I thank my family – Gabriella, Gaetano, Lorenzo, and Marco – for the unconditional love they bestowed upon me

I thank my friends, especially Vacchiz, Jack, Moro, Bellinz, Tancre, and Mr. ML, for the immense happiness they brought me

I thank my professors, chiefly my supervisor Om, for the invaluable knowledge and passion they conveyed to me

I thank my struggles for the profound self-awareness they instilled in me

Here's to everything that made the past three years magnificent

Bayesian Optimization for Database Management

Giulio Caputi

Contents

1 Introduction

Database Management Systems (DBMSs) are software systems employed to work with databases. They serve as an interface between users and databases, handling crucial operations like data storage, retrieval, and management, and providing tools to create, read, update, and delete data. Such systems are used by nearly every organization handling a significant amount of data, to work with several types of databases, including relational, NoSQL, and distributed databases. The most popular DBMSs include Oracle Database, MySQL, and MariaDB.

Working with optimized database systems is crucial for companies. Indeed, efficient DBMSs imply fast data retrieval (especially important in time-sensitive applications), real-time access to information, and enhanced decision-making (as efficient DBMSs also make it easier to integrate advanced analytics tools to extract meaningful insights). Additionally, well-managed database software ensures data integrity and security, scalability of operations, and a significant reduction in costs and resources associated with hardware upgrades and data storage, maintenance, and retrieval. *Dulcis in fundo*, optimized DBMSs dramatically improve the customer experience, as fast data retrieval and real-time access to information ensure that customer interactions are smooth and responsive.

Typical performance metrics of DBMSs include latency, throughput, and running time, all of which are generally computed by internal tracking applications. Latency is defined as the delay between the initiation of a single operation and its completion, throughput is the number of queries or transactions that can be processed per unit of time, and running time is simply the total time taken to execute a series of operations. In this work, the focus is on minimizing the latter, but many of the findings presented here are easily transferable to attempts to optimize other metrics.

Given that DBMSs are expensive-to-evaluate black-box functions with many inputs, the chosen approach to minimize the running time of these software systems is Bayesian Optimization (BO). What follows discusses how this method works, why it is used in this context, and which challenges are common when employing BO to optimize DBMSs. Subsequently, we discuss the functioning of two real-world commercially available

algorithms, which both improve DBMS performance by making use of the concepts explained in this work. Ultimately, some ideas to further improve these state-of-the-art methods are presented.

2 Problem Overview

Much of the literature on improving Database Management System (DBMS) performance discusses ways to optimize their physical design (Chaudhuri and Narasayya, 2007), which often involves attempts to develop models for automatically selecting a set of materialized views and indexes (Agrawal, Chaudhuri and Narasayya, 2000), and/or for performing database partitioning and replication (Curino et al., 2010). The focus of this thesis is instead on finding an optimal setting configuration for the target DBMS, rather than on optimizing its internal features. Specifically, DBMSs have dozens (or sometimes hundreds) of configurable parameters, called *knobs*, which control various aspects of these systems, such as the amount of memory allocated to different activities, how joins are processed, how often to write data to storage, *et cetera*. Changing the values of such knobs can greatly affect the performance of DBMSs.

When looking for an optimal knob configuration, many companies use a manual approach, chiefly for simplicity reasons (Debnath et al., 2008). This implies hiring Database Administrators (DBAs) to manually tune DBMSs. However, this has several problems, which include the following:

- Finding an optimal DBMS knob configuration is an NP-hard problem (Sullivan, Seltzer and Pfeffer, 2004). Indeed, in light of the fact that some database systems feature hundreds of knobs, and many of them take on continuous values, the search space is enormous, and thus manual approaches are highly inefficient.
- DBAs are sometimes prohibitively expensive for many organizations (Bureau of Labor Statistics, 2023).
- Humans can reasonably only target a handful of knobs.
- The effects of configurations on the performance of these systems are greatly

non-linear, and the interrelations among knobs go well beyond what humans can reason about.

- The default configurations of many DBMSs are generally far from optimal (Pavlo, 2022).

Rather than on manual approaches, this work concentrates on automatic algorithms for identifying optimal knob configurations, for a given DBMS application. The current literature undoubtedly shows that this methodology is beneficial (e.g., Pavlo et al., 2017b). There are two general ways to proceed here. One possibility is to employ static (i.e., hard-coded) rules, meaning general principles that are known to work (e.g., Kwan et al., 2002; Montgomery and Reif, 2018; Vasyliev et al., 2024). This addresses many of the problems outlined above, and in fact can lead to significant improvements over default configurations (Zhu et al., 2017). Nevertheless, such methods only target around a dozen knobs (that are considered the most influential), and typically only work with one specific type of DBMS, because they are often developed by vendors themselves, and thus only support one type of DBMS (e.g., Dias et al., 2005; Narayanan, Thereska and Ailamaki, 2005). Most importantly, optimization techniques employing static rules often assume the number of knobs in a DBMS to be fixed, in spite of the fact that such a number is constantly increasing (since 2001, the number of knobs has increased approximately threefold for Postgres and around sixfold for MySQL).

Introducing now the problem itself, let $f(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}$ be a function representing the behavior of a certain DBMS application. Specifically, this function takes as input a vector $\boldsymbol{x} \in \mathbb{R}^d$ representing a knob configuration (this DBMS application has at most d tunable knobs, and we decide to tune d of them), and returns the running time of the application it represents, for a given input configuration $\boldsymbol{x}$. This function might also return an error, and the discussion of this case is held in Section 4.5. An important feature of the objective function is that it has a stochastic component. Indeed, there are uncontrollable variables (e.g., other users' activity) that influence the running time of DBMSs. The fact that the objective is not deterministic gives rise to many of the issues outlined in Section 4, which discusses common challenges in automatic DBMS tuning, and to some departures from standard algorithms as treated by the majority of the literature, especially in Section 3,

which is about Bayesian Optimization (BO) itself.

Evaluating the objective (meaning, executing the target DBMS application and recording the associated running time) is expensive (a single run could take hours), and is generally done by setting the input configuration to a copy of the actual DBMS to optimize (in order not to jeopardize real performance during the training phase), executing it, and recording its running time. Going into the specifics of how this particular step is performed would require knowledge of the peculiar algorithms that firms use for this task, which is outside the scope of this thesis.

The values of each knob have constraints, which range from simple (e.g., some knobs only take on integer values) to complex (e.g., hardware-imposed limitations on cache size, buffer pool size, and log file size). Such constraints can be either known or unknown. In this section, the focus is on the former, and a discussion of how to handle the latter is held in Section 4.5. Known constraints depend on the role of each knob and are static (meaning that they define a fixed range the values of which can be taken by the knob, where this range does not depend on the values of other knobs). In this work, such known constraints are hard-coded in a function $c(\boldsymbol{x}) : \mathbb{R}^d \to \{0, \infty\}$ defined as

$$c(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \boldsymbol{x} \text{ is valid} \\ \infty & \text{if } \boldsymbol{x} \text{ is not valid} \end{cases} \tag{1}$$

So $c(\boldsymbol{x})$ returns 0 if the knob configuration $\boldsymbol{x}$ satisfies the known constraints, and ∞ otherwise. Evaluating this function takes $O(1)$ time, as everything it does is checking hard-coded rules. Defining $f_c(\boldsymbol{x}) \coloneqq f(\boldsymbol{x}) + c(\boldsymbol{x})$, the problem we want to solve is finding

$$\boldsymbol{x}^* = \operatorname*{argmin}_{\boldsymbol{x}} f_c(\boldsymbol{x}) \tag{2}$$

Handling the constraints in this way effectively turns the problem from a constrained-optimization task to an unconstrained-optimization endeavor, in which configurations violating constraints will be naturally avoided, since they result in an objective value of ∞. Moreover, it is worth noting that when the input $\boldsymbol{x}$ is not valid, the DBMS does not start running, so evaluating $f_c(\boldsymbol{x})$ takes $O(1)$ time in this case

(exactly as computing $c(\boldsymbol{x})$), which implies that any optimization algorithm will be fast in identifying the known constraints and will soon start suggesting only valid configurations. In practice, global minima of such complex and highly non-linear functions are often hard to find. Therefore, results are generally considered satisfactory if they improve over the benchmark used, which is typically the performance of the target DBMS with the current knob configuration. The approach followed in this thesis focuses on machine learning (ML) algorithms that study the behavior of a given DBMS application and suggest knob configurations based on it.

3 Optimization

3.1 Introduction to the Bayesian Optimization Approach

The idea behind Bayesian Optimization (BO) (e.g., Mockus, 1989) is to minimize or maximize an expensive-to-evaluate black-box function, $f_c(\boldsymbol{x})$ in our case, by using a probabilistic model to predict the function value in unexplored regions, and then decide where to evaluate next based on these predictions. This method does not make use of derivatives of $f_c(\boldsymbol{x})$, as computing (or approximating) them could involve substantial additional computational time. This is indeed a type of *derivative-free optimization*. The first component of this method is a *probabilistic model*, the most common of which is the Gaussian Process (GP). This is used to define a prior and a posterior distribution over the objective function. The second component is an *acquisition function*, which guides the optimization process by deciding where to sample next. After having selected which knobs to tune and having collected data $D_n = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$, the optimization loop is the following:

- Fit the GP on the data D_n (which means updating the GP)
- Optimize the acquisition function to select the next point to sample, $\boldsymbol{x}_{n+1}$
- Run the DBMS (i.e., evaluate the function) at $\boldsymbol{x}_{n+1}$, and record the associated running time y_{n+1} (i.e., the output of the function to optimize)
- Update the dataset $D_{n+1} = D_n + \{(\boldsymbol{x}_{n+1}, y_{n+1})\}$

- Repeat until a stopping condition is met

Now this process is described in depth.

3.2 Initial Data Collection

The first step is to collect some initial data regarding the behavior of the objective. Here the aim is essentially to build the set $D_n = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, composed of n input-output pairs, where each $\boldsymbol{x}_i \in \mathbb{R}^d$ is a knob configuration, and each $y_i \in \mathbb{R}$ is the running time of the target Database Management System (DBMS) application recorded with configuration $\boldsymbol{x}_i$. For this sampling task, the Latin Hypercube Sampling (LHS) algorithm is discussed (e.g., McKay, Beckman and Conover, 1979).

LHS is a sampling technique that spreads out points with the goal of encouraging diversity of data. This should give models a more complete picture of the objective, which in turn should lead to more accurate optimization. LHS is a stochastic algorithm, so it disperses samples in a probabilistic sense, aiming at uniformity. The key idea behind this approach is to ensure that the sample is representative of the entire distribution, by dividing the distribution of each input (i.e., of each knob) into equally probable intervals, and sampling from each interval. The aim is to sample n points from a d-dimensional set, where each dimension represents a different knob.

The first step is to partition the range of possible values for each dimension into intervals of equal probability. This ensures that the sampled points are spread out evenly across the entire range of the distribution. For each dimension i (where $i = 1, ..., d$), we divide the cumulative probability scale into n equal segments, with each segment corresponding to a probability interval of $\frac{1}{n}$. The boundaries of these intervals are computed, in the general case, using the inverse cumulative distribution function (cdf), also known as the quantile function, of the distribution for that dimension. So, in general, the lower boundary $b_{l,ij}$ and upper boundary $b_{u,ij}$ of the j^{th} interval in the i^{th} dimension are given by

$$b_{l,ij} = F_i^{-1}\left(\frac{j-1}{n}\right) \tag{3}$$

and

$$b_{u,ij} = F_i^{-1}\left(\frac{j}{n}\right) \tag{4}$$

where F_i^{-1} is the inverse cdf of the i^{th} dimension, and $j = 1, ..., n$.

Once the intervals are determined, for each dimension i and for each interval j, the LHS algorithm randomly selects one value within the interval. The sampled value v_{ij} from the j^{th} interval of the i^{th} dimension can be expressed as

$$v_{ij} = F_i^{-1}\left(\frac{j-1+u_{ij}}{n}\right) \tag{5}$$

where u_{ij} is a random number uniformly distributed in the range [0,1]. This ensures that the sampled value lies within the j^{th} interval in the i^{th} dimension.

Now, for each number j, the algorithm creates a sample point by combining the sampled values from each dimension, so that the j^{th} sample point is the vector $[v_{1j}, ..., v_{dj}]$. The resulting set of n points forms the Latin Hypercube Sample. These points should be evenly distributed across the entire range of each dimension, ensuring that the sample is representative of the distribution. Due to the just-described methodology, LHS provides a more efficient and representative sample compared to simple random sampling, especially in the context of DBMS tuning (Kanellis, Alagappan and Venkataraman, 2020).

Since, in this problem, each dimension denotes a knob, it is typically possible to simplify the just-outlined procedure by assuming, for each knob, a uniform distribution in the range determined by the known constraints. For the i^{th} knob, we denote this range as $[a_i, b_i]$, with $a_i < b_i$. This choice is reasonable because the objective values are not "observed in nature", but depend on the choices of input configurations. The use of the uniform distribution implies that each interval within a dimension will have the same width of $w_i = \frac{(b_i - a_i)}{n}$. So, the lower and upper boundaries of the j^{th} interval in the i^{th} dimension are simplified to, respectively,

$$b_{l,ij} = a_i + (j-1)w_i \tag{6}$$

and

$$b_{u,ij} = a_i + jw_i \tag{7}$$

Additionally, if we assume a uniform distribution for each knob, each sampled value v_{ij} simplifies to

$$v_{ij} = a_i + (j - 1 + u_{ij})w_i \tag{8}$$

which again lies inside the j^{th} interval in the i^{th} dimension.

3.3 Selecting which Knobs to Tune

Since DBMSs sometimes feature more than 100 tunable knobs, and the efficacy of any optimization algorithm suffers from the curse of dimensionality (e.g., Venkat, 2018), a selection of which knobs to tune has to be performed. There are two criteria a knob has to satisfy in order to justify its tuning. First, it is paramount that by changing it we do not risk endangering the reliability or the safety of the system. A discussion of this case is held in Section 4.3. For now, it is sufficient to know that, in the vast majority of cases, such "dangerous" parameters are either non-tunable, or anyway present in some kind of "black list" in the DBMS documentation. Such a list contains knobs the values of which should be changed with particular caution (if changed at all). Second, to justify its tuning, a knob has to be relevant, meaning that altering its value must have a certain effect on the behavior of the objective. This is the criterion discussed in this section.

To assess the relevance of knobs, a good starting point is the current literature, which often provides useful guidance in this regard. Nevertheless, a review of scientific papers cannot substitute for a variable-selection task performed directly with the target DBMS, since the most impactful variables often change depending on the application (Kanellis et al., 2022). Among the plethora of existing feature-selection techniques, one which has proven to be particularly effective in the context of DBMS tuning is the Least Absolute Shrinkage and Selection Operator (Lasso) algorithm (Van Aken et al., 2017), due to its advantages over other feature selection methods in terms of stability, computational efficiency, and accuracy (Efron et al., 2004; Tibshirani, 1996).

We start by collecting data on the behavior of the objective as described in Section 3.2. We then fit a linear regression model with this data, where knobs are the regressors and running time is the dependent variable. This is usually performed with the Ordinary Least Square (OLS) method. To perform variable selection with Lasso, the usual OLS

loss function $L = (\boldsymbol{y} - X\boldsymbol{\beta})^T(\boldsymbol{y} - X\boldsymbol{\beta})$ has to be modified. Indeed, we add a penalty term to it, which encourages the coefficients of less important predictors to shrink towards zero, effectively performing variable selection. Specifically, the Lasso loss function is

$$L = (\boldsymbol{y} - X\boldsymbol{\beta})^T(\boldsymbol{y} - X\boldsymbol{\beta}) + \lambda \sum_{j=2}^{d+1} |\beta_j| \tag{9}$$

Here, $\boldsymbol{y} \in \mathbb{R}^{n\times 1}$ contains the running times from the data-collection phase, and $X \in \mathbb{R}^{n\times(d+1)}$ is the matrix of predictors (with each row corresponding to a knob configuration, with an additional 1 as the coefficient of the intercept). These predictors are often standardized, to ensure that the regularization penalty is applied uniformly to all regressors. Moreover, $\boldsymbol{\beta} \in \mathbb{R}^{(d+1)\times 1}$ is the vector of coefficients (intercept included), and $\lambda \in \mathbb{R}_+$ is the regularization parameter that controls the amount of shrinkage applied to the coefficients. This shrinkage occurs because every non-zero weight β_j increases the value of the loss. This is a type of L1 regularization, as we add the absolute values of each coefficient to the loss function. Notice that the summation in formula (9) starts from $j = 2$, so the intercept is not shrunk.

To select the most relevant knobs, Lasso starts by fitting the model with a high penalty term, which shrinks all weights to 0 (which means no knobs are selected in the regression model). The algorithm progressively decreases λ, re-fits the model, and tracks which knobs are added to it. The order in which knobs are added is also considered the order of their importance. Deciding how many knobs to add is left to the user. Something worth noting is that identifying the most important knobs using Lasso generally results in selecting many of the same variables that Database Administrators (DBAs) also consider important (Van Aken et al., 2021). After having chosen which knobs to work with, a second data-collection phase is conducted, again as explained in Section 3.2, this time encouraging sparsity only in terms of the important knobs.

3.4 Gaussian Process

This is the probabilistic model used to represent our beliefs about the function to optimize. It provides a prior and a posterior distribution over the function $f_c(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}$ to

optimize. A GP (e.g., Rasmussen, 2003) is a collection of random variables fully specified by a mean function $m(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}$ and a covariance function $k(\boldsymbol{x}, \boldsymbol{x}') : \mathbb{R}^d, \mathbb{R}^d \to \mathbb{R}_+$, where $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^d$ are points in the input space (that is, knob configurations). The covariance function is also called kernel function. This is a *non-parametric* approach, meaning that it can adapt to the complexity of the data without requiring a predetermined number of parameters. We assume any finite number of realizations of $f_c(\boldsymbol{x})$ to be jointly Gaussian (i.e., to follow a multivariate Gaussian distribution) with mean vector returned by $m(\boldsymbol{x})$ and covariance matrix K, with $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. So, we write $f_c \sim \mathcal{N}(m(\boldsymbol{x}), K)$. Now, a description of the mean function $m(\boldsymbol{x})$ and the covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$ is held, before discussing their usage in the context of BO.

3.4.1 Mean Function

The mean function $m(\boldsymbol{x}) = \mathbb{E}[f_c(\boldsymbol{x}) \mid D_n]$ outputs the expected value of the GP at a point $\boldsymbol{x}$. In this thesis it is chosen to be zero, for simplicity reasons, so that $m(\boldsymbol{x}) = 0 \quad \forall \boldsymbol{x} \in \mathbb{R}^d$. This choice might seem bizarre, as in this application the function $f_c(\boldsymbol{x})$ returns the running time of a certain DBMS with knob configuration $\boldsymbol{x}$, and it is thus strictly positive. However, this assumption does not imply that the actual expected value of the function $f_c(\boldsymbol{x})$ is zero for all inputs $\boldsymbol{x}$, but rather that we are centering our model around a mean of zero for computational convenience and without loss of generality (Williams, 1998).

3.4.2 Kernel Function

This function returns a measure of the covariance between the two input vectors (i.e., a positive scalar), and is used to construct the matrix K. There is a number of possible choices for the kernel function, all of which satisfy symmetry (so that $k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}', \boldsymbol{x})$ for all $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^d$) and positive semi-definiteness (so that, for any finite set of points, the resulting covariance matrix must be positive semi-definite). In this thesis, the chosen kernel function is the Radial Basis Function (RBF), also known as the Squared

Exponential (SE), due to its simplicity and flexibility. This function is

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sigma_f^2 \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2l^2}\right) \tag{10}$$

In this formula, $\sigma_f^2 \in \mathbb{R}_+$ is the signal variance, $\exp(x)$ denotes e^x for a generic $x \in \mathbb{R}$, $l \in \mathbb{R}_+$ is the length scale, and $\|\boldsymbol{x} - \boldsymbol{x}'\|^2 \in \mathbb{R}_+$ is the squared Euclidean distance between vectors $\boldsymbol{x}$ and $\boldsymbol{x}'$ (so that $\|\boldsymbol{u} - \boldsymbol{v}\|^2 = \sum_{i=1}^{d}(u_i - v_i)^2$ with $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d$). The signal variance determines by how much the function values can deviate from the mean, while the length scale controls the smoothness of k, as smaller l values make the function change more rapidly with small differences between $\boldsymbol{x}$ and $\boldsymbol{x}'$, and larger l values instead make the function smoother. This kernel assumes that the function being modeled is infinitely differentiable.

The signal variance σ_f^2 and the length scale l are typically estimated by maximizing the marginal likelihood of the observed data D_n under the GP model. This marginal likelihood is

$$p(\boldsymbol{y} \mid X, \sigma_f, l) = \int p(\boldsymbol{y} \mid f, X) p(f \mid X, \sigma_f, l)\, df \tag{11}$$

where $p(f \mid X, \sigma_f, l)$ is the prior distribution over the function values f under the GP model with RBF kernel function. To increase numerical stability and turn products into sums (with which it is easier to work), it is common to use the logarithm of the marginal likelihood (called log likelihood), since its maximizer is also the maximizer of the original marginal likelihood. The log likelihood function is

$$\log(p(\boldsymbol{y} \mid X, \sigma_f, l)) = -\frac{1}{2}\boldsymbol{y}^T(K + \sigma_n^2 I)^{-1}\boldsymbol{y} - \frac{1}{2}\log(\det(K + \sigma_n^2 I)) - \frac{n}{2}\log(2\pi) \tag{12}$$

where σ_n^2 is the noise variance, and $\det(K + \sigma_n^2 I)$ denotes the determinant of the matrix $K + \sigma_n^2 I$. So, σ_f^2 and l are estimated by maximizing this log marginal likelihood, typically with gradient-based methods such as gradient

descent (e.g., Lecun et al., 1998) or quasi-Newton methods (e.g., Broyden, 1967).

Now that $m(\boldsymbol{x})$ and $k(\boldsymbol{x}, \boldsymbol{x}')$ have been described, we proceed with a discussion of how they are employed in BO. We define the vector $\boldsymbol{k}_* \in \mathbb{R}^n$ as the vector of covariances between the input vector $\boldsymbol{x}_*$ and the data in X. So, each i^{th} element of $\boldsymbol{k}_*$ is $\boldsymbol{k}_{*_i} = k(\boldsymbol{x}, X_i)$, where $X_i \in \mathbb{R}^{1\times d}$ here denotes the i^{th} row of X, without the first element (which is 1, the coefficient of the intercept). We further define $k_{**} \in \mathbb{R}_+$ as the variance of the input vector $\boldsymbol{x}_*$, so that $k_{**} = k(\boldsymbol{x}_*, \boldsymbol{x}_*)$. The GP posterior mean $\mu_* \in \mathbb{R}$ and posterior variance $\sigma_*^2 \in \mathbb{R}_+$ at a new point $\boldsymbol{x}_*$ are given by

$$\mu_* = m(\boldsymbol{x}_*) + \boldsymbol{k}_*^T[K + \sigma_n^2 I]^{-1}(\boldsymbol{y} - m(X)) \tag{13}$$

and

$$\sigma_*^2 = k_{**} - \boldsymbol{k}_*^T[K + \sigma_n^2 I]^{-1}\boldsymbol{k}_* \tag{14}$$

Here, $m(X) \in \mathbb{R}^n$ is the mean function applied pairwise to the rows of the matrix X (again excluding the coefficient of the intercept), and $\boldsymbol{y} \in \mathbb{R}^n$ is the vector of observed values in the training set.

We add the noise term σ_n^2 to the diagonal elements of K to represent the independent noise at each observation. There are several methods to estimate it. Two possible approaches are maximum likelihood estimation and cross-validation. The former approach implies maximizing the log likelihood of the available data (that is, formula (12) above with σ_n^2 as a parameter) with respect to σ_n^2. The latter approach instead involves splitting the data into training and validation sets, then repeatedly fitting the GP model on the training set with different σ_n^2 values, and identifying the σ_n^2 value that minimizes the prediction error on the validation set.

As said before, we assume $m(\boldsymbol{x}) = 0 \quad \forall \boldsymbol{x} \in \mathbb{R}^d$ without loss of generality. Therefore, the mean of the GP posterior simplifies to

$$\mu_* = \boldsymbol{k}_*^T[K + \sigma_n^2 I]^{-1}\boldsymbol{y} \tag{15}$$

Now, the predictive distribution at a point $\boldsymbol{x}_*$ is Gaussian with mean μ_* and variance σ_*^2.

The just-outlined framework makes it now possible to describe what a BO algorithm does for each new point $\boldsymbol{x}_*$:

- By using the kernel function, it computes the covariance matrix K for the training data, the $\boldsymbol{k}_*$ vector indicating the variances between the training data and the new point, and the variance k_{**} for the new point.
- It then computes the predictive mean μ_* and variance σ_*^2
- Now, it is possible to make predictions for $\boldsymbol{x}_*$. The prediction is a Gaussian distribution with parameters μ_* and σ_*^2
- If the function value at $\boldsymbol{x}_*$, call it y_*, is observed, it has to be added to the set of observations D

After having discussed the usage of GPs in BO, an analysis of acquisition functions is presented in the following section.

3.5 Acquisition Function

Based on the GP posterior, the acquisition function is used to decide where to sample next. It guides the search process by quantifying the expected utility (i.e., performance improvement) from evaluating the objective function at a new point. It should balance *exploration* (probing areas with high uncertainty) and *exploitation* (focusing on areas with high predicted values). Common acquisition functions include Expected Improvement (EI) (e.g., Jones, Schonlau and Welch, 1998), Probability of Improvement (PI) (e.g., Wang et al., 2023), and Upper Confidence Bound (UCB) (e.g., Srinivas et al., 2010). These functions can also be combined together to improve the overall model performance (e.g., De Ath et al., 2019).

3.5.1 Expected Improvement

For a new point $\boldsymbol{x}_*$, this function is defined as

$$EI(\boldsymbol{x}_*) = \mathbb{E}[\max(0, y^+ - y_*)] \tag{16}$$

Here, y^+ is the current best observed value of the objective function (not to be confused with y^*, the unknown global minimum of $f_c(\boldsymbol{x})$), and y_* is the realization of the random variable representing the function value at $\boldsymbol{x}_*$ (modeled by the GP), so essentially $y_* = Y(\boldsymbol{x}_*) \mid D_n$. In general, the most convenient way to compute EI is through Monte Carlo (MC) approximation. This involves drawing $y_t \sim Y(\boldsymbol{x}_*) \mid D_n$, for $t = 1, ..., T$, from their posterior predictive distribution, and approximating EI as

$$EI(\boldsymbol{x}_*) \approx \frac{1}{T} \sum_{t=1}^{T} \max(0, y^+ - y_t) \tag{17}$$

As T tends to ∞, such an approximation becomes exact. This approach works regardless of the posterior distribution $Y(\boldsymbol{x}_*) \mid D_n$, as long as it is possible to simulate random draws from it. However, if the posterior distribution is Gaussian, and GP surrogates are being used, as it is the case in the application which is the object of this thesis, the EI function has the following closed form

$$EI(\boldsymbol{x}_*) = (y^+ - \mu_* - \xi)\Phi\left(\frac{y^+ - \mu_* - \xi}{\sigma_*}\right) + \sigma_* \phi\left(\frac{y^+ - \mu_* - \xi}{\sigma_*}\right) \tag{18}$$

Here, Φ is the standard Gaussian cumulative distribution function (cdf), ϕ is the standard Gaussian probability density function (pdf), and $\xi \in \mathbb{R}_+$ is a small positive number the role of which is to balance exploration and exploitation. In particular, in the limit case in which $\xi = 0$, the EI function focuses solely on improving performance over the current best value y^+. This implies that the algorithm focuses on exploitation of already-gained information regarding regions with a high μ_* and low σ_*. Instead, when $\xi > 0$, this parameter effectively shifts the target for improvement by a small amount. This translates to the fact that even if μ_* is slightly below y^+, the algorithm can still consider points with higher uncertainty σ_* as potentially valuable. When this is the case, the algorithm is performing exploration of unknown regions. Therefore, a higher ξ encourages exploration, and it is thus set high at the early stages of the optimization process. As this process continues, ξ

decreases, and the algorithm focuses more on exploiting information regarding known good regions.

3.5.2 Probability of Improvement

This function computes the probability that a new sample will improve over the current best. It is intuitively defined as

$$PI(\boldsymbol{x}_*) = \mathbb{P}(y_* < y^+ \mid D_n) \tag{19}$$

As it is the case for EI, also here the most convenient way to compute PI is, in the general case, to employ MC sampling, which leads to the following approximation

$$PI(\boldsymbol{x}_*) \approx \frac{1}{T} \sum_{t=1}^{T} \mathbb{I}_{y_t < y^+} \tag{20}$$

Here, the y_t's are again drawn from the posterior $Y(\boldsymbol{x_*}) \mid D_n$ for $t = 1, ..., T$, and $\mathbb{I}$ is the indicator function, so that

$$\mathbb{I}_{y_t < y^+} = \begin{cases} 1 & \text{if } y_t < y^+ \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

Again, since in this application the posterior is Gaussian, we have a closed form for the PI function, which is

$$PI(\boldsymbol{x}_*) = \Phi\left(\frac{y^+ - \mu_* - \xi}{\sigma_*}\right) \tag{22}$$

In this closed form, ξ plays the same role as in EI.

3.5.3 Upper Confidence Bound

The UCB function is

$$UCB(\boldsymbol{x}_*) = \mu_* + \eta\sigma_* \tag{23}$$

Here, $\eta \in \mathbb{R}_+$ is a tunable parameter that controls the trade-off between exploration and exploitation. Higher η values favor exploration over

exploitation. Indeed, when η is high, the second term of the UCB function, $\eta\sigma_*$, becomes more dominant, and so points with higher uncertainty σ_* will have higher UCB values. As a result, the algorithm is more likely to explore regions of the input space where the model is uncertain about the objective function value. As it is the case for ξ in the other two acquisition functions, η is generally relatively high in early stages of the optimization process, to encourage exploration. As the optimization continues, η decreases, to let the algorithm focus on known good regions.

3.5.4 Slight Changes for Noisy Objectives

What has been said so far regarding acquisition functions needs small adjustments when the objective function is noisy, as opposed to deterministic, which is the case in this thesis with $f_c(\boldsymbol{x})$. There is nothing wrong with the form of the previously discussed acquisition functions, but special attention must be paid to how their components are defined. In particular, since $f_c(\boldsymbol{x})$ is not deterministic, then $Y(\boldsymbol{x}_i) \neq y_i$, and instead the responses $Y_1, ..., Y_n$ are themselves random variables. This implies that y^+ as well is a random variable, so using the closed-form equations presented above might be problematic. A possible solution is to set $y^+ = \min_{\boldsymbol{x}} \mu(\boldsymbol{x})$, which is essentially our model's guess about the global minimum of $f_c(\boldsymbol{x})$. This is sometimes referred to as the "plug-in" method (Picheny et al., 2013). Using a deterministic $\min_{\boldsymbol{x}} \mu(\boldsymbol{x})$ instead of the random variable y^+ works well in practice, despite under-accounting for a degree of uncertainty (Gramacy, 2021).

At each iteration of the optimization process, the next point to sample is selected by maximizing the chosen acquisition function. Therefore, we have two nested optimization loops, as the one for the function $f_c(\boldsymbol{x})$ itself requires, at every step, the optimization of the acquisition function. The choice of which acquisition function to use has the potential to heavily influence the quality of the optimization process. This choice depends on the specific characteristics of the problem and on the goals one aims at achieving. When choosing which acquisition function to utilize, the following factors should be considered:

- The EI function balances exploration and exploitation well and has good empirical performance in many applications. However, it can sometimes get stuck in local optima, especially in high-dimensional problems (Ament et al., 2024).

- The UCB function can be more effective than the EI function in exploring the search space, since it places a stronger emphasis on uncertainty. EI, on the other hand, is more exploitative, as it focuses more on regions close to the best observed outputs. Therefore, in the case of a huge search space, or in the presence of an objective with many local optima, the UCB is preferable; if instead these two conditions do not hold, EI is generally better (Raihan et al., 2023).

- The PI function has the advantage of being simple and intuitive, focusing directly on the probability of finding an improvement. However, it tends to be more conservative than the other two acquisition functions (meaning that the steps taken when working with PI are quite small), which leads to less exploration and potentially slower convergence to the global optimum. The reason is that the input with the greatest probability of improving $f_c(\boldsymbol{x})$ may not hold the highest potential for large improvement (intuitively, it is obvious that maximizing the probability of getting any improvement is significantly different from maximizing the improvement itself). This poses problems, especially given that the objective is expensive to evaluate.

- The EI function contains PI as a component in a larger expression. Indeed, $EI(\boldsymbol{x}_*) = (y^+ - \mu_* - \xi)\Phi\left(\frac{y^+ - \mu_* - \xi}{\sigma_*}\right) + \sigma_* \phi\left(\frac{y^+ - \mu_* - \xi}{\sigma_*}\right)$ is the sum of two terms, where one is $PI(\boldsymbol{x}_*) = \Phi\left(\frac{y^+ - \mu_* - \xi}{\sigma_*}\right)$ weighted by the amount by which the predictive mean is better than y^+ (with the ξ parameter), and the other is instead the predictive variance weighted by a Gaussian density evaluation. This essentially implies that all the information used by PI is also used in EI, which often makes the latter more powerful than the former.

Up to this point we have described a process that, after having collected initial data on input-output relations of the objective $f_c(\boldsymbol{x})$ with LHS and having used it to select the most relevant knobs to tune, re-performs the same LHS data-collection procedure as in the beginning, but this time only changing the knobs considered relevant. Subsequently, the

process iteratively maximizes an acquisition function, evaluates $f_c(\boldsymbol{x})$ on the maximizer of such acquisition function, and updates $D_n = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ with this new data. As already introduced before, this loop continues until a stopping condition is met.

3.6 Stopping Condition

Stopping conditions, also called terminal criteria, are occurrences which determine that the BO algorithm should stop looking for the optimal value of $f_c(\boldsymbol{x})$. The simplest stopping condition is to set a maximum number of iterations, after which the algorithm stops. Another common terminal criterion is the convergence of the parameters, so that the BO algorithm stops if $\|\boldsymbol{x}_t - \boldsymbol{x}_{t-1}\| < \varepsilon$, for a small and positive ε, generally in the order of 1e-3 or smaller. This means that new information is changing the input to try by an extremely small amount. One can also focus on the convergence of the objective itself. In this case, the BO algorithm ceases to look for the optimum if $|f_c(\boldsymbol{x}_t) - f_c(\boldsymbol{x}_{t-1})| < \delta$, again for a small and positive δ. This indicates that the marginal improvement of the objective function associated with new BO iterations is minimal. It is also possible to stop the iterations when the acquisition function converges, so when $|\alpha(\boldsymbol{x}_t) - \alpha(\boldsymbol{x}_{t-1})| < \theta$, where $\alpha(\boldsymbol{x})$ is the acquisition function, and θ is again a small and positive number. Since the acquisition function guides the selection of the next point to evaluate, when changes in this function become negligible, so are changes in the points the BO algorithm is evaluating. In practice, a combination of these criteria is often used, sometimes also in conjunction with performance thresholds (if there is a satisfactory known $f_c(\boldsymbol{x})$ value, we can stop if we reach a better value) and resource constraints (Shahriari et al., 2015).

3.7 Using Bayesian Optimization with a Function-Learning Approach

As it is evident from the content of this thesis, trying to learn the function $f_c(\boldsymbol{x})$ itself is generally a difficult task, as it requires many evaluations of the DBMS (which are expensive), and the number of knobs to tune is typically high, which makes the search space enormous. Nevertheless, there are instances in which a naive function-learning approach can be beneficial if used in conjunction with BO. Specifically, it makes sense

in cases in which $f_c(\boldsymbol{x})$ is not particularly noisy, evaluating it takes relatively little time, and the number of knobs to tune is limited (so d is small, generally less than 5). This case is more relevant than it may seem at first glance. Indeed, current literature reports how, in the context of DBMS tuning, it is possible to achieve very good performance improvements over default configurations by only optimizing the knob controlling the size of the redo log file on disk, and the one managing the amount of RAM used for the buffer pool cache (Van Aken et al., 2021).

This function-learning approach involves developing a model to approximate the objective itself, which can be done with notable precision if the above conditions are satisfied. Given a certain DBMS application, the first step is to generate n valid knob configurations (that is, n valid $\boldsymbol{x} \in \mathbb{R}^d$ vectors) to try on the software system. Such $\boldsymbol{x}$ inputs should be generated with LHS, as explained in Section 3.2, to ensure homogeneous sampling. The second step is to set these configurations to a copy of the DBMS, execute it, and record its running time. This aims at constructing a table keeping track of the relationship between running time (the dependent variable) and knob configurations (the independent variables). Such a table has to be employed to train a supervised learning algorithm (e.g., a Random Forest or a Neural Network) that predicts the running time associated with a given knob configuration.

This method (after the model has been developed) allows one to estimate the running time of the target software system with a certain input configuration without actually running the DBMS. Provided that the approximation of this model is reliable, this method can be used in conjunction with the BO approach. This means that, when testing the new configuration obtained by optimizing the acquisition function, it can be set as the input of the model approximating the DBMS, not of the DBMS itself. The purpose is to trade a bit of accuracy for significant speed.

Additionally, having a reliable model of the running time of DBMSs has a huge advantage over optimization methods that only focus on the most promising areas (as BO does). This benefit lies in the fact that such a model enables companies to experiment with different configurations, if for some reason it becomes necessary to depart from the optimal configuration and change the values of some knobs. The reasons why organizations might

decide to do so are often dictated by legal and/or ethical constraints and are outside the scope of this thesis.

3.8 Pros and Cons of this Approach

Employing BO to automatically tune DBMSs has several advantages:

- The main pro is that this approach performs *smart sampling*, which means that the inputs with which the function $f_c(\boldsymbol{x})$ is evaluated are chosen based on specific rules that balance exploration and exploitation. This aims (indirectly) at minimizing the number of evaluations of the objective. We say this method is *sample efficient*, which is important in cases (like the one this thesis discusses) in which executing the function is significantly expensive. To explicitly keep track of the number of evaluations of the objective, it is common in certain applications to record the best value of $f_c(\boldsymbol{x})$ found so far as a function of the number of evaluations of the objective. In this work however, we reason in terms of a fixed "budget" for evaluation, which can be employed at will.

- Another benefit is that the use of an acquisition function enables one to also quantify the uncertainty of predictions (that is, the posterior variance), which is used to balance exploration and exploitation. Indeed, during a more explorative (thus initial) phase, points with higher uncertainty will receive more priority, while during a more exploitative phase, the acquisition function is maximized by points with high expected value μ_*, rather than high uncertainty σ_*.

- Another upside of BO is that it can deliver robust results even with noisy and non-convex functions, as DBMSs are (e.g., Duan, Thummala and Babu, 2009; Zhang et al., 2021).

- A further benefit of this approach is that BO is significantly different from stochastic optimization methods. Indeed, although it is common to randomize the initialization of a BO algorithm (for instance, with the u_{ij} random numbers in LHS), the rules followed here are deterministic. Stochastic optimizations, in which decisions involve some degree of randomness, are known to perform poorly with expensive black-box

functions, due to the fact that random evaluations of the objective are often too wasteful in computational resources (Gramacy, 2021).

- One last advantage to mention is that classical optimization methods emphasize local refinements of $f_c(\boldsymbol{x})$, whereas using surrogate models (in our case, GPs) holds potential for large steps because they view the response surface more broadly, which sometimes helps such methods to escape local minima.

In spite of the existence of these benefits, this approach has also some drawbacks:

- The main disadvantage here is that, although the whole idea of this thesis is to present automated methods to optimize DBMS performance, human choices still have the potential to greatly influence BO algorithms. These choices regard, for instance, which kernel or acquisition function to use, and how to set the parameters of the latter. Such decisions are often non-trivial, and making optimal and informed choices might require additional optimization, at the expense of more computational resources.

- Another problem lies in the fact that BO's black-box nature might pose challenges in cases in which the objective is influenced by complex interactions at multiple levels (as it is often the case for DBMS tuning), since BO does not explicitly require knowledge of the system's inner workings. In such instances, white-box algorithms that understand these interactions might result in close-to-optimal tuning with significantly lower overheads compared to BO (Kunjir and Babu, 2020).

- Although the curse of dimensionality is a problem of any optimization algorithm, BO is particularly sensitive to it, and it tends to perform poorly when the number of dimensions exceeds 20 (Chen et al., 2021). This is something to consider in the context of DBMS, which in some cases have more than 100 configurable parameters. Nevertheless, as explained in Section 3.3, a variable-selection phase is always implemented here.

- A further disadvantage of BO is that, although this method is sample efficient, it is nevertheless computationally expensive, since at each step it has to update

the GP and optimize the acquisition function (which itself can require many steps). Indeed, at every step, the algorithm needs to invert the covariance matrix $K+\sigma_n^2 I$ to compute $\mu_* = m(\boldsymbol{x}_*) + \boldsymbol{k}_*^T[K + \sigma_n^2 I]^{-1}(\boldsymbol{y} - m(X))$ and $\sigma_*^2 = k_{**} - \boldsymbol{k}_*^T[K + \sigma_n^2 I]^{-1}\boldsymbol{k}_*$. Computing the inverse of such an $n \times n$ matrix scales as $O(n^3)$, where n is the number of datapoints, which can pose computational challenges.

- Lastly, when using a GP to optimize $f_c(\boldsymbol{x})$, we assume it to be a smooth function, meaning that points close to each other should have similar function values. While this is often true when the constraints are satisfied, there are cases in which even slight changes to the input knob configuration result in large changes in the DBMS running time, or to ∞ values of $f_c(\boldsymbol{x})$. Nevertheless, BO remains a widely used approach in the literature on automatic DBMS tuning (e.g., Alipourfard et al., 2017; Hsu et al., 2018; Venktesh et al., 2020).

4 Challenges

Some of the challenges inherent to the optimization process described in this thesis have already been outlined above. These include the fact that finding the optimal Database Management System (DBMS) knob configuration is an NP-hard problem (Sullivan, Seltzer and Pfeffer, 2004), the vastness of the search space (the size of which scales exponentially with respect to d, the number of knobs to tune), and the fact that the optimal value of a knob often depends on the values of other knobs. Other challenges are now presented in more detail.

4.1 Cloud Storage

As it is known, many companies use cloud services such as Amazon Web Services (AWS) as the main platform to host their databases. Such platforms offer a cloud computing environment which, in spite of several benefits (not discussed here), poses challenges to DBMS tuning. Indeed, their non-local storage partially reduces the efficacy of the machine learning (ML) algorithms described in this thesis, in light of the fact that DBMS performance depends also on other users' activity, not just on the knob configuration used

or on hardware characteristics. This sometimes leads to high variance in performance (meaning that the same knob configuration set on the same DBMS application might result in quite different running times depending on other users' activity) and to huge read/write latency, given the multi-tenant cloud environment. Additionally, the fact that many papers on this topic do not consider databases living in cloud environments makes it even harder to handle such cases. Good practices to address this issue include running optimization algorithms at roughly the same time of the day (which should reduce the variance in external uncontrollable factors) and testing a certain configuration throughout multiple days and times before establishing that it is optimal.

4.2 Workload

In the context of DBMSs, the term *workload* refers to the set of queries (of any kind) that the system is required to process, i.e., the "amount of work" it has to do. Since, as explained in Section 2, knob configurations are usually first tested on copies of the actual DBMS (in order not to set bad configurations in a production environment, that is, in the "real world"), it is necessary to replicate the real workload that the DBMS performs in such a production environment. This essentially means making the copy of the software system perform "fake" (or, better, "useless") operations, to test its performance. This is what a synthetic workload is: an artificially created workload with no real usage except testing the system.

Setting the right amount of synthetic workload is a non-trivial task. First, good knowledge of the kind of DBMS is required (which is itself not common). Indeed, there exist several kinds of workloads, and a configuration which works well under a certain type of workload might perform poorly with a different kind of queries. For example, an online transaction processing (OLTP) workload generally involves a high volume of short, simple transactions, whereas an online analytical processing (OLAP) workload consists of complex queries that analyze large amounts of data. To select optimal configurations, the synthetic workload with which the optimization process is performed should be similar to the production one with which the DBMS actually works.

Another issue regarding this concept is how much work to assign to the system. Indeed,

on the one hand, in order to compare knob configurations among each other, they have to be executed with the same workload, but at the same time a good configuration must also be scalable, and thus it has to perform well under different volumes of work.
Lastly, even when all of the above is taken into consideration, it is still impossible for synthetic workloads to fully capture the complexity of real-world scenarios. Nevertheless, optimal knob configurations identified with the use of a well-set synthetic workload tend to be efficient also in a production environment.
It is worth noting that, although there are algorithms that can analyze a certain production workload and characterize it (e.g., Pavlo et al., 2017a), such algorithms simply compute a numerical representation of the workload, which does not help in re-creating it in a synthetic environment.

4.3 Knobs to Change Manually

The fact that not 100% of the tuning process can be automatic is self-evident. Indeed, there are some knobs that should never be modified, which are generally variables that, if set to extreme values, result in the best performance, but lead to severe problems, often putting the system's reliability and integrity at risk. A trivial example is the knob which controls the maximum number of users that can access a database concurrently. Indeed, setting this knob to 1 (that is, letting only 1 user per time access a certain collection of databases) would dramatically improve the running time of the system, but would also be detrimental. This implies that, theoretically, the user should know exactly what the system's resources are, and what each knob does, to avoid assigning undesirable values to some of them. Thankfully, since this is often unrealistic, many of such dangerous parameters are either non-tunable, or anyway present in some kind of "black list" in the DBMS documentation. These lists contain knobs the values of which should be changed with particular caution (as stated in Section 3.3). Nevertheless, the fact that some human judgement is involved in setting the values of certain knobs, or in deciding which knobs not to modify, means that the performance of the ML algorithms described here still depends on how the manually adjusted parameters are set.

4.4 Time

As already discussed, searching for the optimal knob values might require a significant amount of time. Therefore, since eventually running time is the actual variable to optimize, a good practice sometimes is to halt a run if the running time is higher, by a pre-determined percentage, than the one of the best configuration seen so far. In the literature, this practice is referred to as *early abort* (e.g., Van Aken et al., 2021). Typically, in such cases, the running time recorded for the configuration is scaled, assuming that, if the system performed a certain percentage $p \in (0, 1)$ of the total assigned workload in t units of time, then the total time required to complete the entire workload would be $\frac{t}{p}$. We generally compute p as $\frac{w_c}{w_a}$, where w_c denotes the completed workload and w_a denotes the total assigned workload, both measured, for instance, in numbers of equally complex queries. In spite of such attempts to reduce the time required to obtain an optimal configuration, this process is nevertheless inherently long, also due to the necessity to ask for multiple approvals within the organization. It is thus important, before tuning a DBMS, to assess its economic significance and the potential gains from such a tuning, and to proceed only if the expected benefits are greater than the expected costs.

4.5 Failed Configurations

Failed configurations are configurations resulting in a fatal error, i.e., one that makes the DBMS halt. This is the case of unknown constraints, which was previously mentioned in Section 2. A typical example is the case of the value of a knob that exceeds its natural limits determined by hardware characteristics. Once an error occurs, determining which knob(s) caused the failure is hard. Indeed, sometimes the bounds of the values of some knobs depend on the values of other knobs, so that the variable x_i might normally take on values in a certain range without causing issues, but if another knob x_j takes on certain values, the range in which x_i can live without causing errors changes.
As an example, let:

- x_i be the knob which determines the amount of memory allocated for shared memory buffers used to cache table and index data. This knob has different names depending on the type of DBMS used, let us denote it as *shared_buffers*. Increasing this value

can improve read performance, but at the same time consumes more system memory

- x_j be the knob which sets the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. Let us call it *work_memory*.

In this case, if the knob *work_memory* (x_j) is set too high, the amount of memory available for the knob *shared_buffers* (x_i) may significantly decrease, potentially leading to out-of-memory errors. Thus, increasing the value of x_j effectively restricts the range in which x_i can live.

Notice that, in this example, it is mentioned that the same knob might have different names in different DBMSs. This is another potential cause of problems, as these differences in the names of variables make it hard to transfer the knowledge gained when tuning one of such software systems to the optimization of a different one.

Another complex problem with failed configurations is what to do with data collected as a result of one of them. Indeed, such data should never be discarded, as the algorithm must know that a certain configuration has to be avoided. At the same time, the running time of a failed configuration is generally artificially low, since the DBMS stopped running before exhausting its workload. To balance these two aspects, the metrics recorded in times of failure are generally scaled as described in Section 4.4, or simply set to ∞, in order to signal to the algorithm to avoid such configurations. It is sometimes beneficial to analyze failed configurations manually as well, as they might reveal insights about how the DBMS itself works.

5 Real-World Examples

The framework presented above is not merely a theoretical concept, but it is used by commercially available services which, given a real Database Management System (DBMS) application, search for the knob configuration optimizing its performance. Below, two state-of-the-art Bayesian-Optimization (BO) based algorithms for automatic DBMS tuning are discussed.

5.1 Gaussian Process Regression by OtterTune

OtterTune (Zhang et al., 2018) is a DBMS tuning service developed by the Carnegie Mellon Database Group. It can optimize several metrics, such as latency, running time, or throughput (which are described in Section 1), and its peculiarity is that it reuses data collected during previous tuning activities with similar DBMSs when looking for the optimal knob configuration of the target software system. The fact that the algorithm does not start from scratch when tuning a new DBMS application significantly reduces the time it takes to converge to a solution. Its main components are a *controller* (written in Java) and a *tuning manager* (written in Python).

5.1.1 Controller

The controller collects data from the target DBMS (e.g., performance metrics, knobs values, or target metrics), and sends this information to the tuning manager, which uses it to suggest the next configuration to try on the DBMS. Then, the controller sets the suggested values to the DBMS knobs, collects new data, and the cycle repeats until a stopping condition is met (stopping conditions are discussed in Section 3.6).

5.1.2 Tuning Manager

The tuning manager is responsible for the actual optimization of the objective. It starts by identifying the most relevant metrics for evaluating the improvement of new configurations and for characterizing the type of workload that the DBMS performs. Indeed, as stated in Section 4.2, the optimal knob configuration strongly depends on the kind of operations that the DBMS performs in a production environment.

Subsequently, it uses the Lasso technique as discussed in Section 3.3 to sort the knobs based on their relevance in affecting the objective, and only selects the d most impactful knobs to tune. OtterTune also has a black list of knobs that should not be changed (as outlined in Section 4.3), and each knob it chooses to tune must not be present in this list. Now that the system has good knowledge

of both the kind of workload of the DBMS application and of which knobs to modify, it identifies the instance within its data repository that best resembles the target DBMS. It does so by finding the optimization session within its repository the elements of which have the smallest Euclidean distance from the data of the target DBMS. Describing the way in which OtterTune encodes the type of DBMS and workload to later compute distance metrics between different tuning sessions is outside the scope of this thesis. Subsequently, as described is Sections 3.4 and 3.5, OtterTune fits a Gaussian Process Regression (GPR) model to the data for the target DBMS, along with the data from the most similar workload in its repository. It employs an Expected Improvement (EI) acquisition function (see Section 3.5.1) and, after having identified the knob configuration that maximizes this function, returns it to the controller along with the value of the EI function with such configuration (which is an estimate of the performance improvement in which this new configuration should result). The use of the GPR effectively lets the algorithm balance exploration and exploitation, leaning more towards the former in the beginning of the tuning process, and then gradually shifting towards the latter.

5.2 Contextual Gaussian Process Bandit Optimization by CGPTuner

Bandit algorithms (e.g., Bubeck and Cesa-Bianchi, 2012) are a class of decision-making strategies employed to solve problems involving sequential choices in the presence of uncertainty. Just like BO, they also try to balance exploration with exploitation, with the ultimate goal of maximizing the total reward over many rounds.

CGPTuner (Cereda et al., 2021) is a online learning (e.g., Hoi et al., 2021) technology designed for the automatic tuning of DBMSs that employs Contextual Gaussian Process Bandit Optimization (CGPBO) (Krause and Ong, 2011), which is a machine learning (ML) technique that combines bandit algorithms, GPs, and contextual information to optimize an objective function. Contextual information here refers to the workload, which means that a DBMS tuning CGPBO algorithm, when searching for the optimal knob

configuration, also considers the production workload of the target DBMS application. This method assumes that the workload to which a given DBMS application is exposed is not static, and so neither the optimal knob configuration is. Formally, we denote by $\boldsymbol{x}_t^* \in \mathbb{R}^d$ the optimal configuration at time t, and by $\boldsymbol{x}_t^+ \in \mathbb{R}^d$ the best configuration found so far at time t. They both depend on $\boldsymbol{w}_t \in \mathbb{R}^p$, which is a vector encoding the characteristics of the DBMS workload at time t. There are several methods to compute $\boldsymbol{w}_t$, such as the workload characterization algorithms used by OtterTune (as briefly mentioned in Section 5.1.2), or Peloton (Pavlo et al, 2017a). The use of CGPBO, which is essentially a contextual extension of BO, drastically reduces the need to rely on a knowledge base (although, as explained in Section 4.3, this need is always present).

The idea behind CGPTuner is to optimize several correlated functions $f_{\boldsymbol{w}}(\boldsymbol{x})$, where the data from a workload $\boldsymbol{w}$ can offer useful information about the behavior of the DBMS with another workload $\boldsymbol{w}'$. In fact, it is reasonable to assume that the performance of a certain configuration-workload pair is correlated with the performance of similar configurations and workloads. To capture this, CGPTuner employs the following kernel function

$$k((\boldsymbol{x}, \boldsymbol{w}), (\boldsymbol{x}', \boldsymbol{w}')) = g(\boldsymbol{x}, \boldsymbol{x}') + g(\boldsymbol{w}, \boldsymbol{w}') \tag{24}$$

where g is a Matérn 5/2 kernel (Shahriari et al., 2015), defined as

$$g(\boldsymbol{v}, \boldsymbol{v}') = \left(1 + \sqrt{5}r + \frac{5}{3}r^2\right) e^{-\sqrt{5}r} \tag{25}$$

where $r = \sqrt{\sum_{i=1}^{d} \frac{(v_i - v_i')^2}{l^2}}$ is the Euclidean distance between the two input vectors $\boldsymbol{v}$ and $\boldsymbol{v}'$, scaled by the length parameter l. The g function, with its moderate level of smoothness, is generally effective when the objective does not follow the very smooth patterns assumed by the Squared Exponential (SE) kernel described in Section 3.4.2. Calculating the overall kernel as the sum of the configuration kernel and the workload kernel implies that two points are considered similar if they have either a similar $\boldsymbol{x}$ or a similar $\boldsymbol{w}$ (or both).

The bandit part of the algorithm aids in selecting the acquisition function to use. Indeed, CGPTuner does not use the same acquisition function at every step, but instead employs

an online multi-armed bandit strategy (e.g., Roy, Thirmulai and Zurier, 2017) to select the best function from a portfolio of acquisition functions, something that has already been done in the DBMS tuning literature (Brochu, Hoffman and de Freitas, 2011). This allows the algorithm to consider multiple acquisition functions at each iteration, and progressively select the best one according to previous performance.

To obtain a representative GP prior distribution, it is common to standardize observed values. In the literature, this is typically done without considering the production workload, but CGPTuner instead does take it into account. Specifically, it employs a modification of the *Normalized Performance Improvement* (NPI) metric (Ashouri et al., 2016) defined as

$$NPI(\boldsymbol{x}, \boldsymbol{w}) = \frac{f(\boldsymbol{x}_0, \boldsymbol{w}) - f(\boldsymbol{x}, \boldsymbol{w})}{f(\boldsymbol{x}_0, \boldsymbol{w}) - f(\boldsymbol{x}^+_{\boldsymbol{w}}, \boldsymbol{w})} \tag{26}$$

Here, $\boldsymbol{x}$ is the configuration currently being evaluated, $\boldsymbol{x}_0$ is the baseline configuration (e.g., the default one, or the one before the automatic tuning process), $\boldsymbol{w}$ is the current production workload, $f(\boldsymbol{x}, \boldsymbol{w})$ is the value of the objective recorded with configuration $\boldsymbol{x}$ and workload $\boldsymbol{w}$, and $\boldsymbol{x}^+_{\boldsymbol{w}}$ is the best configuration identified so far under workload $\boldsymbol{w}$. Every time $\boldsymbol{x}^+_{\boldsymbol{w}}$ changes as the tuning progresses, past values are re-normalized. $NPI(\boldsymbol{x}, \boldsymbol{w})$ essentially computes the improvement of configuration $\boldsymbol{x}$ over the baseline $\boldsymbol{x}_0$ under workload $\boldsymbol{w}$. If $NPI(\boldsymbol{x}, \boldsymbol{w}) = 0$, then $\boldsymbol{x}$ brings no improvement over $\boldsymbol{x}_0$, if instead $NPI(\boldsymbol{x}, \boldsymbol{w}) = 1$, then $\boldsymbol{x} = \boldsymbol{x}^+_{\boldsymbol{w}}$, so the current configuration is the best one so far under the current workload.

CGPTuner combines all the elements described above to automatically tune DBMSs. Specifically, the algorithm starts the optimization process by collecting tuples of the form $(\boldsymbol{x}_i, \boldsymbol{w}_i, y_i)$ for $i = [1, ..., n]$ with random sampling (which, as seen in Section 3.2, could be improved by using Latin Hypercube Sampling (LHS)). This data forms the set $D = \{(\boldsymbol{x}_i, \boldsymbol{w}_i, y_i)\}_{i=1}^n$. Then, another loop starts, and at each iteration j the algorithm measures the current workload $\boldsymbol{w}_j$, and finds $\boldsymbol{x}^+_{\boldsymbol{w}_j}$ within its repository (which is the best configuration identified so far for workload $\boldsymbol{w}_j$). If $\boldsymbol{x}^+_{\boldsymbol{w}_j}$ has just changed from the previous iteration, CGPTuner re-normalized all the observed performance data as $NPI_j = \frac{f(\boldsymbol{x}_0, \boldsymbol{w}_j) - f(\boldsymbol{x}_j, \boldsymbol{w}_j)}{f(\boldsymbol{x}_0, \boldsymbol{w}_j) - f(\boldsymbol{x}^+_{\boldsymbol{w}_j}, \boldsymbol{w}_j)}$, and thus modifies D. This cycle repeats without end, so that CGPTuner keeps optimizing the system in an online way, to accommodate for the fact

that the production workload is subject to change. Clearly, it is always possible to modify the algorithm to include a stopping criterion, as discussed in Section 3.6.

Using online learning to tune DBMSs offers several advantages. First of all, since the behavior of the objective changes over time, as a result of changes in the workload, a tuning algorithm that can adapt to these changes without the need for a complete retraining of the model is beneficial. Another relevant benefit of online learning is that, since the DBMS can adjust its configurations in real-time based on the latest data, the performance tuning may be more responsive and efficient. Lastly, this method allows for continuous improvement of knob configurations. As more data is collected (which happens naturally in a production environment), the model can refine its predictions and potentially discover better configurations over time (Cai et al., 2022).

6 Future Work

The methods described in this thesis may benefit from some modifications. Some of them are discussed below.

6.1 Sparse Gaussian Processes

One of the main drawbacks of standard Gaussian Processes (GPs) is their intensive use of resources. Indeed, denoting with $n \in \mathbb{R}_+$ the number of training points, the time complexity for training a GP model is $O(n^3)$ (since it requires inverting the $n \times n$ covariance matrix, which is itself a $O(n^3)$ operation), while the space complexity is $O(n^2)$ (because of the storage of the $n \times n$ covariance matrix, that has n^2 elements). Sparse Gaussian Processes (SGPs) alleviate this problem by using a subset of the data, referred to as the set of *inducing points* or *pseudo-inputs*, to approximate the full GP model. Such points are generally selected randomly, and they are used to construct a low-rank approximation of the covariance matrix. This reduces the size of the matrix that needs to be inverted, decreasing the resources needed. When selecting $m \ll n$ inducing points, the training complexity is reduced to $O(mn^2)$ in terms of time and $O(mn)$ in terms of space (Hensman, Fusi and Lawrence, 2013). It goes without saying that the predictions made with a SGP will be approximations of what would be obtained with a complete GP.

6.2 Parallelized Optimization

As explained above, some Database Management System (DBMS) tuning sessions can be very long. Running such sessions in parallel on virtual machines might be beneficial, although particular attention is required to ensure the system does not behave unpredictably (Song and Jiang, 2023). Parallelizing a Bayesian Optimization (BO) algorithm requires running multiple evaluations of the objective function simultaneously, instead of sequentially. The idea is to first choose a parallelization strategy, meaning choosing whether to perform synchronous parallelism (which involves starting all parallel evaluations at the same time, and waiting for all evaluations to complete before updating the model and sampling again) or asynchronous parallelism (which instead involves starting an evaluation of $f_c(\boldsymbol{x})$ as soon as resources are available, and thus updating the model whenever an evaluation completes). The first approach is easier to implement but can be inefficient if the evaluation times vary significantly. In a parallel setting, the acquisition function has to be modified to take into account the fact that multiple points are being evaluated simultaneously. Such adjustments are outside the scope of this thesis.

6.3 Considering Hardware Features

The influence that knob configurations have on DBMS performance is significantly greater than the influence of the hardware used, especially in organizations that use relatively standardized machines across their offices. Nevertheless, some characteristics of the machine used can still be useful when attempting to predict the running time of a certain DBMS with a pre-determined knob configuration. The procedure described in this thesis takes into little consideration the hardware employed to run the database software systems (the only time it is considered is when setting the limits of the domain of certain knobs, as mentioned in Section 2). Therefore, including hardware features in models similar to the ones described in this work might offer some possibility of improvement.

6.4 Employing other Probabilistic Models

For several reasons, GPs are the most widely used probabilistic model in BO. Such reasons include their flexibility (as they do not require a fixed number of parameters), their

efficiency (as they offer a closed-form expression for the posterior distribution), and the fact that they also quantify the uncertainty of predictions, namely σ_*^2. Nevertheless, GPs are not the only possible choice when it comes to selecting which surrogate model to use. Some of the most popular alternatives are Random Forests, Bayesian Neural Networks, Treed Gaussian Processes, and Deep Gaussian Processes. Delving deeper into these alternatives is outside the scope of this work, but it is important to remember that GPs are not the only existing surrogate model for BO.

7 Conclusion

This thesis discussed the problem of optimizing the performance of a Database Management Systems (DBMS) by selecting optimal values for its configurable settings, called knobs. Such systems are therefore treated as functions, of which knobs are the inputs, and a performance metric (e.g., running time, latency, or throughput) is the output. An important feature of such functions is that they have a stochastic component, since the performance of a DBMS depends also on uncontrollable external factors (e.g., network traffic). This is an optimization problem, in which we look for the optimal input $\boldsymbol{x}^*$ which minimizes or maximizes the chosen performance metric. Such optimization is constrained by the fact that each knob assumes values in a specific range. The algorithms described here aim at solving the problem automatically, thus, once deployed, they require minimal human intervention. For several reasons, chiefly the fact that DBMSs are expensive-to-evaluate black-box functions, Bayesian Optimization (BO) is the chosen approach. Indeed, this method does not try to learn the objective function, but focuses directly on optimizing it. It constructs a probabilistic model of the objective and uses it to make intelligent decisions about where to evaluate next, balancing exploration of new areas with exploitation of known good regions. BO is an iterative process, which progressively updates the model with new observations. This tuning process is complicated by several challenges proper of this domain, such as the high resource requirements, cloud storage, and configurations causing errors. This thesis also described some state-of-the-art algorithms which use BO to automatically tune DBMSs, along with some ideas to further improve such commercially available methods.

References

Agrawal, S., Chaudhuri, S. and Narasayya, V. (2000). "Automated selection of materialized views and indexes in SQL databases". *Very Large Data Base Endowment*, vol. 2000, pp. 496-505

Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M. and Zhang, M. (2017). "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics". *14th USENIX Symposium on Networked Systems Design and Implementation*, pp. 469-482

Ament, S., Daulton, S., Eriksson, D., Balandat, M. and Bakshy, E. (2024). "Unexpected Improvements to Expected Improvement for Bayesian Optimization". *Advances in Neural Information Processing Systems*, vol. 36

Ashouri, A. H., Mariani, G., Palermo, G., Park, E., Cavazos, J. and Silvano, C. (2016). "COBAYN: Compiler Autotuning Framework Using Bayesian Networks". *ACM Transactions on Architecture and Code Optimization*, vol. 13(2), 1-25

Brochu, E., Hoffman, M. and de Freitas, N. (2011). "Portfolio Allocation for Bayesian Optimization". *Uncertainty in Artificial Intelligence*, pp. 327-336

Broyden, C. G. (1967). "Quasi-Newton methods and their application to function minimisation". *Mathematics of Computation*, vol. 21(99), pp. 368-381

Bubeck, S. and Cesa-Bianchi, N. (2012). "Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems". *Foundations and Trends in Machine Learning*, vol. 5(1)

Bureau of Labor Statistics (2023). "Database Administrators and Architects". *Occupational Outlook Handbook*, U.S. Department of Labor. Available at `https://www.bls.gov/ooh/computer-and-information-technology/database-administrators.htm`

Cai, B., Liu, Y., Zhang, C., Zhang, G., Zhou, K., Liu, L., Li, C., Cheng, B., Yang, J. and Xing, J. (2022). "HUNTER: An Online Cloud Database Hybrid Tuning System for Personalized Requirements". *Proceedings of the 2022 International Conference on Management of Data*, pp. 646-659

Cereda, S., Valladares, S., Cremonesi, P. and Doni, S. (2021). "CGPTuner: a contextual gaussian process bandit approach for the automatic tuning of IT configurations under varying workload conditions". *Proceedings of the VLDB Endowment*, vol. 14(8), pp. 1401-1413

Chaudhuri, S. and Narasayya, V. (2007). "Self-tuning database systems: a decade of progress". *Proceedings of the 33rd international conference on Very Large Data Bases*, pp. 3-14

Chen, Y., Bi, K., Wu, C. H., Ben-Arieh, D. and Sinha, A. (2021). "A Tutorial on Bayesian Optimization". *arXiv preprint arXiv:2108.02289*

Curino, C., Jones, E., Zhang, Y. and Madden, S. (2010). "Schism: a workload-driven approach to database replication and partitioning". *Very Large Data Base Endowment*,

vol. 3(1-2), pp. 48-57

De Ath, G., Everson, R., Rahat, A. and Fieldsend, J. (2019). "Greed Is Good: Exploration and Exploitation Trade-offs in Bayesian Optimisation". *ACM Transactions on Evolutionary Learning and Optimization*, vol. 1(1), pp. 1-22

Debnath, B., Lilja, D. and Mokbel, M. (2008). "SARD: A statistical approach for ranking database tuning parameters". *2008 IEEE 24th International Conference on Data Engineering Workshop*, pp. 11-18

Dias, K., Ramacher, M., Shaft, U., Venkataramani, V. and Wood, G. (2005). "Automatic Performance Diagnosis and Tuning in Oracle". *CIdR*, pp. 84-94

Duan, S., Thummala, V. and Babu, S. (2009). "Tuning Database Configuration Parameters with iTuned". *Proceedings of the VLDB Endowment*, vol. 2(1), pp. 1246-1257

Efron, B., Hastie, T., Johnstone, I. and Tibshirani, R. (2004). "Least angle regression". *The Annals of Statistics*, vol. 32(2), pp. 407-499

Gramacy, R. (2021) "Surrogates: Gaussian Process Modeling, Design, and Optimization for the Applied Sciences". United States: Chapman & Hall

Hensman, J., Fusi, N. and Lawrence, N. D. (2013). "Gaussian Processes for Big Data". *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence*, pp. 282-290

Hoi, S. C. H., Sahoo, D., Lu, J. and Zhao, P. (2021). "Online Learning: A Comprehensive Survey". *Neurocomputing*, vol. 459, pp. 249-289

Hsu, C. J., Nair, V., Freeh, V. W. and Menzies, T. (2018). "Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM". *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 660-670

Jones, D. R., Schonlau, M. and Welch, W. J. (1998). "Efficient Global Optimization of Expensive Black-Box Functions". *Journal of Global Optimization*, vol. 13, pp. 455-492

Kanellis, K., Alagappan, R. and Venkataraman, S. (2020). "Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-Selecting Important Knobs". *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*

Kanellis, K., Ding, C., Kroth, B., Müller, A., Curino, C. and Venkataraman, S. (2022). "LlamaTune: sample-efficient DBMS configuration tuning". *Proceedings of the VLDB Endowment*, vol. 15(11), pp. 2953-2965

Krause, A. and Ong, C. S. (2011). "Contextual Gaussian Process Bandit Optimization". *Advances in Neural Information Processing Systems*, vol. 24, pp. 2447-2455

Kunjir, M. and Babu, S. (2020). "Black or White? How to Develop an AutoTuner for Memory-based Analytics". *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 1667-1683

Kwan, E., Lightstone, S., Storm, A. and Wu, L. (2002). "Automatic configuration for IBM DB2 universal database". *Proceedings of the IBM Perf Technical Report*

Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998). "Gradient-based learning applied to document recognition". *Proceedings of the IEEE*, vol. 86(11), pp. 2278-2324

McKay, M. D., Beckman, R. J. and Conover, W. J. (1979). "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code". *Technometrics*, vol. 21(2), pp. 239-245

Mockus, J. (1989). "Bayesian Approach to Global Optimization: Theory and Applications". United States: Kluwer Academic

Montgomery., M. and Reif, D. (2018). "SQL Tuning Primer". Available at `https://github.com/BMDan/tuning-primer.sh`

Narayanan, D., Thereska, E. and Ailamaki, A. (2005). "Continuous Resource Monitoring for Self-Predicting DBMS". *MASCOTS*, pp. 239-248

Pavlo, A. (2022). "Configuring DBMS 'Knobs': 6 Ways to Avoid Surprises". *Database Trends and Application*, available at `https://www.dbta.com/Editorial/Trends-and-Applications/Configuring-DBMS-Knobs-6-Ways-to-Avoid-Surprises-150731.aspx`

Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T. C., Perron, M., Quah, I., Santurkar, S., Tomasic, A., Toor, S., Van Aken, D., Wang, Z., Wu, Y., Xian, R. and Zhang, T. (2017). "Self-Driving Database Management Systems". *Conference on Innovative Data Systems Research*, vol. 4

Pavlo, A., Butrovich, M., Joshi, A., Ma, L., Menon, P., Van Aken, D., Lee, L. and Salakhutdinov, R. (2017). "External vs. internal: an essay on machine learning agents for autonomous database management systems". *IEEE bulletin*, vol. 42(2), pp. 32-46

Picheny, V., Ginsbourger, D., Richet, Y. and Caplin, G. (2013). "Quantile-Based Optimization of Noisy Computer Experiments with Tunable Precision". *Technometrics*, vol. 55(1), pp. 2-13

Raihan, A. S., Khosravi, H., Das, S. and Ahmed, I. (2023). "Accelerating Material Discovery with a Threshold-Driven Hybrid Acquisition Policy-Based Bayesian Optimization" *arXiv preprint arXiv:2311.09591*

Rasmussen, C. E. (2003). "Gaussian Processes in Machine Learning". *Advanced Lectures on Machine Learning*, vol. 3176, pp. 63-71

Roy, U., Thirmulai, A. and Zurier, J. (2017). "Online Multi-Armed Bandit". *arXiv preprint arXiv:1707.04987*

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P. and de Freitas, N. (2015). "Taking the human out of the loop: A review of Bayesian optimization". *Proceedings of the IEEE*, vol. 104(1), pp. 148-175

Song, X. and Jiang, B. (2023). "Parallel Bayesian Optimization Using Satisficing Thompson Sampling for Time-Sensitive Black-Box Optimization". *arXiv preprint arXiv:2310.12526*

Srinivas, N., Krause, A., Kakade, S. and Seeger, M. (2010). "Gaussian process optimization in the bandit setting: No regret and experimental design". *ICML 2010 - Proceedings, 27th International Conference on Machine Learning*, pp. 1015-1022

Sullivan, D. G., Seltzer, M. I. and Pfeffer, A. (2004). "Using probabilistic reasoning to automate software tuning". *ACM SIGMETRICS Performance Evaluation Review*, vol. 32(1), pp. 404-405

Tibshirani, R. (1996). "Regression shrinkage and selection via the lasso". *Journal of the Royal Statistical Society*, vol. 58, pp. 267-288

Van Aken, D., Pavlo, A., Gordon, G. and Zhang, B. (2017). "Automatic database management system tuning through large-scale machine learning". *Proceedings of the 2017 ACM international conference on management of data*, pp. 1009-1024

Van Aken, D., Yang, D., Brillard, S., Fiorino, A., Zhang, B., Bilien, C. and Pavlo, A. (2021). "An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems". *Proceedings of the VLDB Endowment*, 14(7), pp. 1241-1253

Vasyliev, O. (2024). "PG Tune". Available at `https://github.com/le0pard/pgtune`

Venkat, N. (2018). "The Curse of Dimensionality: Inside Out". Available at `https://github.com/nmakes/curse-of-dimensionality`

Venktesh, V., Bindal, P. B., Singhal, D., Subramanyam, A. V. and Kumar, V. (2020). "OneStopTuner: An End to End Architecture for JVM Tuning of Spark Applications." *arXiv preprint arXiv:2009.06374*

Wang, X., Jin, Y., Schmitt, S. and Olhofer, M. (2023). "Recent Advances in Bayesian Optimization". *ACM Computing Surveys*, vol. 55(13), pp.1-36

Williams, C. K. I. (1998). "Prediction with Gaussian processes: From linear regression to linear prediction and beyond". *Learning in Graphical Models*, pp. 599-621

Zhang, B., Van Aken, D., Wang, J., Dai, T., Jiang, S., Lao, J., Sheng, S., Pavlo, A. and Gordon, G. J. (2018). "A Demonstration of the OtterTune Automatic Database Management System Tuning Service". *Proceedings of the VLDB Endowment*, vol. 11(12)

Zhang, X., Wu, H., Chang, Z., Jin, S., Tan, J., Li, F., Zhang, T. and Cui, B. (2021). "Restune: Resource oriented tuning boosted by meta-learning for cloud databases". *Proceedings of the 2021 international conference on management of data*, pp. 2102-2114

Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., Song, K. and Yang, Y. (2017). "Bestconfig: tapping the performance potential of systems via automatic configuration tuning". *Proceedings of the 2017 symposium on cloud computing*, pp. 338-350