Reproducing CGPTuner: an automatic online DBMS configuration tuner based on Contextual Gaussian Process

Wuh-Chwen Hwang, Cheng-Wei Lu, Do-Men Su

1 Introduction

In this project, we reproduce CGPTuner [1], the database management systems (DBMS) auto-tuning algorithm proposed by Cereda et al. in [2]. We evaluate its effectiveness, examine the advantages and disadvantages of its design choice, and point out the key factors for building up an effective tuner that deserves future research efforts.

1.1 The Problem

Numerous applications rely on DBMS to store and retrieve data and their performance largely depends on that of the DBMS. To unleash all the performance potential of the DBMS, one needs to carefully tune numerous parameters. However, this is not a trivial task. There are hundreds of configurable parameters, composing a large space to explore. Also, parameters can have dependencies with each other in intricate ways, and changing one of them may have unexpected counter-effects. Lastly, there is not a configuration that can handle disparate workloads well. To obtain a better performance, tuners have to accommodate changing workloads. Figure 1 in CGPTuner [2] shows carefully tuned parameters significantly improve the throughput of DBMSes (figure (a), (b)), and different workloads may require disparate configurations (figure (b), (c)). Considering the number of parameters and the heterogeneity of workloads, manually tuning parameters is not trivial work.

There are offline tuners (e.g. OtterTune [7]) that conduct extensive evaluations of different configurations before going to production and perform well to predict the best configuration given a new workload. However, this kind of approach takes a long time to initialize, and such wait is undesirable or even prohibitive when many parameters are to be tuned. The goal of this project is to find an effective tuner that can start fast and observe the good configurations under fluctuating workloads.

[1] `https://github.com/c5h11oh/WiscCGPTuner`

1.2 Proposed Solution: CGPTuner

Cereda et al. present an automated configuration tuner called CGPTuner [2]. It is an online tuner, meaning little time and knowledge is needed before being in production. It uses Contextual Gaussian Process Bandit Optimisation (CGPBO) [5], which leverages Bayesian Optimization framework to comprehend the DBMS workloads that contain contextual information and Gaussian Process to construct the surrogate model and find the optimal values to tuning parameters under such workloads. CGPTuner considers the entire system stack, keeping an eye on all factors affecting the performance instead of only the DBMS itself. Lastly, instead of considering different workloads' optimizations as disparate problems, CGPTuner shares the prior knowledge with similar workloads and adapts to changing workload automatically.

2 Background

MongoDB MongoDB is a document-oriented database management system. It is categorized as a NoSQL database program, because the pieces of data it stores are JSON-like documents, as opposed to most classic relational database systems that store data as relations. Unlike relational databases, data in MongoDB does not have to follow predefined schemas, while on the other hand, unlike vanilla key-value stores, MongoDB can retrieve metadata from the documents to optimize queries running on them. MongoDB supports user-defined JavaScript functions in queries. WiredTiger is MongoDB's default storage engine. It uses multiversion concurrency control (MVCC) to support concurrent operations at the document level. Data compression, buffer management, and persistence guarantees are also done by WiredTiger. To ensure durability, WiredTiger builds database snapshots and periodically makes checkpoints. The changes from the most recent checkpoint are recorded by journal (write-ahead-logging). Buffer size (`wiredTigerCachedSizeGB`), checkpoint flushing frequency (`syncdelay`), and buffer eviction policies (`eviction_dirty_target`, `eviction_dirty_trigger`) are configurable.

YCSB The Yahoo! Cloud Serving Benchmark (YCSB)

[3] is a benchmark program that evaluates the performance of (mostly NoSQL) database systems by issuing read/write requests to a table. It is comprised of a "load" phase and a "run" phase. In the load phase, users specify the desired number of records (key-value pairs), and YCSB inserts randomly generated records to the benchmark table. In the run phase, YCSB injects the workload into the database system until either the user-defined time interval passes or a certain amount of requests are executed. There are six (A-F) preset workloads that represent different scenarios (e.g. write-intense, read-intense, read-only, etc.). Users can also customize the workload by specifying the read, update, scan, insert ratio, the number of requests, and the request distribution (i.e. existence of popular records). In addition, the number of request threads and the targeted total throughput can be configured in the run phase. In our evaluations, 10 million records are generated in the table, and the preset workload A, B, and C, which respectively represent write-intense, read-intense, and read-only scenarios, are used.

Bayesian optimization Bayesian optimization is a sequential-based method that aims to find global minimum or maximum for a black-box function. A black-box function is different from a function with an explicit form that allows us to find its gradient and use it to converge to a global minimum or maximum. Since we do not know the objective function, the Bayesian method treats the black-box function as a function and places a priori estimation ("surrogate models" in the CGPTuner paper) over it. The prior is updated by using the new function evaluations (new data points) to form a posterior distribution considering the objective function. Then, it uses the posterior distribution to form the acquisition function and chooses the next recommended point by minimizing the acquisition function. There are different acquisition functions, and we use Lower Confidence Bound (LCB) which balance between exploration and exploration of the configuration space in our project. There are different models to construct the priori estimation (surrogate model) of Bayesian Optimization, one of which is Gaussian Process. To construct Gaussian Process model, we need a kernel which is a covariance matrix of the distribution. The main contribution of CGPTuner is that it uses a newly defined kernel that splits the calculation of configuration kernel and workload kernel and adds them together, which is an implementation of Contextual Gaussian Process Bandit Optimisation (CGPBO). More details will be introduced in the evaluation section. To connect the DBMS tuning problem with Bayesian Optimization, we can observe its objective function. When we are running a DBMS experiment, what we do is to give DBMS a set of configurations (independent variable), and wait for it to return the result of its throughput or latency (dependent variable) after the workloads are fully processed. One thing worth noting is that we cannot find an explicit form of the objective function. There is not a mathematical equation for us to derive to find anything like the steepest descent direction. Therefore, the objective function is a black-box function, and only derivative-free methods such as Nether Mead method [6] or Bayesian optimization can be applied to it.

3 Design

CGPTuner is an online tuner based on Bayesian Optimization with a kernel that combines a parameter kernel and a workload kernel. This design allows it to share knowledge across different workloads.

Parameter kernel in CGPTuner includes the parameters of the entire IT stack, which includes both the DBMS and its underlying systems (e.g. JVM, Linux). For the workload kernel, its input workload information is represented by two integers: the workload ID and the number of request threads. In our experiment, we use three types of YCSB preset workloads (workload A, B, and C), and each type of workload is recognized by its workload ID. In real-world scenarios, the workload classification may be based on human knowledge (e.g. classifying into weekday workload and weekend workload). In the simulation, we simply give each YCSB workload an ID. The number of request threads represents the number of concurrent users issuing requests; as mentioned in the Background section, this can be configured at the YCSB run phase.

Figure 1 shows the system design of our CGPTuner implementation. We use two virtual machines (VM) on Google Cloud Platform to perform the experiment. They are respectively the database server and YCSB load injector/GCPTuner. The configurations and functionality of the VMs are described as follow:

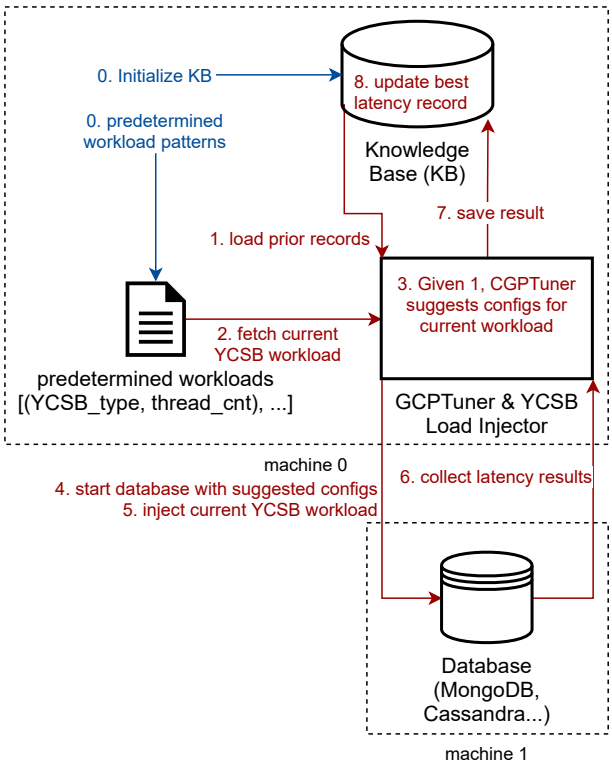
- **Database VM** represents the database server. It runs the database management system (MongoDB) and accepts commands from the YCSB VM. The type of this VM is `e2-standard-8`[2], which has 8 vCPUs and 32 GB of memory. We additional mounted an 128 GB SSD persistent disk (`pd-ssd`[3]) on this machine as the storage of the database management system.

- **YCSB VM** represents the database users and runs the YCSB program [3]. This machine is in charge of controlling the experiment process and measuring

[2] https://cloud.google.com/compute/docs/general-purpose-machines#e2_limitations

[3] https://cloud.google.com/compute/docs/disks#disk-types

Figure 1: System Design Diagram



the results of experiments. It manages the knowledge base for the tuner in memory. We choose to put the most of the CGPTuner logic in YCSB VM so that the database VM needs only to run the DBMS program, minimizing the interference caused by CGPTuner and YCSB. The machine type of this VM is `e2-highcpu-16`, which has 16 vCPUs and 16 GB of memory. We use a high-CPU machine for this VM because we want to avoid the performance bottleneck when the workload has a higher number of threads. However, we cannot allocate more than 16 vCPUs to this VM due to Google Cloud's limitation on total vCPUs allocation in the same region.

As shown in figure 1, the tuning process is categorized into the follow steps:

- **Initialization** (step 0): To initialize our experiment, we use predetermined workload patterns and initialize the knowledge base with random configurations. Since Bayesian optimization requires a few initial data in the knowledge base to suggest the next configuration, we initialize the knowledge by randomly sampling 5 configurations for each workload-thread pair.
- **Suggesting configuration** (step 1, 2, and 3): Given the previous knowledge base and the current workload information, CGPTuner uses Bayesian optimization to suggest the configuration for the current workload.
- **Setting up the database** (step 4): After CGPTuner suggests the configuration, the YCSB VM will configure the database VM with the suggested configuration. The YCSB VM uses ssh commands to change the system configurations on the database VM and to start the DBMS with suggested configuration.
- **Injecting the YCSB workload** (step 5): Injecting the YCSB workload requires two phases, loading the data to the database and running YCSB queries. The load phase is only used to insert records into the table, which is irrelevant to the performance metric (latency) we are interested in. Therefore, we run the YCSB load phase locally on database VM, while the YCSB queries in the run phase are sent from the YCSB VM, simulating the read-world scenario.
- **Collect results** (step 6): After running the simulation, our CGPTuner implementation collects the performance indicator (latency) from the YCSB reports.
- **Saving results & Updating best record** (step 7 and 8): We store the results in knowledge base as a Python dictionary. If the latest configuration has the best performance, we update the best record.
- **Repeat**: The tuner returns to step 1 and repeats this process.

4 Evaluation

We provide as many implementation details here to help others who want to implement CGPTuner on their own. We perform parameter tuning on MongoDB.

4.1 3 methods compared

In our experiment, we compare between three parameter tuning methods (these methods are mainly implemented with the help of GpyOpt package):

1. CGP (Contextual Gaussian Process): the method used by CGPTuner. There are a few things that make CGP different from the methods in the past. First is the creation of new kernel, and the second is that they fix the workload to optimize the acquisition function. About the new kernel, as we know,

Bayesian Optimization requires a multivariate distribution such as multi-variate Gaussian distribution. To model Gaussian distribution, we have to specify a kernel which is a covariance matrix. When CGP is calculating the kernel, it splits the configuration part and workload part. The mathematical formulation of the kernel in the original paper is described as follow:

$$k((\vec{x},\vec{w}),(\vec{x'},\vec{w'})) = k((\vec{x},\vec{x'})) + k((\vec{w},\vec{w'})) \quad (1)$$

, the creation of the new kernel can be done by similar method in [5](The author has a Github page dedicated for this work, which can help a lot for the implementation). Second, when CGP recommends the next configuration, it does so by first fixing the workload, and then minimize the acquisition function. This can be easily done by using the `suggest_next_locations` method in GpyOpt package.

2. GP (Gaussian Process): this method is using Bayesian optimization with the original kernel that only has configuration space. (i.e. its kernel is simply just $k(\vec{x},\vec{x'})$.) Since it only uses configurations as input for the Bayesian optimization model and no workload related information is provided, it is not able to recommend configuration according to a certain workload.

3. Random: at every iteration, Random method will randomly generate a configuration regardless of workload.

To sum up, CGP incorporates a new kernel and can recommend the configuration according to the workload, while GP cannot do so because it does not have workload $\vec{w}$ as its input.

4.2 Parameter Range

In the original paper, only the tuned parameter names are revealed. Here we will add a few more details such as the value range we use for the parameters.

We choose 15 parameters to tune in MongoDB as described in table 1. The first 4 parameters are related to MongoDB, and the rest is about OS. For simplicity, we assume that all numerical parameters only accept integer values within the parameter range. Also, for most of the parameters, there are no upper-bounds for the value range. Therefore, we use about $3\times$(the default parameter setting) as the range's upper bound. One phenomenon we observed when running experiments is that the system can sometimes crash. We think that might be because some parameters might be co-related with each other, or some parameter range values should depend on the hardware information of the machine we run on. We currently do not have enough background to deal with the system crash, so we instead give a penalty value to the experiments that have system crashes.

Parameters	
Parameter Name	Range
wiredTigerCacheSizeGB	$(1, 20)$ (GB)
eviction_dirty_target	$(1, 99)$ (%)
eviction_dirty_trigger	$(1, 99)$ (%)
syncdelay	$(1, 180)$ (second)
sched_latency_ns	$(1, 72000000)$ (ns)
sched_migration_cost_ns	$(1, 1500000)$ (ns)
vm.dirty_background_ratio	$(1, 99)$ (%)
vm.dirty _ratio	$(1, 99)$ (%)
vm.min_free_kbytes	$(1, 270000)$ (kbytes)
vm.vfs_cache_pressure	$(1, 300)$
RFS	True, False
noatime	True, False
nr_requests	$(1, 24000)$
scheduler	bfq, kyber, none, mq-deadline, deadline, cfq, noop
read_ahead_kb	$(1, 384)$ (KB)

Table 1: Parameter Range

4.3 Workload

We use the MongoBD 4.0.3 and run YCSB 0.15.0 as a load generator. We use YCSB to generate $10,000,000$ records in the table, which is about 10 GB in size. Then, we run several different workloads in a specific order to simulate the workload in a typical day. There are three kinds of workloads used in our evaluation:

1. Workload A : Update heavy workload with 50% read ratio and 50% write ratio.

2. Workload B : Read heavy workload with 95% read ratio and 5% write ratio.

3. Workload C : Read only workload with 100% read ratio and 0% write ratio.

Aside from the different workloads that have various read/write ratios, we also let the workload run with a different number of threads to simulate the high contention at the peak time. As shown in figure 2, we run the workload in a total of 10 cycles. Each workload type with a specific thread number will run for 5 iterations before the next workload-thread pair is run. Each iteration runs for 2 minutes, so the whole experiment of 10 cycles will run for 12 - 13 hours.

Figure 2: workload cycle for the experiment

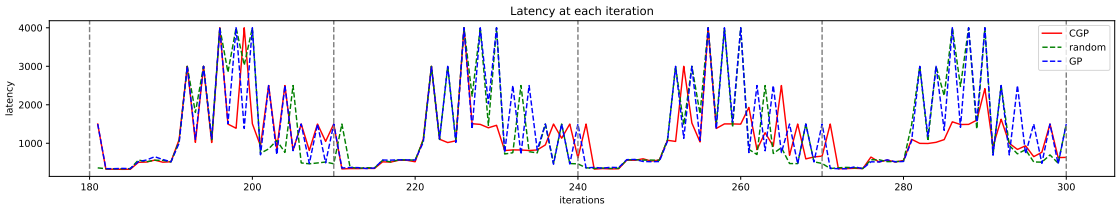


Figure 3: latency for the last four cycle

Average Latency (us) of Last 4 Cycles			
Workload and Thread	CGPTuner	GP	Random
Workload A 60 Threads	**1466**	2015	1860
Workload A 80 Threads	**2035**	3216	3008
Workload B 30 Threads	1017	**699**	1090
Workload B 50 Threads	1224	**1140**	1447
Workload C 20 Threads	511	**410**	412
Workload C 40 Threads	**543**	551	551

Table 2: table for average latency

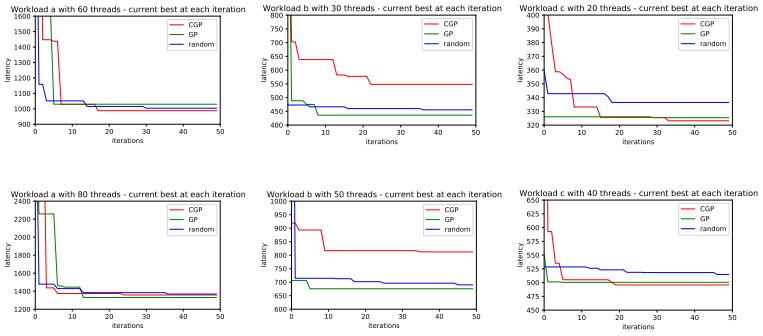


Figure 4: Best latency before each iteration under different workloads

4.4 Result

To evaluate the performance of each method, we decide to use latency (us) per operation as the metric for database performance.

In figure 3, we look at the latency per operation for the last for cycles. Cycles are separated by the grey dash lines. We can see in the graph that at the 7th cycle (from iteration 180 to iteration 210), there is not a big difference among the three methods. However, we can see that starting from the 8th cycle (from iteration 210 to iteration 240), the performance of CGP starts to become better than the other two. In the last cycle (from iteration 270 to iteration 300), the latency of CGP is often the lowest among the three for different workloads. We conclude that this is because the knowledge base of CGP keeps growing. One advantage about CGP is that at each iteration, it can record the configuration and its corresponding latency in its knowledge base and then acquire better recommendations because it will have more sample points for the Gaussian Distribution.

To further analyze the performance of each method for each workload, we take the experiment results of the last four cycles, and then calculate the average latency for each workload-thread pair. The result is shown in table 2. First, we can see that for each workload-thread pair, the best performance is achieved by either CGPTuner or GP. This tells us that Bayesian optimization is able to recommend good configurations compared to some randomly generated configurations. The second thing we can see is that CGPTuner is the best when it is dealing with workload A and a high number of threads. We think this tells us that CGPTuner is good at heavy and high contention workload because workload A is composed of 50% read and 50% write. This result is in line with the results in the original paper [2].

We now look at the best latency before each iteration in figure 4. What it shows us is the best latency we observe before an iteration for each workload. Therefore, the best latency keeps decreasing. For workload C and workload A with 60 threads, CGP is able to find a better configuration in the experiments, which makes its latency the lowest among the three at the end. For workload B, CGP is struggling to lower its best latency. The result shows two things. First, CGP can perform nicely on heavy workloads such as workload A. Second, it has a better ability to keep improving the current best latency at later iterations.

5 Discussion and Related Work

From the works we collected, we believe DBMS autotuner is a relatively new field with plenty of debates going on. Not only do the tuning methods differ from each other, but the definitions of the tuning problem are also not the same either. For example, CGPTuner is the first work we have seen that claims tuning the entire IT stack make more sense. Some tuners aim to optimize one workload at a time, while some believe configurations of similar workloads can have similarities as well. There is no clear evidence showing one is better than the others, and for the latter case, what indicators effectively determine the similarity of workloads has not been systematically evaluated yet, at least in CGPTuner's paper (they arbitrarily determine thread count and read-write-ratio as the workload characteristics). Regardless of the method, we would like the tuner to perform well under changing workloads. How well do those changing workload patterns in the experiments relate to real-world scenarios is the next question. We here list some tuners and point out their similarities and differences from CGPTuner.

iTuned [4] is an online tuner that is very similar to CGPTuner. It uses Bayesian Optimization to tune the database system and Gaussian Process to model the response surface, the performance (goal) function. As an online tuner, it also balances between exploring new configurations and exploiting current best configurations

when choosing the next configuration. However, iTuned does not consider the relevancy of similar workloads' configurations. Instead, it builds a response surface for each distinct workload. Another distinction is that iTuned does not configure parameters of the entire IT stack but only that of the DBMS.

OtterTune [7] is an offline tuner that uses supervised and unsupervised machine learning methods to relate unseen workload with prior knowledge and determine the best configuration. However, to bootstrap an offline tuner, a huge amount of data are required to construct the knowledge base (the author of OtterTune [7] used "over 30k trials per DBMS"), which can take months to collect. One highlight that differentiates OtterTune from other tuners is that it determines the most influential parameters to tune. For CGPTuner and most other tuners, the subset of parameters to be tuned is arbitrarily determined by the author of the tuners.

OpenTuner [1] is a framework to build an online tuner. It is designed to be highly customizable so that domain-specific searching processes can be integrated and the tuning targets can be specified. Its core is an AUC Bandit Meta Technique (multi-armed bandit) that leverages multiple search techniques (e.g. differential evolution, pattern search, hillclimbing, and random) at the same time. Techniques getting better results will be allocated more tests. Since it is highly customizable, it can be configured to tune the entire IT stack, if desired.

BestConfig [8] is an online tuner that tries to find the best configuration for a single workload under constrained resources. It uses the divide-and-diverge sampling method and the recursive bound-and-search algorithm to search in the configuration space.

6 Future work

Throughout the project, we discover some problem definitions, design decisions, and evaluation methods are not backed with clear reasoning or compelling evidence. The clarifications of these backgrounds can be the future research directions. Due to the time constraint, some interesting amendments to our CGPTuner implementation and their performance evaluations are not conducted in this project. We also list them out as future work.

6.1 Future Research Direction

What is the tuning problem

1. DB v.s. IT stack: Is tuning the entire IT stack the right way to go? We can use the same tuning method but only tune the DBMS parameters and see how it performs.
2. Determine workload characteristics: Under the DBMS tuner's context, evaluate the most important workload attributes that determine the similarity between workloads.
3. Model the real-world: What are the real-world scenarios that DBMS tuners are needed? Are the tuners' changing workload patterns in line with what happens?

Best approach to tackle the problem

1. Online versus offline performance: Offline algorithms incur a higher initial cost but perform better. How long do online algorithms take to perform as well as their offline counterparts, and is this trade-off worth it?
2. Online versus offline difference: online algorithms such as CGPTuner still require some initial knowledge to start. Does this blur the line between online and offline algorithms?

Tuners implementation - parameters

1. Can we let tuners decide the subset of parameters to tune, instead of deciding by the tuner authors' or the database administrators' discretion?
2. There are valid parameters that could cause the DBMS crashes. How do we select the good range for parameters? When crashes occur, how do we evaluate the performance of this configuration and give appropriate feedback to the tuner algorithm?

6.2 Our Future Work

CGPTuner evaluations - hardware

1. Use local NVMe SSDs to store database data.
2. Use bare-metal machines to run evaluations.

CGPTuner implementation - software

1. Integrate throughput into the performance (goal) function
2. Run that tunes Cassandra.

7 Role of Each Team Member

All three of us write this project report.

1. Wuh-Chwen Hwang : Implemented the code that interfaces the tuner and the system parameters. Conducted experiments. Made the poster.

2. Cheng-Wei Lu : Implemented CGPTuner, GP tuner and random configuration models in the experiments. Conducted the experiments and analyzed the ending results of them.

3. Do-Men Su : Implemented the function in CGP-Tuner that controls the database VM. Studied the database and system parameters.

References

[1] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[2] S. Cereda, S. Valladares, P. Cremonesi, and S. Doni. Cgptuner: A contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions. *Proc. VLDB Endow.*, 14(8):1401–1413, Apr. 2021.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[4] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

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

[6] J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[7] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024, 2017.

[8] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350, 2017.