CGPTuner: a Contextual Gaussian Process Bandit Approach for the Automatic Tuning of IT Configurations Under Varying Workload Conditions

Stefano Cereda
Stefano Valladares
Paolo Cremonesi
name.surname@polimi.it
Politecnico di Milano
Milan, Italy

Stefano Doni
stefano.doni@akamas.io
Akamas
Milan, Italy

ABSTRACT

Properly selecting the configuration of a database management system (DBMS) is essential to increase performance and reduce costs. However, the task is astonishingly tricky due to a large number of tunable configuration parameters and their inter-dependencies. Also, the optimal configuration depends upon the workload to which the DBMS is exposed. To extract the full potential of a DBMS, we must also consider the entire IT stack on which the DBMS is running, comprising layers like the Java virtual machine, the operating system and the physical machine. Each layer offers a multitude of parameters that we should take into account. The available parameters vary as new software versions are released, making it impractical to rely on historical knowledge bases. We present a novel tuning approach for the DBMS configuration autotuning that quickly finds a well-performing configuration of an IT stack and adapts it to workload variations, without having to rely on a knowledge base. We evaluate the proposed approach using the Cassandra and MongoDB DBMSs, showing that it adjusts the suggested configuration to the observed workload and is portable across different IT applications. We try to minimise the memory consumption without increasing the response time, showing that the proposed approach reduces the response time and increases the memory requirements only under heavy-load conditions, reducing it again when the load decreases.

PVLDB Reference Format:
Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. CGPTuner: a Contextual Gaussian Process Bandit Approach for the Automatic Tuning of IT Configurations Under Varying Workload Conditions. PVLDB, 14(8): 1401-1413, 2021.
doi:10.14778/3457390.3457404

1 INTRODUCTION

A modern database management system (DBMS) has hundreds of tunable configuration parameters that control its behaviour[12]. Selecting the proper configuration is crucial to improve performance or reduce cost. Manually finding well-performing configurations, however, can be a daunting task, since the parameters often behave in counter-intuitive ways and have inter-dependencies. Furthermore, a DBMS sits on top of a complex IT stack which comprises several layers, like the Java Virtual Machine (JVM) or the Operating System (OS). Each layer has its tunable parameters, which affect the final behaviour of the DBMS, as we show in Section 2. To unlock the full performance potential of a DBMS, we have to tune the entire IT stack jointly.

Unfortunately, we cannot run an extended search and find the optimal configuration, which is the best one for our particular combination of DBMS, OS and hardware. Even if we had an infinite budget to run this search, we would still find a suboptimal solution as the optimal configuration depends upon the particular workload to which the DBMS is exposed.

We could even imagine running an extensive search to find the optimal IT stack configuration for each particular workload, or at least a well-performing configuration for each workload. However, all this knowledge would become obsolete pretty fast, as new software versions are released, changing the effects of the parameters. Furthermore, new software releases also modify the available parameters, increasing the complexity of reusing old knowledge bases, which lack information about novel parameters.

In this paper, we propose CGPTuner, an automated configuration tuner based on *Contextual Gaussian Process Bandit Optimisation* [16] (CGPBO), a machine learning optimisation algorithm specifically developed to deal with tasks with contextual information, just like the DBMS workload, without having to rely on a knowledge base. CGPBO is a contextual extension to the Bayesian Optimisation framework. Bayesian Optimisation has already been successfully applied to the performance autotuning problem [2, 5, 10, 12, 14]. CGPTuner successfully tunes the configuration of an IT system while considering multiple layers of the IT stack and the current workload and, more importantly, it does not rely on a previously collected knowledge base, since collecting such knowledge bases becomes practically unfeasible when dealing with many layers.

We evaluate CGPTuner using the MongoDB[1] and Cassandra [17][2] DBMSs. We use Yahoo! Cloud Serving Benchmark (YCSB) [9][3] to simulate three different workload patterns. We let CGPTuner control several configuration parameters of the DBMS, the JVM and the

Proceedings of the VLDB Endowment, Vol. 14, No. 8 ISSN 2150-8097.
doi:10.14778/3457390.3457404

[1]https://www.mongodb.com
[2]https://cassandra.apache.org
[3]https://ycsb.site/

OS. In all the considered patterns, CGPTuner understands the problem in less than thirty tuning iterations, adapting the configuration to the incoming workload.

We start by analysing the autotuning problem in Section 2, then we explain our tuning algorithm (CGPTuner) in Section 3. In Section 4 we go over the evaluation methodology and report and comment on the obtained results.

2 MOTIVATION AND RELATED WORK

Tuning a DBMS configuration is very tricky: the tunable parameters interact in complex ways, live in an enormous search space, and their effect depends on the workload to which the DBMS is exposed.

However, tuning the DBMS configuration is only a part of the story. A DBMS sits on top of (at the very least) an Operating System, which, in turns, has its tunable parameters. To highlight the effect of these parameters, we ran a series of experiments on the MongoDB[1] and Cassandra[2] DBMSs using the YCSB load injector[3]. We modified the values of the tunable parameters while measuring the throughput of the DBMS. The results are reported in Figure 1.

In Figure 1a, we modify the cache size of MongoDB and the `vm.dirty_ratio` parameter of the Linux kernel. Correctly setting the latter parameter gives a significant boost to the performance of the DBMS: without considering the entire IT stack we would have lost this improvement. In Figure 1b, we use Cassandra, which is written in Java. The Java Virtual Machine has its tunable parameters, here we tune the number of concurrent garbage collection threads alongside the `read_ahead_kb` Linux parameter. The read-ahead has an impressive effect on the performance of Cassandra and selecting an improper value destroys the performance. In Figure 1c, we modify the two same parameters, but using a different YCSB workload. More precisely, we move from an update-heavy workload to a read-only workload. The effect of the two parameters is severely different in the two workload conditions.

These examples motivate the need to consider the entire IT stack and the current workload when tuning a DBMS. To further increase the complexity, we could consider other tunable parameters like the compilation flags of MongoDB, the version of the JVM or the kind of cloud instance on which to run the experiments. However, increasing the number of considered layers increases exponentially the number of parameters and the problem's complexity.

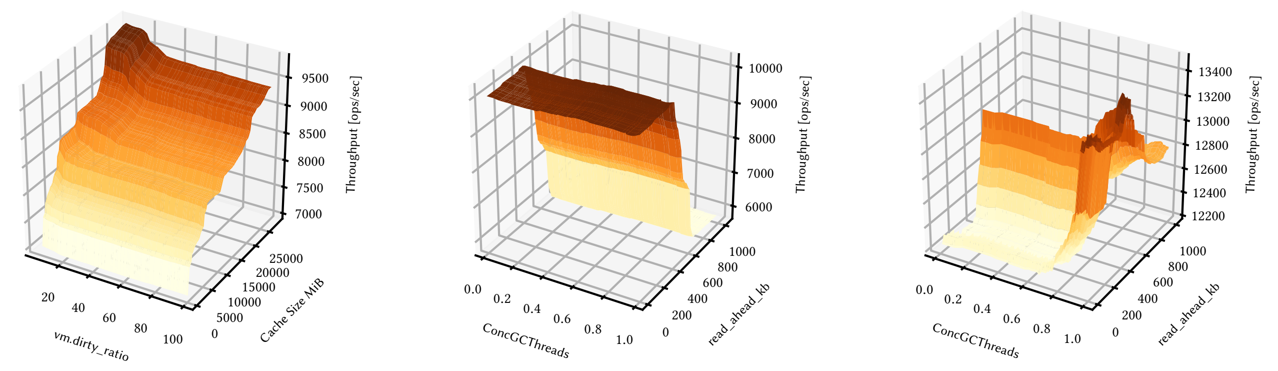
Notice that, if we were to tune the entire IT stack while looking only at the DBMS, we might destroy the system's performance, as some OS settings that are beneficial to the DBMS could be detrimental to other services that are running on the same machine. However, a proper selection of the target performance metric to be optimised is sufficient to avoid this problem. As a simple example, we can consider a single server running a DBMS and a Java backend application. If we tune the OS so to increase as much as possible the DBMS performance the backend would suffer, leading to a poor user experience. On the other hand, if we tune to minimise the response time of some user-facing services we will convergence toward configurations that allow both the DBMS and the backend application to work at their best.

The DBMS autotuning problem has been studied quite a lot. iTuned [12] works by creating a response surface of the DBMS performance with Gaussian Processes and using this model to select the next configuration to test. However, a different response surface is built for each workload, without sharing potentially useful information.

In their seminal paper [27], Van Aken et al. introduced OtterTune: a machine learning solution to the DBMS tuning problem. OtterTune leverages past experience and collects new information to tune DBMS configurations: it uses a combination of supervised and unsupervised machine learning methods to (1) select the most impactful parameters, (2) map unseen database workloads to previous workloads from which it can transfer experience, and (3) recommend parameters settings. The key aspect of OtterTune is its ability to leverage past experience to speed up the search process on new workloads. However, to do so, it requires to have an extensive collection of previous experiments. To fully leverage the OtterTune approach, all these experiments should contain all the available parameters. This might be feasible when considering only the DBMS, but it gets more and more complex as we consider more layers of the IT stack since the dimension of the search space grows exponentially. As an example, in [27] the authors had to collect "over 30k trials per DBMS" just to bootstrap OtterTune. Even with very short measurement periods of 5 minutes per trial, this results in more than three months of computation just to collect the initial knowledge base. After that, they proceed to run the tuning sessions, which lasts 60 iterations. Conversely, our tuner does not require any initial knowledge-gathering, and it is capable of suggesting well-performing configurations already after 30 iterations. Furthermore, OtterTune is exploring a single layer of the IT stack, whereas we want to consider as many layers as possible so to extract all the potential performance. As the number of layers increases, so does the complexity of collecting a knowledge base.

Furthermore, this abnormous knowledge base should be updated periodically to reflect changes in hardware components and software versions. Different software versions react differently to the same configurations, and new software versions introduce new tunable parameters. To take these parameters into account, one would have to periodically re-build the knowledge base from scratch.

In short, we would like to tune the entire stack to extract all the performance, but building a knowledge base on the entire stack would be too expensive. Thus, our goal is to design a tuning algorithm able to consider the entire IT stack and continuously adapt to the current workload, but without needing a previously collected knowledge base. For this reason, we compare our solution only with tuning methodologies that proceeds in an online fashion and do not rely on a knowledge base, gathering all their knowledge as the tuning proceeds. Thus, the tuners are faced with two opposite requirements, as they need to *explore* the search space (building up their knowledge) and, at the same time, they need to be very good to *exploit* the (limited) knowledge they have collected up to a certain point, suggesting configurations which perform well and we can use as the tuning proceeds. As shown in Section 2, we build our solution on top of Bayesian Optimisation, which is a technique specifically designed to handle the exploration-exploitation tradeoff, with strong theoretical guarantees about its convergence and data-efficiency [21]. We thus compare the proposed solution against a naive random tuner, OpenTuner [3] and BestConfig [30].



(a) MongoDB with update-heavy workload **(b) Cassandra with update-heavy workload** **(c) Cassandra with read-only workload**

Figure 1: Throughput of a DBMS as a function of some of its tunable parameters. In (a) we use MongoDB and vary its cache size and the `vm.dirty_ratio` Linux parameter. In (b) we use Cassandra, varying the number of threads for the concurrent garbage collector of the JVM (expressed as a percentage of the available cores) and the `read_ahead_kb` parameter of the Linux kernel. In (c) we repeat the experiment of (b), but using a different YCSB workload.

OpenTuner is a framework for building domain-specific program autotuners. Its core idea is to avoid focusing on a single search technique. Instead, it provides several search algorithms (like genetic algorithms, hill climb and multi-armed bandits) and dynamically decides which one to use. When tuning a program, OpenTuner starts by randomly trying different techniques, and, as the tuning proceeds, it allocates a larger proportion of tests to better performing techniques. In this way, OpenTuner selects the best-performing search algorithm for the specific program it is tuning. Nonetheless, many works reported that (under the right circumstances) a simple random search can perform as well as more sophisticated techniques, and is indeed an effective tool for exploring the space of the available configurations [1, 6, 8, 15].

BestConfig is an autotuning system for automatically finding the best configuration setting within a resource limit for a deployed system under a given application workload and is based on an iterative sampling strategy.

3 PROPOSED MODEL

In this section, we start by formalising the configuration autotuning problem, then we describe Bayesian Optimisation and its contextual extension, and finally we describe the main contribution of this work: CGPTuner.

The goal of our optimisation process is to find the configuration vector $\vec{x}$ in the configuration space X and apply it to the IT system so to optimise a certain performance indicator $y \in \mathbb{R}$. We want to select $\vec{x}$ by taking into account the particular workload $\vec{w} \in W$ to which the system is exposed, where W is the space of the possible workloads and $\vec{w}$ is a description of the workload provided by a particular workload characterisation methodology. y can be any measurable property of the system (like throughput, response time, memory consumption or a combination thereof).

Since the workload evolves over time, we want to update the suggested configuration as well. At time t, the optimisation algorithm will suggest a candidate configuration $\vec{x}_t$ which is tailored to the current workload $\vec{w}_t$. Applying $\vec{x}_t$ to the system under workload $\vec{w}_t$ results in a performance measurement y_t.

The tuning process works as depicted in Figure 2, and it is repeated iteratively as time passes. We start ① by measuring and characterising the initial workload $\vec{w}_0$. Then, we feed this information to the tuner, which has access to the knowledge base (KB) of previous experiments ② (which is initially empty) and uses both the KB and the current workload to suggests a candidate configuration vector $\vec{x}_0$. We apply the configuration to the system ③ and measure the associated performance y_0 ④. At this point, we have concluded the first iteration of the tuning process, and we store the obtained information in the KB ⑤. Now we iterate the process: we measure the new workload $\vec{w}_1$ ① and consider the previous result ② to obtain the new configuration $\vec{x}_1$ ③, which results in performance y_1 ④. Notice that, at iteration i, the tuner can exploit all the information of previous iterations: $(\vec{x}_{0,\dots,i-1}, \vec{w}_{0,\dots,i-1}, y_{0,\dots,i-1})$. However, no other knowledge is required.

In this work, we take for granted the availability of a numerical description of the workload $\vec{w}$, which we assume comes from an external workload characterisation module. As en example, we can use the workload characterisation provided by Ottertune [27], Peloton [18] or even a reaction-based characterisation used in compiler autotuning [7].

3.1 Bayesian Optimisation

Bayesian Optimisation (BO) is a powerful tool that has gained great popularity in recent years. An extensive review of BO can be found in [21], here we give just a brief introduction, visually summarized in Figure 3.

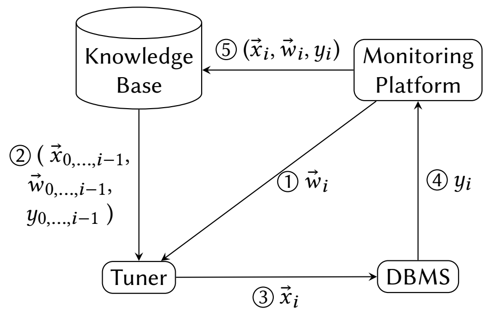

Knowledge Base
⑤ $(\vec{x}_i, \vec{w}_i, y_i)$
Monitoring Platform
② ($\vec{x}_{0,\dots,i-1}$, $\vec{w}_{0,\dots,i-1}$, $y_{0,\dots,i-1}$)
① $\vec{w}_i$
④ y_i
Tuner
DBMS
③ $\vec{x}_i$

Figure 2: Tuning process and architecture.

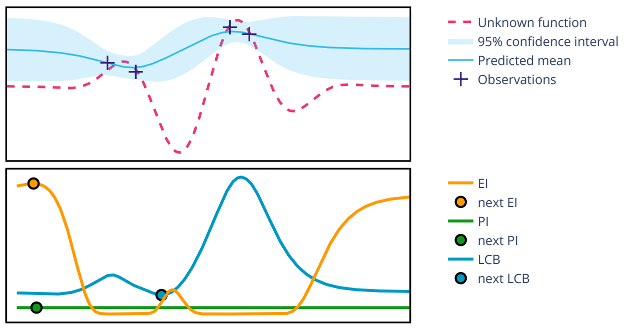


Figure 3: Bayesian Optimisation. We want to minimise the unknown function and have already observed 4 points, which we use to compute the GP predicted mean and uncertainty. The two values are combined in the acquisition functions to select the next point to evaluate. Different Acquisition functions select different points.

Formally we want to optimise an unknown objective function f:

$$\vec{x}^* = \arg\max_{\vec{x} \in X} f(\vec{x}) \tag{1}$$

where f has no simple closed-form but can be evaluated at any arbitrary query point $\vec{x}$ in the domain. The evaluation produces noisy observations $y \in \mathbb{R}$, and usually is quite expensive to perform. Our goal is to converge toward good points so to optimise f quickly. Moreover, we want to avoid evaluating points that lead to bad function values. In DBMS terms, we want to find a configuration that optimises a specific performance indicator, and simultaneously (1) explore the configuration space to gather knowledge, and (2) exploit the gathered knowledge so to converge toward well-performing configurations quickly.

Notice that Equation (1) is equivalent to the DBMS autotuning problem that we exposed at the beginning of the section, except for the workload dependence which we are not considering for now.

BO is a sequential model-based approach for solving the optimisation problem of Equation (1). Essentially, we create a surrogate model of f and sequentially refine it as more data are observed. Using this model, we iteratively compute the value of an acquisition function a_i, which is used to select the next point $\vec{x}_i$ to evaluate. Intuitively, the acquisition function evaluates the utility of candidate points for the next evaluation of f by trading off the exploration of uncertain regions with the exploitation of promising regions. As the acquisition function is analytically derived from the surrogate model, it is straightforward to optimise. Thus, BO has two key ingredients: the surrogate model and the acquisition function. In this work, we focus on Gaussian Processes (GPs) as surrogate models, which is a popular choice in BO [19, 21, 29]. For the acquisition function we follow the GP-Hedge approach presented in [13], which, instead of focusing on a specific acquisition function, adopts a portfolio of acquisition functions governed by an online multi-armed bandit strategy. The idea is to compute many different acquisition functions at each tuning iteration, and progressively select the best one according to previous performance.

A GP(μ_0, k) is a nonparametric model fully characterised by its prior mean function $\mu_0 : X \longrightarrow \mathbb{R}$ and its positive-definite kernel or covariance function, $k : X \times X \longrightarrow \mathbb{R}$. Let $D_i = \{(\vec{x}_n, y_n)\}_{n=0}^{i-1}$ be the set of past observations and $\vec{x}$ an arbitrary test point. The random variable $f(\vec{x})$ conditioned on observations D_i follows a normal distribution with a mean and variance functions that depend on the prior mean function and the observed data through the kernel.

Essentially, the GP is a regression model that is very good at predicting the expected value of a point and the uncertainty over that prediction. In BO, the two quantities are combined by the acquisition function to drive the search process. The GP prediction depends on the prior function and the observed data through the kernel. The prior function controls the GP prediction in unobserved regions and, thus, can be used to incorporate our prior beliefs about the optimisation problem. The kernel, instead, controls how much the observed points affect the prediction in nearby regions and, thus, governs the predictions in the already explored regions. In this work, we use a Matérn 5/2 kernel, which is the most widely used class of kernels in literature [21].

3.2 Contextual Gaussian Process Bandit Optimisation

Our assumption so far has been that the performance f can be expressed as a function of the configuration $\vec{x}$ only. However, the performance of a DBMS also depends on others, uncontrolled, variables; such as the workload $\vec{w}$ to which the application is exposed.

BO has been extended to handle such situations [16]. The key idea is that there are several correlated functions $f_{\vec{w}}(\vec{x})$ that we want to optimise. Essentially, the data from a workload $\vec{w}$ can provide information about another workload $\vec{w}'$. To capture this, we define a new kernel function that works over configuration and workload pairs: $k((\vec{x}, \vec{w}), (\vec{x}', \vec{w}'))$. This new kernel is formalised as the sum of two kernels defined over the configuration space X and the workload space W, respectively:

$$k((\vec{x}, \vec{w}), (\vec{x}', \vec{w}')) = k(\vec{x}, \vec{x}') + k(\vec{w}, \vec{w}') \tag{2}$$

The functions sampled from a GP with a covariance function as the one above have an additive form made of two components: $f = f_{\vec{x}} + f_{\vec{w}}$. The $f_{\vec{w}}$ component models overall trends among workloads, while the $f_{\vec{x}}$ models configuration-specific deviation from this trend. Similarly to what we did for the configuration kernel, we use a Matérn 5/2 kernel for the workload kernel.

Essentially we are saying that we expect the performance of a certain configuration-workload pair to be correlated to the performance of nearby configurations and workloads. As the two kernels are combined using a sum operation, we consider two points similar when they have either a similar configuration or a similar workload. By multiplying the kernels, we would instead consider two points as similar when they have both a similar configuration and a similar workload. We will explore these two possibilities in Section 4.5. In other terms, by using this kernel we are enlarging the GP space with the workload characterisation component, but, when optimising the AF, we only optimise the configuration subspace. What we obtain is a suggested configuration which is tailored for the current workload, and this configuration is selected by leveraging all the configurations we have tested in the past, even on different workloads.

3.3 Matching Prior Functions to Workloads

Bayesian Optimisation is, as the name suggests, a Bayesian technique. Nonetheless, it is often used just as an optimisation technique, without giving the proper attention to the Bayesian component.

As we explained above, BO works by iteratively suggesting points that optimise an acquisition function, which is computed starting from the value predicted by a regression model and from the prediction uncertainty. However, the regression model is a Gaussian Process, which derives its predictions (or posterior distribution) by combining the observed values with a prior distribution.

In other terms, when we ask the GP to predict the value of a configuration which is very different from any previously observed one, it will resort to the prior distribution. In most of the implementations, the prior distribution is a zero mean, unitary variance normal distribution. If we are trying to maximise the throughput of a DBMS, the GP will predict that any unknown configuration will likely destroy the DBMS performance. This impacts heavily on BO, which will avoid the exploration of uncertain regions.

To easily obtain a relevant prior distribution, it is common to standardise the observed data. The majority of the available BO implementations start by evaluating a small set of randomly selected configurations, and use the collected information to initialise the GP and to standardise future data. In this way, the GP will reasonably predict that, by picking a random configuration, we will likely obtain a performance value equal to the average value that we obtained by evaluating some randomly-selected configurations.

When taking into consideration different workloads, however, it becomes crucial to standardise each point by taking into account the relevant workload, and not just the initial one observed during the initialisation phase. To normalise performance values, we use a modified version of the Normalized Performance Improvement (NPI) [4]:

$$\tilde{NPI}(\vec{x}, \vec{w}) = \frac{f(\vec{x}_0, \vec{w}) - f(\vec{x}, \vec{w})}{f(\vec{x}_0, \vec{w}) - f(\vec{x}^+_{\vec{w}}, \vec{w})} \quad (3)$$

where $\vec{x}$ is the configuration we are evaluating, $\vec{x}_0$ is the vendor default configuration, and $\vec{w}$ is the workload on which we are evaluating $\vec{x}$. $f(\vec{x}, \vec{w})$ is the performance measurement obtained by the configuration $\vec{x}$ under the workload $\vec{w}$ and $\vec{x}^+_{\vec{w}_i}$ is the best configuration we have found so far during the tuning of workload $\vec{w}$. Clearly, $\vec{x}^+_{\vec{w}}$ changes as we go on with the optimisation and discover better configurations. For this reason, we need to re-normalise past values at each iteration. Therefore, $\tilde{NPI}(\vec{x}, \vec{w})$ measures the optimality of the configuration $\vec{x}$ for the workload $\vec{w}$. A $\tilde{NPI}$ of 0 means that, under workload $\vec{w}$, the configuration $\vec{x}$ is performing exactly like the baseline, whereas a $\tilde{NPI}$ of 1 means that configuration $\vec{x}$ is the best one that we have found so far for workload $\vec{w}$.

As we have assumed to have an external workload characterisation module, we also assume that it can cluster the workloads and provide us with the performance value of the baseline configuration for any given workload. Notice that this is easily obtained when a historical knowledge base is available (such as in [27]). On the other hand, when we do not have any previous information, we can collect this information by evaluating a handful of configurations. In some deployments, we could obtain this measurement using a control group (i.e., a deployment running with the baseline configuration and receiving a replica of the actual workload).

In conclusion, we report in Algorithm 1 the pseudocode of CGPTuner, the algorithm that we propose for the automatic tuning of DBMS configurations. Notice that we do not provide a stopping criterion, as we imagine the tuning process to keep optimising the system in an online way. In this way, the tuner takes care of adapting the configuration to the incoming workload. Since CGPTuner is a BO technique, it will converge towards optimum configurations once they are found. The strong theoretical guarantees of BO allow the tuning process to go on indefinitely, adapting configurations to new workloads and quickly converging toward optimal solutions by correctly balancing exploration and exploitation [16, 21, 26]. Nonetheless, adding a stopping criterion is trivial, such as limiting the number of iterations or stopping when no improvement is observed for a certain number of iterations.

4 EXPERIMENTAL EVALUATION

In this section, we describe the experiments we conduct to evaluate the proposed approach. We use the MongoDB 4.0.3 and Cassandra 3.11.4 DBMSs. We run the experiments on Amazon EC2[4] using two instances: the first one running YCSB 0.15.0 as a load generator and deployed on a c5.large EC2 instance with 2 vCPUs and 4GB RAM, the second one running a DBMS and deployed on an i3.xlarge instance with 4 vCPUs, 30GB RAM and an NVMe SSD storage.

We select a set of interesting parameters to tune, including different layers of the IT stack. We use 15 parameters for MongoDB and 24 for Cassandra, reported in Tables 1 and 2. We manually selected the parameters, trying to collect data from different components of the stack. We started from a bigger set of parameters and then kept only those for which we observed some variation in the DBMS response time after an initial sampling over the parameters, hence the different OS parameters for the DBMSs. When using CGPTuner in a real setting, one can automatically select the tunable parameters using available methodologies [11].

For both Cassandra and MongoDB we select three workloads from the YCSB default ones: (a) update heavy with a 50/50 mix in read and write operations, (b) read mostly with a 95/5 reads/write mix and (c) read-only. All the workloads use a Zipfian distribution. We also vary the number of YCSB threads from 10 to 90. We

[4]https://aws.amazon.com/ec2/

Algorithm 1: CGPTuner.

```
input: A number $n$ of starting random configurations
for $i \leftarrow 0$ to $n$ do
    measure current workload $\vec{w}_i$;
    select a random configuration $\vec{x}_i$;
    apply $\vec{x}_i$ to the system under test;
    measure performance score $y_i$;
    store $\vec{x}_i, \vec{w}_i, y_i$ in the KB;
end
while True do
    get previous data from KB: $D_i = \{(\vec{x}_j, \vec{w}_j, y_j)\}_{j=0}^{i-1}$;
    foreach oberseved workload $\vec{w} \in KB$ do
        $\vec{x}^+_{\vec{w}} = \arg\max y_j, (\vec{x}_j, \vec{w}, y_j) \in KB$ ;
    end
    normalize previous data: $\tilde{NPI}_j = \frac{f(\vec{x}_0, \vec{w}_j) - f(\vec{x}_j, \vec{w}_j)}{f(\vec{x}_0, \vec{w}_j) - f(\vec{x}^+_{\vec{w}_j}, \vec{w}_j)}$;
    update the CGP with $\tilde{D}_i = \{\vec{x}_j, \vec{w}_j, \tilde{NPI}_j\}$;
    measure current workload $\vec{w}_i$;
    optimise acquisition function: $\vec{x}_i = \max_x a(\vec{x}, \vec{w}_i)$;
    apply $\vec{x}_i$ to the system under test;
    measure performance score $y_i$;
    store $\vec{x}_i, \vec{w}_i, y_i$ in the KB;
    $i++$;
end
```

Table 1: MongoDB parameters.

Layer	Parameter
MongoDB	`wiredTigerCacheSizeGB`
MongoDB	`eviction_dirty_target`
MongoDB	`eviction_dirty_trigger`
MongoDB	`syncdelay`
OS	`sched_latency_ns`
OS	`sched_migration_cost_ns`
OS	`vm.dirty_background_ratio`
OS	`vm.dirty_ratio`
OS	`vm.min_free_kbytes`
OS	`vm.vfs_cache_pressure`
OS	Network `RFS`
OS	Storage `noatime`
OS	Storage `nr_requests`
OS	Storage `scheduler`
OS	Storage `read_ahead_kb`

use YCSB to create 30 000 000 records, roughly obtaining a 30GB database.

We select a common efficiency problem as tuning goal: minimise memory consumption M with a soft constraint on response time R. Since CGP, and all the considered tuners, are suited to optimise a scalar value, we use a common trick in constrained optimisation and treat the constraint with a penalty term [22].

For both the DBMSs, we thus minimise the following function:

$$M[\text{MiB}] + \sigma \cdot R[\text{ms}] \tag{4}$$

Table 2: Cassandra parameters.

Layer	Parameter
Cassandra	`commitlog_compression`
Cassandra	`commitlog_segment_size_in_mb`
Cassandra	`commitlog_sync_period_in_ms`
Cassandra	`Compaction Strategy`
Cassandra	`compaction_throughput_mb_per_sec`
Cassandra	`concurrent_compactors`
Cassandra	`concurrent_reads`
Cassandra	`concurrent_writes`
Cassandra	`file_cache_size_in_mb`
Cassandra	`memtable_cleanup_threshold`
JVM	`CMSInitiatingOccupancyFraction`
JVM	`ConcGCThreads`
JVM	GC Type
JVM	`Xmx` (max heap size)
JVM	`MaxTenuringThreshold`
JVM	`NewRatio`
JVM	`ParallelGCThreads`
JVM	`SurvivorRatio`
OS	`CPUSchedNrMigrate`
OS	`MemoryTransparentHugepageEnabled`
OS	`MemoryVmDirtyExpire`
OS	`NetworkNetIpv4TcpMaxSynBacklog`
OS	Storage `scheduler`
OS	Storage `read_ahead_kb`

where the term σ is the penalty coefficient. The choice of σ depends both on the employed measurement units and on the importance of the constraint. As we measure R in ms and M in MiB, we run most of the experiments using $\sigma = 10^3$, essentially saying that each additional millisecond costs as much as 1 GiB of memory. We find this function to be reasonably balanced, and the resulting optimisation problem is a difficult one, where the suggested configurations must be adapted to the different workloads. For completeness, we also run the experiments using $\sigma = 10^1, \sigma = 10^4$ and $\sigma = 10^5$, where each millisecond cost 10 MiB, 10 GiB or 100 GiB respectively. The resulting problems are much more unbalanced toward decreasing either the memory or the response time.

We measure the average response time in milliseconds, as reported by YCSB. For MongoDB, we use the `wiredTigerCacheSizeGB` parameter to measure M, multiplied by 1024 so to measure it in MiB. On Cassandra, we use the sum of `file_cache_size_in_mb` and JVM max heap size (`Xmx`), both measured in MiB.

Notice that the memory parameter is one of the tunable parameters, but the considered tuners have no access to the metric formula and, thus, they need to discover this link. Whichever goal function we select (even a much simpler minimise R) we would still be using a function of the applied parameters, which, however, would be unknown to both us and the tuners. Conversely, by using the actual memory parameters we can implement a sanity check on the tuning results by checking their values.

We compare the performance of our CGPTuner against Random, BestConfig [30], OpenTuner [3] and GP. GP is a Bayesian Optimiser based on Gaussian Processes which is workload agnostic, and thus is similar to OtterTune [27] without a knowledge base.

Table 3: Dimensions of datasets and number of collected samples. Each sample represents a 45 minutes experiment. The workload parameters represent the YCSB workload mix and the number of run threads.

Dataset	Tunable params	Workload params	Samples
MongoDB	15	2	4219
Cassandra	24	2	3728

4.1 Data Collection and DBMS Models

We start by running an extensive set of experiments exploring the configuration spaces of the DBMSs and collecting a wide variety of performance metrics. We then use the collected data to build a random forest regressor to predict the performance of all the possible configurations in the search space.

By doing this, we build a model of the IT system performance, which is then used as the target of the tuning process for the algorithm evaluation. In this way, we make sure that our experiments are easily reproducible. Moreover, we can be sure that different tuning algorithms work precisely on the same system and exclude the noise from the evaluation. Furthermore, we can evaluate the variability of the tuning algorithms by repeating multiple times the tuning process without having to run the experiments.

Notice that this step is not a requirement of the tuning algorithm, but instead is just an additional step that we perform to conduct a reproducible evaluation.

Before testing a configuration, we restart the DBMS and restore the database to its original version so to avoid any cross-contamination between the experiments of different configurations. We let the experiment run for 45 minutes, discard the initial 15 minutes and the last minute and then compute the average throughput and response time across the measurement period.

We select the test configurations using Sobol sequences [25], which are quasi-random sequences with a low discrepancy and fill the search domain quickly and evenly.

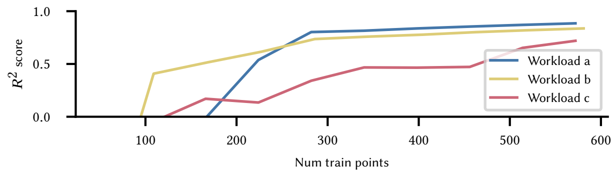
We report in Table 3 the number of parameters and collected points for all the datasets.

4.2 Models Accuracy

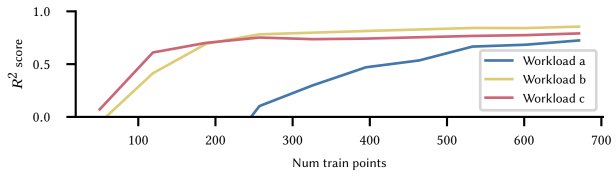
The DBMS models are useful for the evaluation of the tuning algorithms. However, if they were very different from the real systems, they would lead us to incorrect results.

To evaluate how close the prediction models are to the real system we use a simple holdout validation approach and split the measurements into two sets: the first one contains 25% of the collected data and is used as a test set, the second one contains the remaining data and is used as a train set. Then, we consider progressively bigger subsets of the training set and use them for training several regressors, which we evaluate on the test set.

In this way, we can see how the accuracy of the regressors evolves as more data are considered. The results are in Figure 4, and we can see that the regressors reach convergence. This indicates that the amount of data we have is enough to create a good DBMS model. The remaining variability, which the models cannot explain, is probably due to the noise of the measurement, since adding more training points does not help to increase the score of the models. Without using these regressor models, we would have no way of discarding the measurement noise, which would severely affect the algorithm evaluation.



(a) Cassandra



(b) MongoDB

Figure 4: Evolution of the R^2 score of the regressors as more training points are considered.

4.3 Evaluation Metrics

We now define the metrics that we use to compare different tuners. Without considering technological constraints, we have two ways to run an autotuner. We can either let it work directly on the production system (*online*) or replicate the system in a testing environment (*offline*). If we run the autotuner offline, we want it to explore the search space as quickly as possible, finding good configurations that we can then move in the production environment. Conversely, in the online setup, we want the tuner to find good configurations and avoid bad ones: the trade-off between exploration and exploitation is subtler. We use two metrics to measure the tuner quality in the two setups: Cumulative Reward for online tuning and Iterative Best for offline tuning.

Using DBMS models gives us a significant advantage: for each workload condition we know the optimal configuration and thus we can see how close the tuners are to the real optimum. We use this information to derive the Normalized Performance Improvement (NPI) metric[4, 7]. The NPI at tuning iteration i is defined as:

$$NPI(i) = \frac{\text{achieved PI}}{\text{potential PI}} = \frac{y_0 - y_i}{y_0 - y^*} = \frac{f(\vec{x}_0, \vec{w}_i) - f(\vec{x}_i, \vec{w}_i)}{f(\vec{x}_0, \vec{w}_i) - f(\vec{x}^*_{\vec{w}_i}, \vec{w}_i)} \quad (5)$$

where $\vec{x}_0$ is the vendor default configuration, $\vec{x}_i$ is the configuration that we are evaluating at iteration i and $\vec{x}^*_{\vec{w}_i}$ is the optimal configuration for the workload observed at iteration i, which we know from the collected dataset. This metric measures the ratio of the achieved performance improvement over the potential performance improvement. For any workload, an NPI of 0 means that we have the same performance obtained by the vendor default configuration, while a unitary NPI means that we have found the global optimum. Differently from the $\tilde{NPI}$ metric that we used in CGPTuner, here we are using the true optimal configuration (according to the collected dataset) instead of the best one observed during the tuning. This metric can thus be computed only to evaluate tuners, and not during the tuning as it requires to know all the $\vec{x}^*$, which are not available at tuning time.

Starting from NPI, we define the two metrics that we use to compare tuners: the Cumulative Reward and the Iterative Best. The Cumulative Reward (CR) is a standard metric in Reinforcement Learning and is simply defined as the sum of the obtained NPI scores:

$$CR(i) = \sum_{j=0}^{i} NPI(j) \tag{6}$$

As the optimal configuration has an NPI of 1, a perfect tuner has a CR with a constant unitary slope. On the other hand, keeping the baseline configuration leads to a zero reward, while testing bad configurations gives a negative (unbounded) reward. This metric is thus a useful indicator of tuner ability to understand the problem and adapt to workload variations, trading off exploration with exploitation.

The Iterative Best (IB) is defined as:

$$IB(i) = \max_{j: j \le i, \vec{w}_j = \vec{w}_i} NPI(j) \tag{7}$$

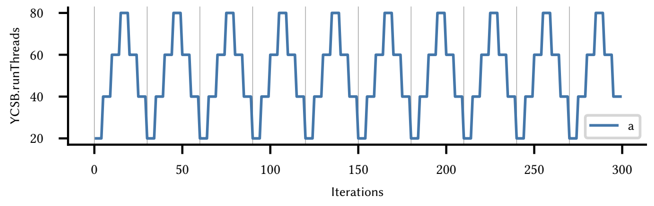
In simpler terms, we keep a separate counter for each possible workload and, at each iteration, compute an iterated maximum on the appropriate counter. The IB thus measures the ability of a tuner to quickly explore the search space and find good configurations.

We repeat the tuning 16 times and then compute NPI, CR and IB for each repetition. Then, we take the median over the 16 repetitions to produce the plots.

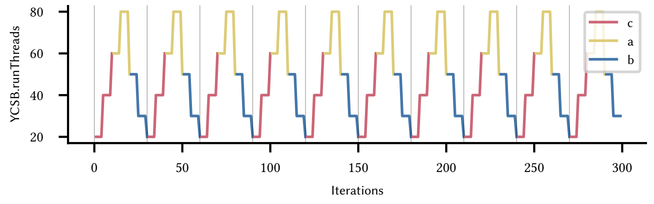
As we normalize all the performance metrics in terms of NPI, they will be more challenging to interpret for the performance expert. However, we use them to quantitatively compare the tuners' efficacy from an optimisation viewpoint and, thus, we must take into account the different workloads. If we used the non-normalized performance measurement, we could conclude that a tuner is performing well when, in reality, we just tested an easily tunable workload. Moreover, comparing the non-normalized performance of the best configurations found would not take into account two other important aspects: the number of tuning iterations required to find the best configurations, and how many bad configurations the tuner has tested. As an example, even a random tuner finds the optimal configuration if given infinite time. The IB and CR metrics, instead, takes these aspects into consideration. Some non-normalized results, which are useful to assess the quality of the tuners from a performance perspective, are available in Section 4.7.

4.4 Workload Patterns

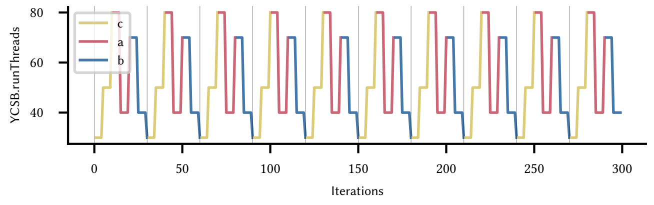
We select three workload patterns for the evaluation of MongoDB and Cassandra. We report the patterns in Figure 5, using the colours to indicate the read/write mix and the y-axes for the number of YCSB threads. The first pattern (Figure 5a) is the simplest one and is used only for the tuning of the hyperparameters, we vary the number of connected threads, simulating a gradual ramp in the load. In the second one (Figure 5b), named *Ramp*, we gradually vary both the number of connected threads and the read/write mix. In



(a) Intensity pattern: fixed mix, variable intensity.



(b) Ramp pattern: variable mix and intensity, one peak.



(c) Peaks pattern: variable mix and intensity, two peaks.

Figure 5: Workload patterns. Vertical lines represent pattern repetitions and corresponds to an entire day of tuning.

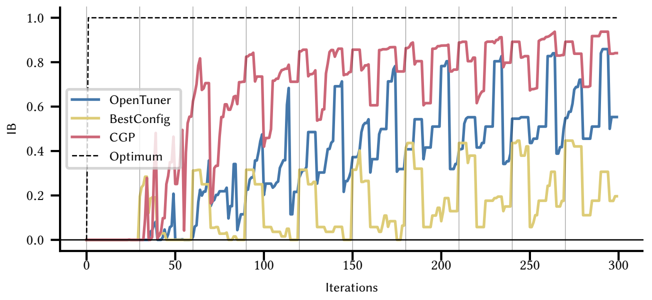
the third one (Figure 5c), named *Peaks*, we try to mimic a typical day-based workload, with the majority of the load concentrated in working hours and a decrease during lunchtime. The three patterns are repeated identically over time, but none of the tuners can exploit this, what matters is just the current workload. Each repetition of the pattern lasts for 30 iterations, and, since each of our measurement takes 45 minutes, each repetition roughly represents a day.

4.5 Selection of Hyperparameters

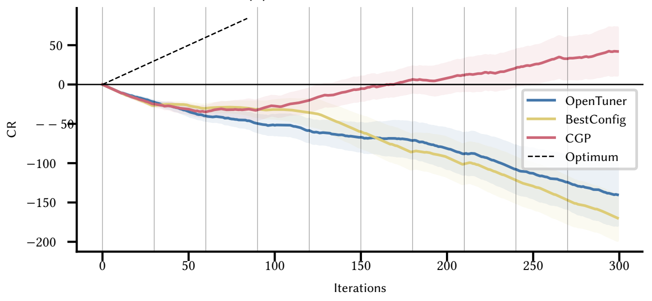
All the considered tuners have some hyperparameters to select: in OpenTuner we have several tuning techniques that we can use, in BestConfig we need to decide the number of tuning rounds (i.e., partial restarts of the search algorithm) and for CGPTuner we need to select whether the workload and configuration kernels should be combined with a sum or a multiplication. We use a cross-validation approach: we select a simple workload pattern (Figure 5a) on Cassandra and compute the final median cumulated reward of all the considered tuners. This validation pattern is then excluded from other evaluations, and it is very different from the patterns that we use to compare the tuners.

For OpenTuner we select the "AUCBanditMetaTechniqueA" which, according to OpenTuner description, is a Meta Technique composed by:

- a Differential Evolution technique with a crossover rate of 20%

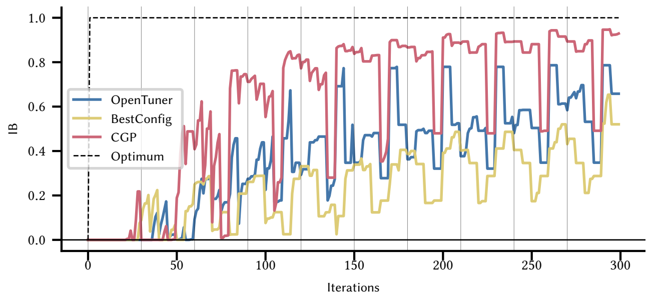


(a) Iterative Best

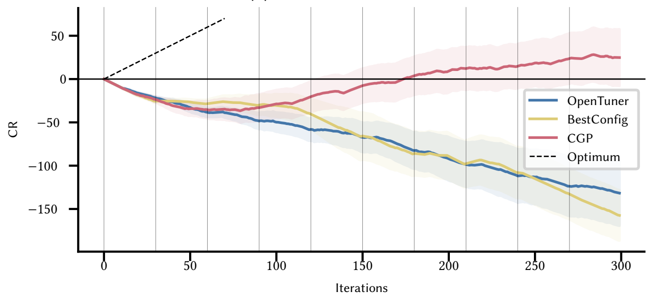


(b) Cumulative reward

Figure 6: Results on Cassandra - Ramp pattern.



(a) Iterative Best



(b) Cumulative reward

Figure 7: Results on Cassandra - Peaks pattern.

- a Uniform Greedy Mutation technique
- a Normal Greedy Mutation technique with a mutation rate of 30%
- a Random Nelder-Mead technique.

OpenTuner uses a multi-armed bandit to decide which technique to use at each iteration.

For BestConfig, we use a number of rounds equal to the number of times that we repeat the entire workload pattern. As we imagine that the entire workload pattern captures an entire day, we are basically starting a new BestConfig round each day.

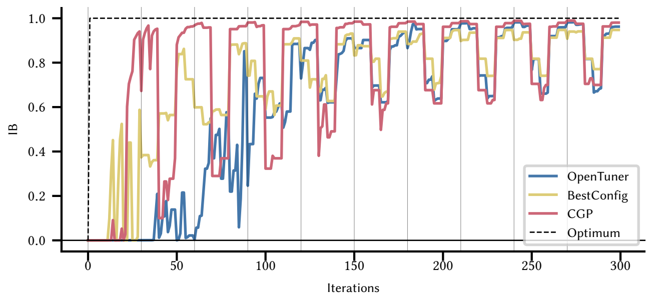
Finally, for CGPTuner we decide to combine the kernels with a multiplication, thus considering two points similar when both the configuration and the workload are similar.

4.6 Experimental Results

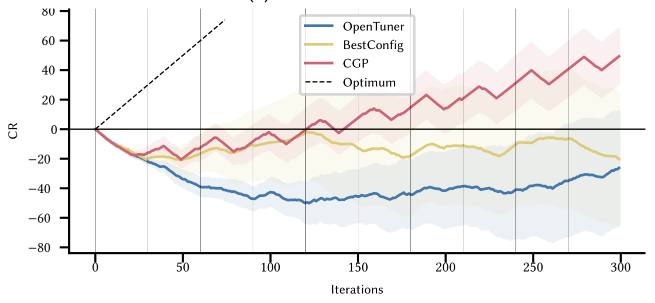
The results of the experimental evaluation are reported in detail in Figures 6 to 9, whereas Tables 4 and 5 give a summary. Since random and GP obtain very bad results, we exclude them from the figures and report them only in the summary tables.

In all the figures, the x-axis represents tuning iterations, and the y-axes represent either the CR or the IB metric. We mark the start of each repetition of the workload pattern with a vertical line. Each vertical line can thus be seen as the start of a new tuning day. The different colours represent different tuners. As we repeat each tuning 16 times, the lines represent the median result, whereas the shaded areas represent the standard deviation.

We start by looking at the results of Cassandra's tuning on the Ramp pattern, reported in Figure 6. Looking at the IB, we see that CGP finds, for all the workloads, configurations which are better than the baseline. Thus, if we use CGP in an offline way, we start



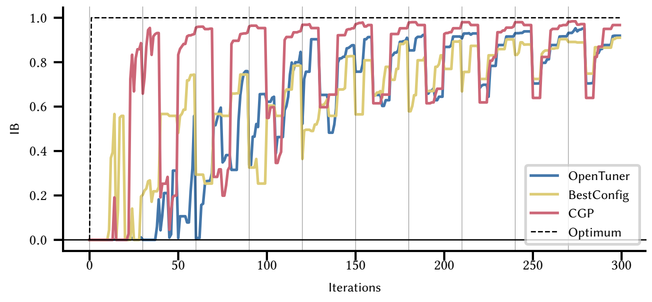
(a) Iterative Best



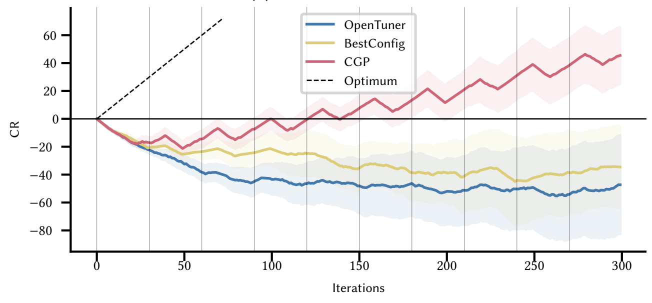
(b) Cumulative reward

Figure 8: Results on MongoDB - Ramp pattern.

to see the benefits already at the second day of tuning. Comparing CGP with the other tuners at the end of the tuning period, we see that CGP has found substantially better configurations on all the workloads. By looking at the CR, we first observe that OpenTuner



(a) Iterative Best



(b) Cumulative reward

Figure 9: Results on MongoDB - Peaks pattern.

and BestConfig remain negative, and get worse over time, meaning that they cannot be used in an online way, as keeping the baseline would lead to better overall results. Conversely, CGP uses the first two days to explore the space (negative slope), on the third it starts to compensate for the exploration (positive slope), and on the sixth day we have the break-even point (positive CR): from this point on, we are gaining from using CGP and have recovered from the cost of exploration.

The results on Cassandra Peaks, reported in Figure 7, are substantially similar: CGP reaches higher IB scores sooner and the CR break-even is located at the end of the sixth day.

The tuning of MongoDB (Figures 8 and 9) is simpler, as testified by the CR break-even reached on the fifth day and by the generally higher IB scores, albeit CGP finds them sooner than the competitors. Interestingly, all the tuners have difficulties in tuning the central section of the workload pattern, which corresponds to workload *a*. All the tuners are able to find good configurations (positive IB), but they are all still far from the optimum and fail to converge.

In Tables 4 and 5 we report the results in terms of the average IB and NPI score. In offline tuning we want the IB curve to reach 1 as soon as possible, we thus measure the average IB score which can be seen as the area under the IB curve normalized by the number of iterations:

$$\bar{IB}(i) = \frac{\sum_{j=0}^{i} IB(j)}{i}, i = 300 \quad (8)$$

In online tuning we want the NPI curve to have a costant unitary slope, we thus measure the average NPI score which can be seen as the overall slope of the CR curve:

$$\bar{NPI}(i) = \frac{\sum_{j=0}^{i} NPI(j)}{i} = \frac{CR(i)}{i}, i = 300 \quad (9)$$

For both the metrics, a perfect tuner would receive a score of 1, and thus they can be seen as offline/online optimalities.

We report the results for different values of σ, which is the response time penalty coefficient as per Equation (4). For the results in Figures 6 to 9 we used $\sigma = 10^3$.

In the majority of the considered scenarios, CGP is the best tuner. Only GP and BestConfig reach similar average IB scores in the simplest scenarios. As we anticipated, in fact, $\sigma = 10^3$ produces the most difficult problem, as testified by the lower CR reached by all the tuners. Finally, notice that even Random has a positive average IB score on most of the scenarios, confirming that, if we can afford to run many iterations and testing bad configurations is not a problem, a simple Random approach finds interesting configurations.

4.7 Suggested Configurations

So far we have compared the algorithms from an optimisation perspective, checking how good they are at optimising a black-box function. That function, however, should represent the performance of a DBMS. To give a useful insight, we report in Figure 10 the non-normalised values of the throughput and memory consumption used as a score for the optimisation. More specifically, we consider the Cassandra Ramp scenario and report the median value obtained by each tuner. We take the median over the repetitions in order to take into account the tuner variability. We do not use the value of the best configurations found as they would be misleading: with a proper number of iterations even a Random tuner can find good configurations (as testified by Table 4) and taking a single configuration over 50 tested ones would produce very noisy results.

After the two first tuning days CGP substantially stops exploring: it settles on a low value for the Cassandra cache and varies the JVM heap's size depending on the incoming workload, increasing it only when necessary. Both the parameters are very similar to the best configuration available in the dataset. Both OpenTuner and BestConfig are not converging, and, in general, they are exploring higher values for both the memory parameters. As for the throughput, CGP achieves better results on most of the workloads. Notice that, under some workloads, the throughput of CGPTuner is higher than the best one. Under the same workloads, however, CGP selects more memory than necessary. According to the selected goal function, the increase in throughput is not enough to justify the increased memory consumption. The close match between CGP and Best confirms the high IB score in Figure 6a, whereas the nearly-random behaviour of BestConfig finds a match in its decreasing reward of Figure 6b.

In Figure 12 we show the maximum throughput achieved by the various tuners in Cassandra Ramp $\sigma = 10^5$, where reducing the response time is most important. CGP finds a higher throughput in almost all the considered workloads.

In Figure 11 we report the best settings for the three most relevant parameters after the memory ones. We select the parameters according to the feature importance computed by the Random Forest (RF) used in Section 4.1 (excluding the parameters used in the score). As the RF is used to predict the response time, these parameters are the ones which affect response time the most. The best configuration found largely varies with the workload, confirming

Table 4: Offline tuner optimality with average IB score (i.e., area under IB curve normalized w.r.t. an optimal tuner, optimal value is 1). σ is the response time penalty coefficient. Best tuners per scenario in bold using Welch's t-test with 1% p-value over 16 repetitions.

		Average IB score				
$\sigma[\frac{\text{MiB}}{\text{ms}}]$	Scenario	Random	GP	OpenTuner	BestConfig	CGP
10^1	Cassandra Ramp	0.00 ± 0.02	0.45 ± 0.20	0.72 ± 0.18	0.70 ± 0.20	$\mathbf{0.89 \pm 0.03}$
	Cassandra Peaks	0.01 ± 0.03	0.33 ± 0.15	0.68 ± 0.13	0.72 ± 0.16	$\mathbf{0.81 \pm 0.09}$
	MongoDB Ramp	0.33 ± 0.10	$\mathbf{0.95 \pm 0.08}$	0.78 ± 0.14	$\mathbf{0.92 \pm 0.01}$	$\mathbf{0.94 \pm 0.05}$
	MongoDB Peaks	0.31 ± 0.06	$\mathbf{0.93 \pm 0.06}$	0.77 ± 0.06	$\mathbf{0.92 \pm 0.02}$	$\mathbf{0.94 \pm 0.02}$
10^3	Cassandra Ramp	0.02 ± 0.03	0.14 ± 0.09	0.36 ± 0.12	0.19 ± 0.08	$\mathbf{0.63 \pm 0.04}$
	Cassandra Peaks	0.04 ± 0.04	0.15 ± 0.09	0.36 ± 0.12	0.24 ± 0.08	$\mathbf{0.57 \pm 0.08}$
	MongoDB Ramp	0.30 ± 0.07	0.30 ± 0.17	0.58 ± 0.10	$\mathbf{0.69 \pm 0.10}$	$\mathbf{0.73 \pm 0.05}$
	MongoDB Peaks	0.27 ± 0.07	0.44 ± 0.19	0.60 ± 0.07	0.60 ± 0.08	$\mathbf{0.75 \pm 0.04}$
10^4	Cassandra Ramp	0.42 ± 0.03	0.41 ± 0.06	0.60 ± 0.05	0.53 ± 0.03	$\mathbf{0.66 \pm 0.03}$
	Cassandra Peaks	0.44 ± 0.04	0.41 ± 0.04	0.59 ± 0.05	0.53 ± 0.03	$\mathbf{0.65 \pm 0.04}$
	MongoDB Ramp	0.51 ± 0.04	0.57 ± 0.07	0.63 ± 0.06	0.64 ± 0.05	$\mathbf{0.71 \pm 0.05}$
	MongoDB Peaks	0.59 ± 0.04	0.63 ± 0.07	0.65 ± 0.06	0.66 ± 0.05	$\mathbf{0.75 \pm 0.03}$
10^5	Cassandra Ramp	0.54 ± 0.02	0.52 ± 0.03	0.63 ± 0.05	0.58 ± 0.03	$\mathbf{0.70 \pm 0.03}$
	Cassandra Peaks	0.54 ± 0.03	0.53 ± 0.03	0.65 ± 0.05	0.58 ± 0.03	$\mathbf{0.68 \pm 0.03}$
	MongoDB Ramp	0.82 ± 0.02	0.82 ± 0.02	0.82 ± 0.03	0.75 ± 0.04	$\mathbf{0.86 \pm 0.02}$
	MongoDB Peaks	0.83 ± 0.01	0.83 ± 0.02	0.83 ± 0.02	0.77 ± 0.04	$\mathbf{0.88 \pm 0.02}$

Table 5: Online tuner optimality with average NPI score (i.e., slope of CR curve, optimal value is 1). σ is the response time penalty coefficient. Best tuners per scenario in bold using Welch's t-test with 1% p-value over 16 repetitions.

		Average NPI score				
$\sigma[\frac{\text{MiB}}{\text{ms}}]$	Scenario	Random	GP	OpenTuner	BestConfig	CGP
10^1	Cassandra Ramp	-0.99 ± 0.00	-0.90 ± 0.05	-0.01 ± 0.12	0.01 ± 0.17	$\mathbf{0.70 \pm 0.07}$
	Cassandra Peaks	-0.99 ± 0.01	-0.94 ± 0.03	0.01 ± 0.09	-0.38 ± 0.16	$\mathbf{0.47 \pm 0.15}$
	MongoDB Ramp	-0.92 ± 0.02	0.13 ± 0.37	0.18 ± 0.13	0.07 ± 0.21	$\mathbf{0.74 \pm 0.11}$
	MongoDB Peaks	-0.92 ± 0.01	-0.14 ± 0.26	0.21 ± 0.10	-0.04 ± 0.18	$\mathbf{0.76 \pm 0.04}$
10^3	Cassandra Ramp	-0.98 ± 0.01	-0.96 ± 0.02	-0.47 ± 0.13	-0.57 ± 0.10	$\mathbf{0.14 \pm 0.10}$
	Cassandra Peaks	-0.98 ± 0.01	-0.95 ± 0.02	-0.44 ± 0.13	-0.52 ± 0.10	$\mathbf{0.08 \pm 0.11}$
	MongoDB Ramp	-0.91 ± 0.02	-0.91 ± 0.06	-0.09 ± 0.13	-0.07 ± 0.15	$\mathbf{0.16 \pm 0.06}$
	MongoDB Peaks	-0.91 ± 0.02	-0.87 ± 0.13	-0.16 ± 0.12	-0.12 ± 0.10	$\mathbf{0.15 \pm 0.07}$
10^4	Cassandra Ramp	-0.69 ± 0.03	-0.73 ± 0.05	-0.25 ± 0.15	-0.28 ± 0.11	$\mathbf{0.32 \pm 0.04}$
	Cassandra Peaks	-0.68 ± 0.03	-0.73 ± 0.03	-0.20 ± 0.24	-0.27 ± 0.07	$\mathbf{0.34 \pm 0.04}$
	MongoDB Ramp	-0.43 ± 0.03	-0.37 ± 0.05	0.01 ± 0.08	0.04 ± 0.08	$\mathbf{0.35 \pm 0.07}$
	MongoDB Peaks	-0.42 ± 0.02	-0.33 ± 0.07	0.05 ± 0.09	0.10 ± 0.10	$\mathbf{0.34 \pm 0.05}$
10^5	Cassandra Ramp	-0.53 ± 0.04	-0.57 ± 0.07	0.23 ± 0.25	0.07 ± 0.13	$\mathbf{0.48 \pm 0.06}$
	Cassandra Peaks	-0.52 ± 0.04	-0.58 ± 0.05	0.24 ± 0.20	0.18 ± 0.11	$\mathbf{0.44 \pm 0.03}$
	MongoDB Ramp	0.46 ± 0.02	0.46 ± 0.05	0.58 ± 0.03	0.58 ± 0.09	$\mathbf{0.63 \pm 0.02}$
	MongoDB Peaks	0.49 ± 0.02	0.47 ± 0.04	0.61 ± 0.03	0.61 ± 0.04	$\mathbf{0.66 \pm 0.02}$

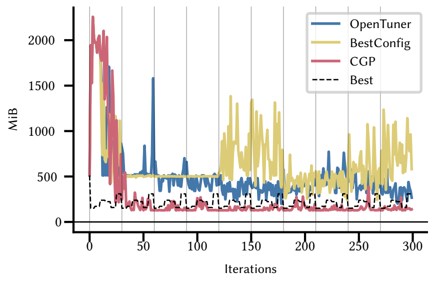
our initial supposition that the configuration should vary with the incoming workload.

Interestingly, the two most important parameters come from the OS layer, and thus can be changed dynamically, without having to restart the DBMS. Considering the non-portability of the best configuration, we should prefer using software that allows to dynamically change its configuration, so to continuously adapt it to the incoming workload.

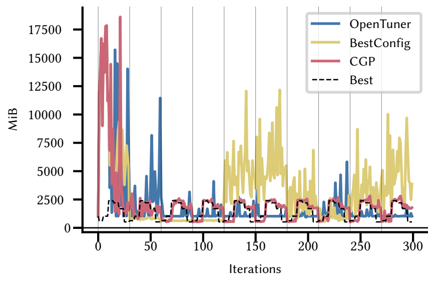
If looking for a single configuration that works over all the observed workloads, one should focus on the GP tuner, which has no access to the workload dimensions and thus is forced to look for a configuration with good overall performance. From the results of Table 4, we see that such an approach still outperforms the baseline configuration.

4.8 CGP complexity

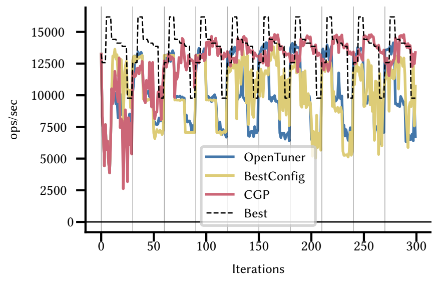
In Figure 13 we report the time (in seconds) required by CGP at each iteration to suggest the next configuration, when run on a laptop equipped with an Intel i5-8250U CPU.



(a) Cassandra `file_cache_size_in_mb`

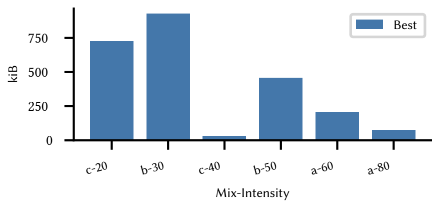


(b) JVM max heap size (`Xmx`)

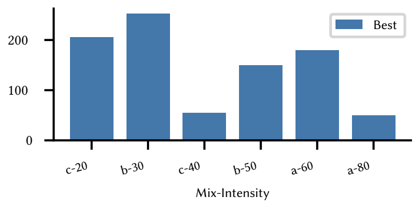


(c) Throughput

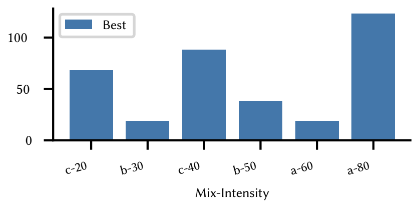
Figure 10: Unnormalised Memory and Throughput for the Cassandra Ramp tuning, Best refers to the best configuration available in the collected dataset according to the goal function.



(a) OS Storage `read_ahead_kb`



(b) OS `CPUSchedNrMigrate`



(c) Cassandra `concurrent_reads`

Figure 11: Value of most relevant parameters in the best configuration found in the dataset for the workloads of Cassandra Ramp. Optimal settings are not portable across workloads.

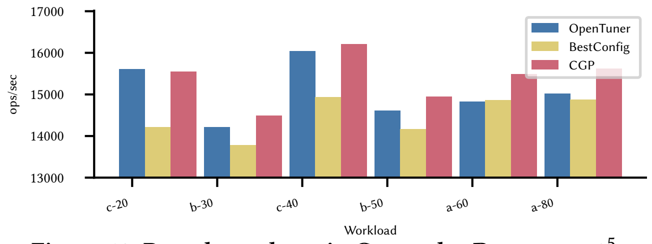


Figure 12: Best throughput in Cassandra Ramp $\sigma = 10^5$.

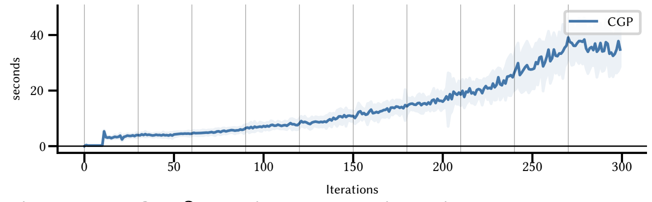


Figure 13: Configuration suggestion time on MongoDB Peaks.

From a theoretical perspective, since CGP uses Gaussian Processes, the time complexity scales cubically with the number of iterations and quadratically with the number of parameters [29].

Form a practical viewpoint, our very inefficient, single core implementation takes, at most, less than a minute to find a configuration that is evaluated over the next hour, resulting in an acceptable overhead given the superior performance reached by CGP.

5 CONCLUSION

In this work, we have presented CGPTuner, a contextual gaussian process bayesian optimiser which successfully tunes the configuration of an IT system while adapting to the incoming workload and without relying on a previously collected knowledge base. We have evaluated CGPTuner on two DBMSs, demonstrating the portability across different database systems. We have evaluated sixteen tuning scenarios, showing that the proposed algorithm quickly understands the configuration space and beats the vendor default configuration. CGPTuner was able to consistently pick well-performing configurations on every workload already after 30 iterations, which is a substantially smaller amount of information than the massive knowledge bases usually employed in the state of the art. CGPTuner adapted the suggested configuration to the observed workload, trading off an increase in the memory consumption only when needed to reduce the response time, and decreasing again the memory when the load decreased.

CGPTuner proved to be a viable solution both to the offline and online tuning problems. As future extensions, we plan to improve CGPTuner to work with more tuning iterations and bigger search spaces. The number of iterations poses a limit on the computational complexity, which scales cubically with the number of iterations. However, several techniques already exists to reduce the temporal complexity of Gaussian Processes, and their application to CGPTuner would be trivial [20, 21, 23, 24]. As for the number of tunable parameters, some approaches are already available in the Bayesian Optimisation literature [21, 28]. However, the main challenge here is to conduct reproducible evaluations as collecting a dataset becomes more and more complex as we introduce more parameters.

REFERENCES

[1] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. 2006. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 11–pp.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 469–482.

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

[4] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 2 (2016), 1–25.

[5] Pooja B Bindal, Devesh Singhal, AV Subramanyam, Vivek Kumar, et al. 2020. OneStopTuner: An End to End Architecture for JVM Tuning of Spark Applications. *arXiv preprint arXiv:2009.06374* (2020).

[6] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. 2007. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 185–197.

[7] Stefano Cereda, Gianluca Palermo, Paolo Cremonesi, and Stefano Doni. 2020. A Collaborative Filtering Approach for the Automatic Tuning of Compiler Optimisations. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 15–25.

[8] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. 2012. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 3 (2012), 1–30.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. 2017. BOAT: Building auto-tuners with structured Bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*. 479–488.

[11] Biplob K Debnath, David J Lilja, and Mohamed F Mokbel. 2008. SARD: A statistical approach for ranking database tuning parameters. In *2008 IEEE 24th International Conference on Data Engineering Workshop*. IEEE, 11–18.

[12] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with iTuned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[13] Matthew D Hoffman, Eric Brochu, and Nando de Freitas. 2011. Portfolio Allocation for Bayesian Optimization.. In *UAI*. Citeseer, 327–336.

[14] Chin-Jung Hsu, Vivek Nair, Vincent W Freeh, and Tim Menzies. 2018. Arrow: Low-level augmented bayesian optimization for finding the best cloud vm. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 660–670.

[15] Toru Kisuki, Peter MW Knijnenburg, and Michael FP O'Boyle. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*. IEEE, 237–246.

[16] Andreas Krause and Cheng S Ong. 2011. Contextual gaussian process bandit optimization. In *Advances in neural information processing systems*. 2447–2455.

[17] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[18] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.

[19] Carl Edward Rasmussen. 2003. Gaussian processes in machine learning. In *Summer School on Machine Learning*. Springer, 63–71.

[20] Matthias Seeger, Christopher Williams, and Neil Lawrence. 2003. *Fast forward selection to speed up sparse Gaussian process regression*. Technical Report.

[21] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.

[22] Alice E Smith and David W Coit. 1997. Penalty functions. *Handbook of evolutionary computation* 97, 1 (1997), C5.

[23] Edward Snelson and Zoubin Ghahramani. 2006. Sparse Gaussian processes using pseudo-inputs. *Advances in Neural Information Processing Systems* 18 (2006), 1259–1266.

[24] Edward Snelson and Zoubin Ghahramani. 2007. Local and global sparse Gaussian process approximations. In *Artificial Intelligence and Statistics*. PMLR, 524–531.

[25] Il'ya Meerovich Sobol'. 1967. On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki* 7, 4 (1967), 784–802.

[26] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. 2009. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995* (2009).

[27] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1009–1024.

[28] Ziyu Wang, Masrour Zoghi, Frank Hutter, David Matheson, Nando De Freitas, et al. 2013. Bayesian Optimization in High Dimensions via Random Embeddings.. In *IJCAI*. 1778–1784.

[29] Christopher KI Williams and Carl Edward Rasmussen. 2006. *Gaussian processes for machine learning*. Vol. 2. MIT press Cambridge, MA.

[30] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*. 338–350.