figure 1 with the CPU column computed with weighted linear regression. This is a screen capture from a real server's queries in our application, although it is a different server from the one measured in our 001.csv dataset.



#	Total time			CPU Time
1		select distinct metric from...	614.3 s	4.683 s
2		insert into `env_?`.`observ...	380.0 s	16.982 s
3		replace into env_?.observ...	317.1 s	3.803 s
4		insert ignore into `env_?`.`...	293.1 s	14.212 s
5		select id from env_?.metri...	132.2 s	1.910 s
...		**Others** - 98 items	291.2 s	12.454 s

Figure 10: Query class execution time and CPU consumption do not always match

9 Acknowledgments

We would like to thank the following people for their help in various phases of this project: John Berryman, Mikio Braun, Neil J. Gunther, Preetam Jinka, Tim Kraft, Stein Kretsinger, Casey Lichtendahl, John Miller, Jaffray Woodriff, and Owen Zanzal.