MIDWAY 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 [4], 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 [6] 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 [2] 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 [3]. 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

Workloads	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
SEATS	10	189	9	5	12	6	6	On-line 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
YCSB	1	11	1	0	0	6	0	Scalable 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 use the "scikit-learn" [5] package for Python as our framework for evaluating different machine learning algorithms. In exploring the best way to map workloads onto benchmarks, we evaluated both clustering algorithms as well as SVM. The rest of this section discusses the results we observed with each method.

Clustering was our first approach as an unsupervised algorithm seemed to best fit the data at hand. As such, we tried the following methods, all of which are inbuilt in scikit-learn:

- K-Means
- Affinity Propagation
- Mean-Shift
- Ward Agglomerative Clustering
- DBSCAN

The clusters found by each algorithm can be seen in Figure 1. We also calculated common clustering metrics such as homogeneity, completeness, and V-measure. These results can be found in Figure 2. From the results, we notice immediately that DBSCAN does not work well with our data. Indeed, it does not find any distinct clusters at all. However, K-Means and Ward Agglomerative Clustering algorithms perform well. Both of 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.

In addition to evaluating clustering algorithms, we also evaluated Support Vector Machines, a supervised classifier. While real-world data will not be labeled and hence a supervised classifier cannot be used, it is still useful to see how effective our features are in discriminating between benchmarks. Using a simple two-fold cross-validation, we found that a simple SVM with an RBF kernel achieved 86% precision but with a very high standard deviation of 28%. However, this is still a very

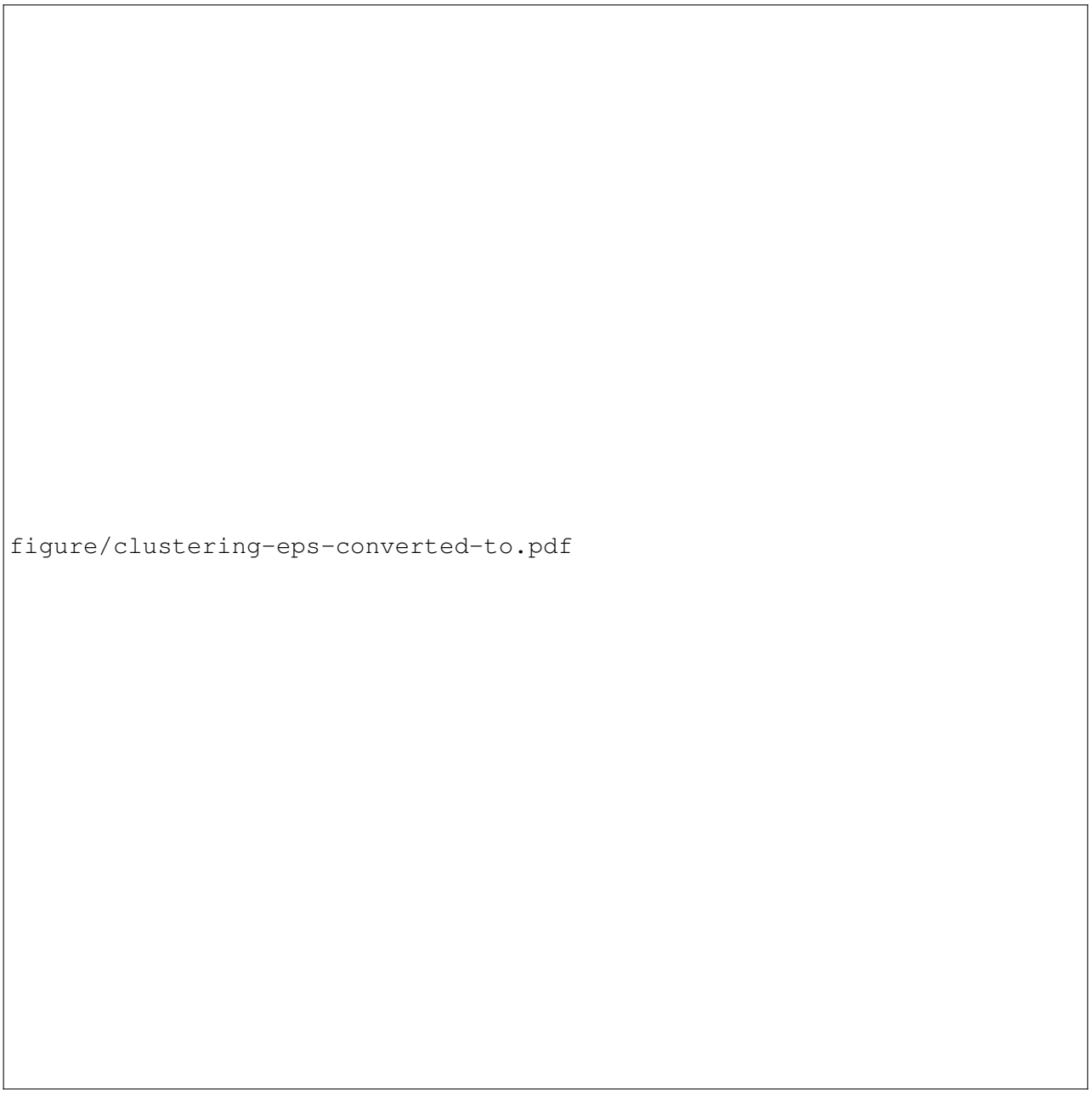

Figure 1: Clusters found by the clustering algorithms.

strong result as it indicates that our features effectively separate our data into our desired classes. We plan to validate the classifier on more sophisticated heterogeneous real-world workloads as part of our stretch goal.

The immediate next step is to build an estimator that allows us to address the second goal of this project. We already have the infrastructure required for collecting metrics and features required for solving this problem. We plan to explore different machine learning algorithms to solve this problem in the coming weeks.

0.0 1.00 1.00 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

———————————— Estimated number of clusters: 10 Homogeneity: 0.614 Completeness: 0.628 V-measure: 0.621 Adjusted Rand Index: 0.429 Adjusted Mutual Information: 0.606 [2 8 9 ..., 3 1 1] Silhouette Coefficient: 0.111

Metrics for Affinity Propogation ———————————— Estimated number of clusters: 88 Homo-

Algorithm	Homogeneity	Completeness	V-Measure
K-Means	1.000	1.000	1.000
Affinity-Propagation	0.824	1.000	0.904
Mean-Shift	1.000	0.795	0.886
Ward Agglomerative Clustering	1.000	1.000	1.000
DBSCAN	0.000	1.000	0.000

Figure 2: Performance metrics of clustering algorithms.

geneity: 0.654 Completeness: 0.317 V-measure: 0.427 Adjusted Rand Index: 0.067 Adjusted Mutual Information: 0.248 [72 8 22 ..., 30 74 39] Silhouette Coefficient: 0.082

Metrics for Mean-Shift ———————————————— Estimated number of clusters: 2 Homogeneity: 0.012 Completeness: 0.368 V-measure: 0.024 Adjusted Rand Index: -0.001 Adjusted Mutual Information: 0.010 [0 0 0 ..., 0 0 0] Silhouette Coefficient: 0.517

Metrics for Ward Agglomerative Clustering ———————————————— Estimated number of clusters: 10 Homogeneity: 0.589 Completeness: 0.635 V-measure: 0.611 Adjusted Rand Index: 0.458 Adjusted Mutual Information: 0.581 [0 4 5 ..., 4 2 6] Silhouette Coefficient: 0.097

5. Conclusion

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

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

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

[5] 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.

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