MASTER'S THESIS 2022

# Efficient Optimization of Databases Using Parameter Importance Methods

Osama Eldawebi

ISSN 1650-2884
LU-CS-EX: 2022-15

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-15

# Efficient Optimization of Databases Using Parameter Importance Methods

Effektiv optimering av databaser med hjälp av parameter importance-metoder.

**Osama Eldawebi**

# Efficient Optimization of Databases Using Parameter Importance Methods

## (A study of parameter importance in order to increase efficiency in Bayesian optimization of database parameters to improve performance.)

Osama Eldawebi
osama.eldawebi@gmail.com

April 4, 2022

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Luigi Nardi, luigi.nardi@cs.lth.se

Examiner: Pierre Nugues, pierre.nugues@cs.lth.se

**Abstract**

Automated machine learning methods have seen increasing success in optimizing database configuration parameters for better performance. While these methods use techniques to efficiently explore and evaluate possible combinations of parameter values, running them may use large computational resources. As database systems have tens of configurable parameters, this master thesis explores the possibility of reducing the number of parameters to optimize by identifying what parameters are most important. To do so, we applied different parameter importance methods on parameters from the database systems RocksDB and PostgreSQL. We combined the methods by introducing a new ensemble method to reach a final set of most important parameters. Using this set, we then compare the performance gains obtained from optimization and discuss whether reducing the set of parameters is a worthwhile effort despite the possibility of achieving lower performance. After setting a threshold for what importance value a parameter must cross to still be considered, the results for RocksDB show that we can match the performance gain by 60-70%, suggesting that the threshold may have been set too high. For PostgreSQL we find that we need more repetitions to come to a similar conclusion. In addition, we observe that the results from applying the parameter importance methods sometimes disagree and the type of workload being used is crucial.

**Keywords**: databases, SQL, NoSQL, graph databases, optimization, machine learning, feature importance

# Acknowledgements

Due to a variety of issues faced, this master thesis took longer than expected to complete. However many obstacles I faced, I am grateful for my supervisor Luigi Nardi (professor at the CS department) who had the patience and dedication to help me move forward with my work. I express my deepest gratitude to Luigi and his team at DBtune, who have taught me a lot about the field of database parameter optimization and the opportunities that this type of work holds for the future. The scope of this master thesis took a lot of twists and turns, but I am pleased to have arrived at the state it is currently at.

The optimization procedures in this thesis were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC), partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

# Contents


**1 Introduction** **7**
1.1 Domain background . . . . . 7
1.2 Database systems in study . . . . . 9
1.2.1 SQL databases (PostgreSQL) . . . . . 9
1.2.2 NoSQL databases (RocksDB) . . . . . 9
1.2.3 Graph databases (Neo4j) . . . . . 10
1.3 Contributions . . . . . 10
1.4 Related work . . . . . 11

**2 Background** **13**
2.1 Database Optimization . . . . . 13
2.1.1 Optimizer . . . . . 14
2.2 Benchmarking . . . . . 14

**3 Approach** **17**
3.1 Parameter importance . . . . . 17
3.1.1 Background . . . . . 17
3.1.2 Parameter importance using Random Forests . . . . . 18
3.1.3 Parameter importance using CAVE . . . . . 21
3.1.4 Parameters considered . . . . . 26

**4 Evaluation** **31**
4.1 Setup . . . . . 31
4.2 Parameter Importance Study . . . . . 32
4.2.1 Test functions: Branin and Rosenbrock . . . . . 32
4.2.2 RocksDB . . . . . 35
4.2.3 PostgreSQL . . . . . 45
4.2.4 Neo4j . . . . . 51
4.3 Summary . . . . . 52

**5 Conclusion** **55**

# Chapter 1
# Introduction

In this chapter we present the research domain of applying automated machine learning to optimize database systems as well as provide the relevant background to the thesis context.

## 1.1 Domain background

Databases are essential for long term storing of data and processing queries received from software programs interacting with the data. A database is commonly referred to as a *DBMS* (Database Management System) because a database typical comes with software that allows end users to efficiently interact with the database. Examples of what a DBMS can offer is software for defining different ways of filtering the data, allow retrieval of data in different formats, and enforce security rules on the user interacting with the system. In general, the way a database is designed to store data and process queries is vital to the application area in which a database will succeed in being widely adopted. This has made the market offering of database technologies diverse with some being better suited for specific applications than others. For example a database system can be designed for storing data that is frequently accessed or for data that acts as a backup state.

Many DBMS allow database administrators (DBA) to change the values of some database configuration parameters. These can be diverse and related to things like caching, memory allocation, read/write concurrency, compression algorithms, etc. The role of DBAs (henceforth called users) includes being knowledgeable about these parameters and configuring the right values so that the DBMS performs optimally. However, tuning configuration parameters can be a difficult task when the number of parameters is large. What values a parameter can hold are also important; if a parameter is continuous within some range, it means in theory it can hold an infinite number of possible values. Many of the popular DBMS expose tens if not hundreds of parameters to users that can feel quite daunting if one is not too familiar with the specific DBMS they are using. Not to mention, hiring database analysts to monitor and tune a DBMS can be quite expensive. Therefore the goal in this research domain is to meet

the growing incentive to apply automated machine learning methods in database systems so that that they can attain better performance with as little resources as possible. When we say database performance, we could be referring to different metrics. The performance of a DBMS is often measured either by the number of transactions a DBMS can process per second called the *throughput*, or by the average time it takes to process a transaction called the *latency*. In general, the most relevant metric is dependent on the use case of the DBMS so there could naturally be others that are more applicable such as energy consumption.

Database configuration parameters are often called *knobs* in the literature, and in practice we often want to limit the number of knobs we use during our automatic tuning process. The number of knobs and the values they can hold can quickly create a very large parameter space that would constitute the search space explored by a machine learning method. To create a good initial set of parameters to tune, expert knowledge is often exploited as much as it is available to choose parameters that we feel confident as having an impact on the performance of a DBMS. This can be done by consulting with DBAs, reading online documentation for the DBMS, or learning from the tuning experiences of other users in articles or blogs. In the case of very limited information the user will just have to use intuition and use trial-and-error to monitor the outcome of changing the values of some parameters here and there. Thus the constriction of the space as much as possible helps the machine learning method more efficiently explore it and converge faster.

Once we have set up a parameter space, we can apply an optimization method to efficiently explore the span of possible parameter configurations. Each configuration of parameters in the search space is called a *sample* that the optimization method evaluates at every iteration. The sample is evaluated by measuring the resulting performance it caused in the database. We typically perform automatic tuning of parameters using established optimization frameworks (called *optimizers)* like HyperMapper [25], Hyperopt [5] and Optuna [1] that include options for users to customize their optimization procedure. The optimization methods that different frameworks choose to employ are diverse, but one particular successful method is called Bayesian optimization (BO). More details on the choice of optimization method is provided in section 2.1.1.

Evaluating a database configuration means we need to measure the resulting performance using a workload containing a specific set of queries (more on this in section 2.2). Since a workload may have many thousands if not millions of queries depending on how we want to test the system, optimizing databases is generally seen as an expensive process. Thus we want to establish an optimization procedure that efficiently explores different configurations to improve the database performance. In this thesis, we continue in this direction but instead of looking at different optimization methods and how we can customize them better, we analyze the parameter space itself in order to settle with a set of parameters that we feel has the most impact on the performance. Reducing the dimensionality of the problem to some $k$ most important parameters and still achieve a relatively high improvement in performance over default would be a huge win. Hence we will explore ways in quantifying the importance of a parameter so that in the future we can start optimizing DBMS more effectively.

## 1.2 Database systems in study

We explore different parameter importance methods in a variety of database systems in order to compare the systems and gather insights on their similarities and differences. Specifically, we aim to compare different categories of databases knowing that there have been several that have garnered a lot of attention. In this thesis we investigate three different categories of databases: SQL (or 'relational'), NoSQL (non-relational), and graph. Although graph databases belong in the NoSQL category, they are different enough that we decided to describe them separately.

### 1.2.1 SQL databases (PostgreSQL)

SQL databases most commonly utilize a relational model for structured data stored as a collection of tables. The relationships between data are thus already pre-defined so that references between different tables are established from the beginning. SQL has been a popular standardized language since the 1970s which eases migration between different database technologies. However, different vendors can still have their own features and compatibility measures that makes SQL code in two different databases look slightly different. Many SQL databases today are utilized through licensed DBMS such as MySQL and Oracle, with the exception of PostgreSQL, which has seen tremendous success as an open source DBMS. PostgreSQL has been developed for over 30 years and has a reputation for being very fast [14]. The DBMS also have a large number of knobs that the user can configure, with the paper by Kanellis et al. [19] from 2020 stating that there are about 170 knobs.

### 1.2.2 NoSQL databases (RocksDB)

As the name suggests, NoSQL databases do not conform to the standard relational model with SQL, which means they provide different ways of storing and interacting with data. NoSQL databases are diverse, with the main types being document, key-value, wide-column and graph databases. They rose in popularity in the late 2000s, where focus started being placed on creating effective ways on managing flexible data models and increase developer productivity [24].

#### RocksDB

As an embedded key-value storage system, RocksDB has seen wide success in being used to support large scale systems at companies like LinkedIn, Yahoo and Pinterest [33]. It was developed by Facebook and optimized for high speed disk drives (e.g. SSDs). RocksDB has three basic structural components: The *memtable*, *sst files* and *log files* [31]. The memtable is an in-memory data structure that fills up with data from new write operations that in turn optionally fill up the log files. When the table gets full, the stored data is flushed into one or more SST files (essentially the basic storage uniets). These files are stored in different layers and events, called *compactions*, are triggered regularly in order clean data in the layers by removing duplicates and overwritten keys [31]. As a strong representative of an embedded

NoSQL database, we study RocksDB to learn more about how its knobs contribute to the DBMS performance.

### 1.2.3 Graph databases (Neo4j)

Graph databases, or Graph Database Management Systems (GDBMS), are themselves NoSQL systems because they do not rely on SQL, but instead process queries in a language that is written specifically for graphical relationships. Some of the currently most popular graph query languages are Gremlin, Cypher, and GSQL which are used by Azure Cosmos DB, Neo4j and TigerGraph respectively. An initiative to create a universal language has been proposed in 2019 called GQL (Graph Query Language) by the widely known ISO organization that is currently in a preparatory development phase [18].

Graph databases aim to more naturally capture the relationships between data in comparison to traditional relational databases. Their adoption has become widespread as social networks and user interactions have increased due to people becoming more actively engaged online. The database parameter tuning literature has shown successful experiments at making traditional database management systems like PostgreSQL [19], MySQL, MongoDB [39], RocksDB [2] and FoundationDB [34] increase their performance but as of yet there has not been any similar efforts for graph databases. It would be interesting to explore optimizing graph databases as they provide a unique set of challenges to this domain because both storing and querying data is done in terms of the relationships that exist in the data. Neo4j, a company with its own platform that is widely regarded as the largest GDBMS, defines a graph database as a database that is designed to treat these relationships equally important to the data itself [29].

In this thesis we will study Neo4j as a GDBMS, which is also known to be user-friendly with a large community support. It exposes a large number of tunable configuration parameters and there are even guides on performance tuning on their website that allows us to delve quickly into the Neo4j ecosystem. We will attempt to optimize Neo4j as a GDBMS and learn more about how its configuration parameters contribute to the performance.

## 1.3 Contributions

The optimization data obtained by applying an optimization framework on different databases allows us to analyze it more closely. In this thesis we look at applying parameter importance methods to obtain more information about what database knobs are most important with some consideration for the type of queries that were executed. This allows us to refine our initial search space in a database optimization problem by identifying a selection of important parameters that we can dedicate all of our resources into efficiently tuning. While it may be that the search space restriction leads to a worse performance improvement than if we were to use the entire initial space, it can still be more advantageous if the result is that we managed to greatly reduce the amount of resources spent and still obtain a 'good enough' improvement. Using different parameter importance methods to help explain the impact of database knobs on the performance is the basis of our contribution to the field. More specifically, we answer the following questions:

- What do different parameter importance methods tell us about what parameters are most important?
- Is focusing only on the identified most important parameters a worthwhile effort to pursue?
- Are the results consistent across different database workloads?

By tackling these questions, we aim to open up for greater opportunities in the research area of database parameter tuning using automated methods.

## 1.4 Related work

The most relevant work that has explored the importance of database knobs is a paper by Kanellis et al. [19]. The authors investigate if a reduced set of parameters can still achieve good performance by looking at two databases: Cassandra and PostgreSQL. The reduced set contains some top most $k$ parameters and they explore this for two different workloads in the YCSB benchmarking tool. In our thesis we follow a similar strategy to find out what parameters are most important. The authors first use an optimizer to tune databases with a large number of parameters. The optimization data is then analyzed to quantify the importance of each parameter. Finally, they choose the most important ones they found and perform the tuning process again and compare the performance improvement gained from this reduced set to the one using the initial set. The authors show that just by tuning the five most important parameters in both Cassandra and PostgreSQL it is possible to match the performance gained at over 99% by tuning an initial set of 30 parameters. They also highlight that the two workloads they look at show very similar results for which parameters are most important.

The results from Kanellis et al. [19] may tell us that by only tuning a few parameters we perhaps do not need to care about the type of workload. While the strategy is similar, in our thesis we aim to focus more on the method we use to quantify the importance of a parameter. The authors determine importance values using a random forest and while they do not provide details on how the calculations are performed, we believe it is similar or the same as we describe it later on in section 3.1.2. Our work extends this by providing a more detailed discussion around what method one should use to calculate parameter importance values by investigating three additional methods from a framework called CAVE (more on this in section 3.1.3).

There are others that have quantified parameter importance to optimize a database with a good set of initial knobs that seem relevant. A paper by Mahgoub et al. [22] investigates optimizing Cassandra for high performance computing (HPC) workloads using an optimization framework called Rafiki. The authors utilize a parameter importance method called ANOVA to identify five key parameters to optimize. In ANOVA, every parameter is analyzed once at a time and a variance is calculated based on the database throughput obtained after only changing the value of the parameter itself. The five key parameters are thus the ones with largest impact on the performance, which corresponds to the largest variances.

Another paper by Schmied et al. [34] used a sampling method to gather 200 samples and train a random forest to provide the 10 most important parameters that they then use

to tune a DBMS called FoundationDB. Again, the details are not provided in the paper but we assume the approach using random forests is similar or the same as we later explain in section 3.1.2. This thesis complements the prevalence of random forests being used to create importance rankings by exploring other methods that measure parameter importance in a different way, for example with a more local or global perspective.

# Chapter 2
# Background

## 2.1 Database Optimization

A popular optimization method to evaluate costly black box problems is widely considered to be Bayesian optimization (BO). As Shahriari et al. [35] point out in their extensive review of the method, BO is efficient when we deem our objective function (or black box function) expensive to evaluate due to lack of derivatives and possibly being multimodal. BO is described as a sequential model-based method due to it involving establishing a prior belief state over how the objective function looks like and then iteratively refining that belief and decreasing the uncertainties in it. These updates of the prior create a posterior that represents our updated beliefs based on our observed data on how the objective function looks like. The model that holds this posterior is called the *surrogate model*, and together with an acquisition function are the two parts that are central to BO.

The acquisition function in BO uses the uncertainties we have in the posterior to guide the BO method on what sample to evaluate in the next iteration. As described in [35], there are many acquisition functions whose purpose is essentially to create a trade-off between exploitation and exploration. The idea is that sometimes we try to exploit samples we found that were good by looking for others that are close by in the search space and sometimes we want to look further away in hopes of finding some other optima.

The surrogate model that is used in BO is often a *Random Forest* (RF). Random Forests are considered an ensemble method in machine learning since they are collections of individual decision trees and predictions are made by aggregating their every tree's contribution. Due to being based on decision trees, they are more intuitive to interpret and debug in comparison to for example neural networks. Random forests can also easily handle categorical parameters according to [35] and [25], which are a type of parameters that we find frequently in database systems.

In some recent efforts, we have also seen some combinations of deep learning and reinforcement learning being used to optimize databases. CDBTune$^{+}$ is a framework by Zhang et

al. [39] that shows high efficiency during 'online' tuning, the process by which a user connects their database system to the framework and tunes it. The drawbacks of this approach is that there is considerable 'offline' model training done prior to creating the online tuning service. The offline model is an established knowledge base that the online tuning process then exploits to make it faster. This also means that the tuning service is restricted to database systems that have been used prior for offline learning. In our case we want to look at a variety of DBMS without gathering the resources to leverage a large offline model. Hence we deem BO to make most sense as an optimization method to use for optimizing our costly objectives functions. Evaluating a single sample through a benchmark takes in our case at least 10 minutes.

### 2.1.1 Optimizer

Each database in the study is optimized using an optimization service called DBtune [11], that itself is based on the open-source optimization framework called HyperMapper [25]. The framework utilizes BO with a random forest as its surrogate model and runs an optimization procedure in two phases: (1) a warm-up phase, followed by (2) a main phase. The aim of the warm-up phase is to build an initial belief system in the surrogate model based on a few samples from different parts of the search space, which is the set of all samples that can be made from combinations of the input parameters values. The sampling method used in the warm-up phase is thus something like random sampling to obtain a diverse set of samples. In the main phase, the surrogate model (also called the *predictor)* continues to be iteratively trained from drawing a sample and evaluating it [25]. The idea is that at each iteration the predictor is used to choose the features/parameters that are most important for the next iteration while at the same time utilizing an 'exploration and exploitation' scheme. Essentially it 'guides' the optimization on what parts of the search space should be exploited more with occasional evaluations of samples further away.

DBtune contains many adjustable parameters, such as the sampling method in the warm-up phase as well as the number of trees that are used in the random forest. Due to time constraints, this thesis will leave most parameter values as their default. This means for example that a random forest will be used with the number of trees being ten and the minimum number of samples in a leaf node being five [17].

## 2.2 Benchmarking

Evaluating a sample consisting of parameter values for a specific database to measure the resulting performance is done by executing a benchmark. A benchmark is a tool that is able to interface with a database and collect performance metrics for how well it processes a set of pre-defined queries. Benchmarking tools for DBMS are numerous, with some being targeted for specific query languages and databases while others can be used for multiple different systems. Database benchmarking tools started being developed many decades ago with the foundation of the Transaction Processing Performance Council (TPC) initiative. TPC was first introduced during the 1980s in response to many companies claiming that they offered the best Online Transaction Processing (OLTP) systems [20]. TPC has evolved over time from creating a benchmarking standard for evaluating a system's capability of handling

online transactions to incorporate a much larger collection of benchmarks for different use cases like TPC-C and TPC-H.

There is no universal benchmarking tool in the database tuning literature that can be used for any arbitrary DBMS. Some database providers like RocksDB offer their own internal benchmarking tool called *db_bench* [32], which makes it easily accessible for existing database users. SQL-based DBMS often use *OLTP-Bench* [12] for benchmarking while NoSQL databases are often benchmarked with *YCSB* [9]. For graph databases there has been an initiative by *LDBC* (Linked Data Benchmark Council) [21] to create a benchmark called *SNB* (Social Network Benchmark), a fairly recent and growing effort to create a robust standard benchmarking tool for evaluating data systems with graphical relationships. In our thesis, we use *db_bench* for RocksDB, *OLTP-Bench* for PostgreSQL, and *LDBC SNB* for Neo4j as tools to evaluate different parameter configurations. More information on each tool and how we use it is provided below.

### db_bench

The internal tool in RocksDB supports many options to generate different types of workloads. It also provides detailed output results, with both overall metrics for the database performance but also for individual components of the database (e.g. each layer of SST files). For this thesis we use the 'readrandomwriterandom' workload type, which performs random reads and writes over time. We can specify the ratio of read to write queries to perform, which means we can compare read-heavy to write-heavy workloads.

### OLTP-Bench

OLTP-Bench is a testbed environment created by Difallah et al. [12] to make benchmarking relational databases easier. It reduces the engineering effort required to set up a benchmarking environment customized for a specific use case. In this tool, we can specify what workload we want to run depending on the DBMS at hand. The tool itself consists of popular workloads from benchmarks like TPC-C, LinkBench, and even YCSB, whose properties can be set up using a configuration file. We use this tool in this thesis to benchmark PostgreSQL.

### LDBC SNB

From the specification [10], the goal of the benchmark LDBC SNB is to define a framework where different graph based technologies can be fairly tested and compared. The benchmark offers two workloads: the *Interactive Workload* with user-centric transaction queries and *the Business Intelligence Workload* with analytic queries. We use the former workload as it is closest to what users may have in their real time graph databases. LDBC SNB is intended to be used by a variety of people: from researchers to graph database companies to end users trying to compare different GDBMS. The benchmark also intends to support many DBMS, both relational and non-relational. The benchmark has a number of queries that perform specific tasks, where some are easy and some complex. In configuring the benchmark, we can specify the ratios between the queries we want to execute and how often we want to perform a write operation between series of reads. In the context of GDBMS, to our knowledge no other benchmark has been developed to the level of sophistication that we see in LDBC SNB.

Therefore we use this benchmark in our study to evaluate the parameter configurations in Neo4j.

An optimization procedure for a specific problem like optimizing a database to maximize throughput consists of many intricate details that can be further explored. Firstly, we discussed the diverse ways we could design an optimization problem such as by choosing an appropriate surrogate model. We also need to think about the validity of the samples we explore during optimization by choosing an appropriate benchmarking tool. In this thesis we have not meddled much with the options for the optimization algorithm that is in HyperMapper (the optimizer we use) due to time constraints, but we have spent time creating an optimization environment that makes sense given the benchmarking tools we have presented and used. More details on the benchmarking setup are later described in section 4.1.

# Chapter 3
# Approach

## 3.1 Parameter importance

### 3.1.1 Background

When optimizing the performance of a database as a function of its parameters, the user may be interested in how each parameter contributes to the performance. Deriving some sort of ranking system for the parameters involved is useful for limiting the parameters involved in the optimization algorithm to the most important parameters. This reduces the complexity of the black-box problem by reducing the number of dimensions in the sample space created by including all initial interesting database parameters. This allows an optimization algorithm to focus on parts of the sample space that matter more towards the measured performance and at the same time converge faster to an optimum.

A parameter ranking system from which a user can then restrict the sample space is subject to a possible trade-off during optimization. The smaller space that is likely to lead to a higher convergence rate and more focused sampling does not guarantee that the best performing configuration lies in this new restricted space. It may be the case that the best performing configuration involves changing values of the parameters that were left out. Given this trade-off, in what kind of optimization problems is this risk still acceptable? It depends on the user's goal and the amount of resources that are available. The user may not have enough time or computing capacity to be able to run an optimizer for very many iterations. In that case the user may be open to restricting the sample space if it would mean that the optimizer would at least obtain some satisfactory level of performance improvement over default.

In essence, a parameter importance ranking helps a user reduce the complexity of the optimization search space to match resource availability. The problem we need to note here, however, is that to create parameter rankings we would need to explore the entire search space first. This might seem counter-intuitive to the argument above stating that the user

may have little resources available to begin with, but here we only need to do this once and then we can use the smaller space that we create to save resources in future optimization runs. We can also think of a situation where a person creates an open tuning system with many users and the person (system owner) runs an optimization process to identify the most important parameters. The users of the system can then use these results, which may be applicable to them if they have low amounts of resources. We also have to note that for the parameter importance results to be applicable in future optimization runs, we make an assumption regarding the database workload. Firstly, database workload should not be drastically different. If the types of requests a DBMS handles change remarkably then it might cause different parameters to be more important and thus result in a different ranking system. Therefore ideally we want to perform the optimization runs for a diverse set of workloads and observe their differences. If the workloads used lead to very similar rankings, then the importance of a parameter is likely to be workload-independent. If the rankings are different, then the results would be only interesting to a user that can expect a consistent workload over time.

In the paper by Kanellis et al. [19], the parameter rankings the authors identify are almost identical for the DBMS Cassandra but much more different for PostgreSQL. In the paper explaining OtterTune [38], a system for automatically tuning database systems, the authors report that three different workloads showed drastically different latency measurements for three configurations in a popular DBMS called MySQL. The authors argue that configurations are non-reusable due to the influence of the workload type ('application') on the database performance. Hence we have to be mindful of these when making conclusions about the generalization of parameter importance rankings we obtain.

In this thesis we identify parameter rankings by utilizing different parameter importance methods. As discussed above, to calculate a ranking it would require that all available and relevant parameters be explored first, which means that in practice we have to optimize our black box function with an initial set parameters that may be large. It is only after we run this once and analyze the results that we then decrease the set of parameters (similar to what was done in [19]). We then compare the improvement gained from optimization using the smaller set to the initial one and discuss the efficiency of using parameter importance rankings to only tune the most important parameters. The reason as to why we perform optimization procedures and do not use just take random samples from the search space (i.e. random sampling) to learn about parameter importance is because want to learn about the contributions of the parameters to the performance metric in areas of the search space where optimization prioritizes. Random sampling would mean drawing random samples across the entire search space, which in order for this to be feasible we would also have to evaluate very many samples depending on the search space complexity (i.e. this leads to longer experiments). A key feature in optimization is to more efficiently navigate a search space and our goal with parameter importance in this thesis is to investigate if the optimization procedure itself can be made simpler by focusing on fewer parameters.

### 3.1.2 Parameter importance using Random Forests

Due to being a collection of individual decision trees, random forests manage to account for less overfitting than what commonly occurs in a regular decision tree. Each decision tree in a random forest model contains a bootstrapped dataset (a randomly sampled copy of the original data with replacement) as well as a random subset of the features/parameters in

the data. A prediction is made by aggregating the predictions from the individual trees and choosing the result that is most popular. Random forests can be used for both classification and regression problems but the implementation of the model will vary slightly depending on which is used. This is also reflected in how we calculate parameter importance rankings using random forests, as we shall see soon.

Third party libraries for implementations of a random forest often come with their own method for calculating the importance of each parameter in a trained instance of the model. In DBtune, the random forest is an instantiation of the model provided by the popular library called *scikit-learn*. It has a method for determining the feature importance values based on the concept of Gini impurity. Gini impurity is a popular method to quantify the quality of a tree node after a split. It is worth noting that the Gini impurity is specifically used during classification (i.e., the random forest model would be a classifier). This is connected to the criterion method of the random forest model, which is how a tree in the model calculates the quality of a node split during training.

In the *scikit-learn* random forest model, the default criterion for classification is the Gini impurity, while for regression it is the squared error. For black-box optimization problems like database parameter tuning in which DBtune is used, we are dealing with a regression problem (i.e., predicting the database performance) and use the random forest as a regressor. Hence the quality of node splits in the random forest trees are based on the the squared error (which is better when lower). Whether we use classification or regression, we still follow a similar concept of determining how to best split nodes in the random forest trees. To understand how parameter importance is calculated, we will first illustrate how decision trees splits its nodes and then show the actual calculations that are made. The concept is more easily illustrated for a classification problem so we will do that and discuss where things would be different if we use regression like in the context of our thesis experiments.

During training, every decision tree in a random forest (henceforth referred to as RF) is constructed in a top-down manner by deciding what features best split the data at every level. Every node answers a binary question in deciding how to split the data. After testing every feature, we calculate the quality of the split by quantifying the impurity as previously mentioned The feature that leads to the least impurity is chosen and we split the data accordingly. Then we move on lower to the next level with the remaining features and repeat. If there are no features left, or if we specify that the number of remaining samples in the data has to be higher than a specific threshold, we label that node as a leaf node (i.e., a node with no outgoing branches). Given a node with the binary classes $x$ and $y$, the impurity of a node $n$ using the Gini impurity is calculated using the following formula:

$$\text{impurity}(n) = 1 - (\frac{\#x}{\#x + \#y})^2 - (\frac{\#y}{\#x + \#y})^2 \tag{3.1}$$

where $\#x$ and $\#y$ are the number of samples in the two classes.

As an example, consider the decision tree example in figure 3.1. The impurity of a node, say the leftmost leaf node (green) would be: $1 - (\frac{4}{4+0})^2 - (\frac{0}{4+0})^2 = 0$, which makes sense since all the samples belong in one class, making it pure with 0 as an impurity value. For regression, the leaf nodes do not contain binary values for the target variable, where the value of a leaf node is given by the modal class that is present. Instead, the value of a node is the average of the target variable values for all samples in the node. Then to calculate the impurity value we use the squared error, where we take we the sum of squares of the difference between every

observed target variable value and the mean value. The higher this difference is, the more impure the node.

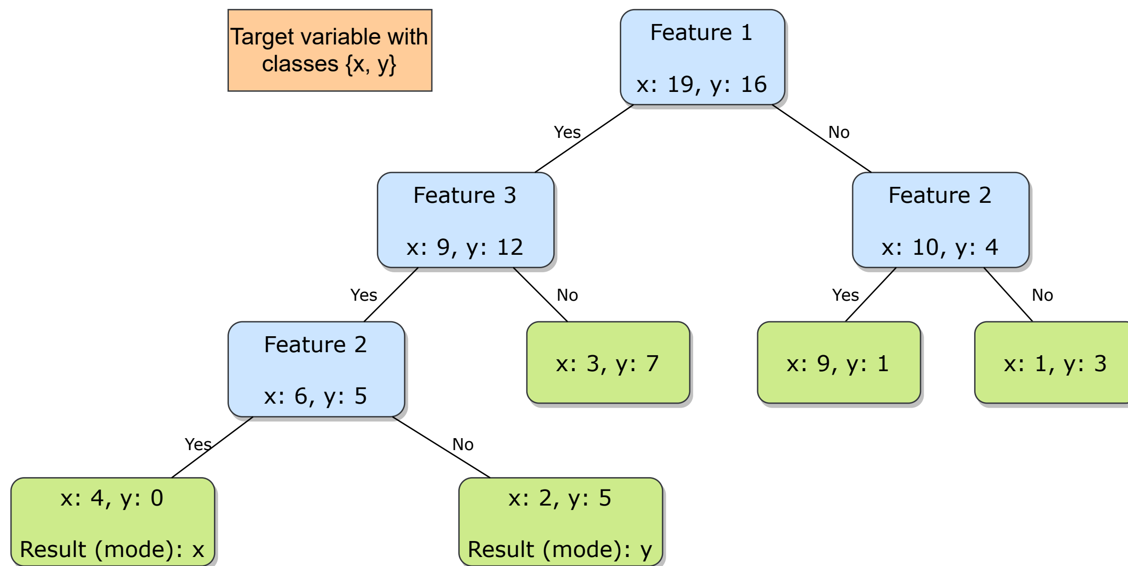



**Figure 3.1:** Example of a decision tree with four features and a target variable with classes $x$ and $y$. The decision in each blue node (or box) that is portraying the situation in which we are answering a specific question relating to the feature with either 'yes' or 'no'.

Computing parameter importance values follows the same concept. After training a random forest, we calculate parameter importance values by analyzing each individual model tree, then average across all trees. When inspecting a tree, we look at every feature node and calculate the quality of the split it creates by analyzing the node itself and its resulting child nodes. For a given feature (parameter) $x$, its importance $i$ in a tree is given by the following formula:

$$\text{imp}(x) = w_x e_x - w_{L(x)} e_{L(x)} - w_{R(x)} e_{R(x)} \tag{3.2}$$

where $w_x$ is the weighted number of samples reaching the node (= samples in node divided by the total number of samples in the tree) that is responsible for splitting itself by $x$, $e_x$ in our regression context is the squared error (Gini impurity in classification) of that node, and the negative terms are similar but correspond to the left $L(x)$ and right $R(x)$ child nodes after the feature split. The weights added to each error term makes sense when you think about how they would be larger (due to more samples) the higher up the feature is located in the tree. After all, being placed higher corresponds to being the better splitting feature during training.

Note that a feature may be responsible for multiple splits in a given decision tree. Looking at the example tree in figure 3.1, we see that 'Feature 2' causes two splits. Thus following the calculation above, we would obtain two importance values. The resulting importance for 'Feature 2' would be the sum of these. Essentially the more a feature plays a role in node splitting as well as the higher it is placed, the higher its evaluated importance will be. For 'Feature 2' in the example tree, the resulting importance value is the sum of two contributions $C_1$ and $C_2$ it has in the tree, given we are using the Gini impurity since we are dealing with

a classification tree:

$$\begin{aligned} C_1 = & \frac{6+5}{19+16} \cdot \left(1 - (\frac{5}{6+5})^2 - (\frac{6}{6+5})^2\right) \\ & - \frac{4+0}{19+16} \cdot \left(1 - (\frac{4}{4+0})^2 - (\frac{0}{4+0})^2\right) \\ & - \frac{2+5}{19+16} \cdot \left(1 - (\frac{2}{2+5})^2 - (\frac{5}{2+5})^2\right) \\ & = 0.0742 \end{aligned} \tag{3.3}$$

$$\begin{aligned} C_2 = & \frac{10+4}{19+16} \cdot \left(1 - (\frac{10}{10+4})^2 - (\frac{4}{10+4})^2\right) \\ & - \frac{9+1}{19+16} \cdot \left(1 - (\frac{9}{9+1})^2 - (\frac{1}{9+1})^2\right) \\ & - \frac{1+3}{19+16} \cdot \left(1 - (\frac{1}{1+3})^2 - (\frac{3}{1+3})^2\right) \\ & = 0.0690 \end{aligned} \tag{3.4}$$

$$\text{imp(Feature 2)} = C_1 + C_2 = 0.0742 + 0.0690 = 0.1432 \tag{3.5}$$

Once we repeat the procedure for the other features, we then normalize the importance values to be between 0 and 1. Finally, we compute average values across all decision trees in the model. One thing to note is that since the trees are bootstrapped, a feature that is missing in a tree is simply assigned a value of zero.

### 3.1.3 Parameter importance using CAVE

An open source framework called CAVE (Configuration Visualization, Assessment and Evaluation) provides multiple parameter importance methods based on previously obtained optimization data [6]. The framework was developed by a research group called AutoML at the universities of Freiburg and Hannover with the goal of expanding the area of machine learning automation [4]. In pursuit of this goal, CAVE allows users to generate detailed insights into the obtained data and produce informative plots. The framework is comprised of different categories of methods that analyze an algorithm's inputs and outputs. One such category is parameter importance, which is what we are interested in for our use case. Nonetheless, the methods are defined in the context of being part of a larger framework that target a general optimizer that they refer to as an Algorithm Configurator (AC). The name originates from a paper by one of the CAVE authors describing an algorithm configuration framework called ParamILS [16], where it is written that *algorithm configuration* is an alternative term to the commonly known *parameter tuning*. They explain that they favor this term due to the fact that they are interested in methods that can deal with a potentially large number of parameters, each of which can be numerical, ordinal or categorical.

To use the methods we are interested in we must first understand some of that context. The research group behind CAVE has developed many tools and ideas for algorithm optimization over the past decade that build on top of each other and include mutual naming conventions. To understand the context behind CAVE, there are specifically two main ideas that should be outlined (1) how an algorithm configurator is defined and (2) how CAVE can

say something about configurations that have not been previously evaluated, i.e. missing from the user's obtained data.

Algorithm Configurators are an extension of the more well-known topic called hyperparameter optimization (HPO). According to Eggensperger et al. [13], the goal is to optimize an algorithm based on a given performance metric $m : \Theta \times \pi \rightarrow R$ across a set of problem instances $\pi \in \Pi$. They thus express the portfolio optimization problem as:

$$\theta^* \in \arg\min_{\theta \in \Theta}\{\frac{1}{|\Pi|} \sum_{\pi \in \Pi} m(\theta, \pi)\}. \tag{3.6}$$

As stated by the authors, the concept of problem instances arises in the context of parameterized solvers for problems like proposition satisfaction. Nonetheless, it is argued that the reason HPO is a special case of AC is because in HPO we can use the cross-validation folds in machine learning models as problem instances $\Pi$. With this in mind, we can also argue that an AC can just as well generalize to database parameter optimization where database workloads (sets of predefined queries) are seen as problem instances. Since we measure the database performance by executing a specific workload, we can define a problem instance as the workload itself. Hence to create different instances we could experiment with different ratios of read and write queries as workloads. Parameter importance can then be studied as a function of these instances. Doing that may reduce the complexity of the parameter importance study by just creating one ranking of parameters that generalizes to all types of workloads. However, performing a parameter importance analysis that way is questionable. In many cases we want to optimize a database for a specific workload if we assume that it remains consistent over a relatively long period of time (e.g. many times longer than what it takes to optimize the database). It thus wouldn't make sense to generalize any results in this context, especially when taking into account the possibility of different workloads leading to very different importance rankings that would get lost in aggregation. In the more appropriate single workload case, we effectively only using one problem instance such that the expression above simplifies to the following:

$$\theta^* \in \arg\min_{\theta \in \Theta}\{m(\theta)\}. \tag{3.7}$$

The second part is understanding how CAVE handles configurations that we have previously not evaluated. CAVE utilizes a surrogate model as an *Emprical Performance Model* (EPM) that is retrained on the input data we provide CAVE from a previous algorithm (optimization) run we have performed. Thus, when calling a CAVE method, all of the configurations that it incorporates are used in predictions to obtain their performance values. This surrogate model is ultimately a random forest model whose own parameters have been optimized using an optimization algorithm. The source code lies in a referenced package that the aforementioned research group is responsible for in [3]. Now that we know that CAVE can be used in the context of databases and how it handles configurations not previously evaluated, we can learn more about some of the framework's interesting parameter importance methods below.

## Local Parameter Importance

One parameter importance method from CAVE is called Local Parameter Importance (LPI). LPI is a straightforward method based on the human strategy to look for performance changes

after changing one parameter in a specific configuration. The method essentially takes the best performing configuration (the *incumbent*) and for each parameter then calculates the variance of the performance values obtained by changing the value of that parameter. The method can be explained as follows. Given the incumbent, we can compute the importance of a parameter $p$ by following these steps:

1. For every value $v$ in $p$, determine a new configuration and evaluate its performance using the EPM. Store all resulting performance values.

2. Calculate the variance of the stored performance values.

3. Calculate the importance of $p$ by dividing the calculated variance by the sum of all variances after using this method on all parameters involved.

The reason why the method has the word 'local' in it is presumably because we only use one sample ( the incumbent configuration) and then explore the samples around it by changing one parameter in the configuration at a time. Essentially we are not taking large leaps away from the incumbent.

One thing we would like to mention regarding point 1 in the method is that in the case of a continuous parameter, CAVE takes 500 evenly spaced samples to consider within the parameter's value boundaries. In this thesis we do not use continuous parameters (see section 3.1.4).

## Ablation Analysis

CAVE provides another parameter importance method called Ablation Analysis (AA), which is based on a drastically different view of how to calculate the importance of a parameter. AA is based on investigating a path (or series) of configurations where parameter values are modified from the source (default configuration) to the incumbent. Parameter values are modified once every iteration and the modification that leads to the best performing configuration makes the modified parameter the most important one during that specific iteration. The steps can be summarized as follows:

1. For every parameter $p$, create a new configuration based on the default configuration. Modify the default of value of $p$ to its value in the incumbent configuration.

2. Out of all new configurations with single modifications, choose the configuration that results in the highest performance (lowest cost) obtained using the EPM. The most important parameter is the one that has been modified to obtain this configuration.

3. In the next iteration, we start with the configuration as previously that showed the best performance. We now make single modifications to the remaining unmodified parameters and repeat what we did above.

4. Repeat for future iterations until there are no remaining parameters that we can modify by changing their default to their incumbent values.

Ultimately the maximum number of iterations in the method is the number of parameters present in a configuration. If the value of a parameter is the same in the default configuration

as it is in the incumbent, it will simply be ignored. Thus it is possible that the importance ranking of the parameters include only a subset of the parameters.

While the method arrives at a ranking in a straightforward fashion, we also want to understand the importance value that is assigned to each parameter. By looking at the source code in [3], we found that the the importance value attached to a parameter is based on the improvement attained in a specific iteration. For example, let's consider the first iteration, where all parameters are subject to modifications. For each parameter, we calculate the resulting performance as described above and store this value. When we move on to the next parameter, if we find a performance value that is better, we replaced the stored value with it. The 'winning' parameter gets an importance value (or *weight*) $w$ that is calculated as follows:

$$w = \frac{\text{current configuration performance} - \text{previous best performance}}{\text{incumbent performance} - \text{ source performance}} \tag{3.8}$$

where the previous best performance is the best performance found so far in the iteration. The reason the value is divided by the difference in the performance of the source and incumbent configurations is to express the value as a percentage of the total improvement made from optimization. Note that it is possible to obtain a negative importance value. If every modified configuration in an iteration leads to worse performance values, we would be in a situation where we are trying to choose the parameter that has the least negative impact. Dealing with negative values is difficult in the context of the this thesis where we compare the values obtained using different parameter importance methods. Hence if a parameter receives a negative importance value from AA, we choose to set it to zero. Also, we normalize the remaining set of importance values to sum to one again.

## Functional ANOVA

Functional ANOVA, or fANOVA, is a method based on treating the objective function as an additive combination of several components. This method in CAVE is based on the implementation by Hutter et al. in [15] (also provided as a standalone library). An objective function, like in our case the throughput of a database, is broken down into components that act as different sources of contributions to the function values. Specifically, we can divide the types of contributions into different *orders*:

- *First-order* effects: Also called *main* effects, these are contributions we see from tuning one single parameter at once.

- *Second-order* effects: Contributions which are due to changing how the values in two parameters at once, emphasizing the interactions between them.

- *Higher-order* effects: Similar to above but covers interactions between multiple parameters at a time, with at least three parameters or more.

The fANOVA method in CAVE calculates parameter importance values by considering first order and pairwise second order effects. For every parameter $p$, we calculate the parameter importance by following these steps:

1. For every value in $p$, calculate the average performance value $y_p$ obtained when marginalizing over all other parameters [1].

2. Compute the parameter importance by calculating the variance of all $y_p$ (later normalized so that all importance values sum to one).

We do similar calculations for pairwise second order importance values, but instead compute an average performance value for every combination of *two* values in two different parameters. The importance values are in the end all normalized so that they sum to one.

The first step requires an exponential number of calculations that scale with the number of parameters, which means that this method can be expensive to run. Fortunately, the clever fANOVA implementation by Hutter et al. [15] specifically targeting decision tree models like random forests is able to run in linear time by exploiting the fact that predictions are made very quickly.

## A New Feature Importance Ensemble Method

Comparing the results obtained from the above four methods is useful in trying to understand their differences. However, in order to see if we can get matching improvement gains from optimization by only using a select number of the most important parameters means we have to aggregate the results in some manner. Since the methods may return different parameter importance rankings, we need to construct an ensemble method based on the results of the four parameter importance methods we are studying. To this end, we aggregate the results by summation: For each parameter $p$, we calculate its aggregated importance $I_p$ by:

$$I_p = \sum_{n=1}^{4} w_p(k) \tag{3.9}$$

where $w_p(k)$ is the performance value returned by a method $k$ for parameter $p$.

After performing this ensemble, we need a way to decide what parameters are most important. We could, for example, pick a number $k$ and choose the top $k$ parameters according to the values obtained above and include them in our final reduced set. In this thesis we instead set a threshold value that would divide the parameters into groups: (1) parameters with values above the threshold and thus included in the reduced set and (2) parameters with values below it. What threshold one should set in itself is an open discussion; but for this thesis we will set it at a value of 0.2. Considering that each parameter importance method returns values that sum to one, we can think of a threshold of 0.2 as allowing parameters with at least 20% contribution to the objective function become labeled as important. This is, however, slightly misleading since when we perform the ensemble calculation in equation 3.9, a parameter may have a value that is larger than 1. Thus in the extreme case where every method identifies only one and the same parameter contributing towards the objective, it would receive the value 4.

To summarize, creating a threshold in this manner allows us to create a reduced search space in a new optimization run that consists of parameters with importance values above the threshold. Then we can evaluate the improvement gain in the optimization run using the

[1] Marginalizing over other all other parameters means we collect all combinations of possible configurations with the fixed value of the parameter $p$.

new and reduced search space. One thing we can note is that when the importance values of the parameters are more equally distributed (i.e. similar to each other), we would see more parameters that lie above the threshold. In contrast, if some parameters are especially dominant, then our new sample space would be formed from fewer parameters.

We also have to remember that the method fANOVA provides importance values for both the individual parameters and all possible combinations of two parameters. Hence in order to aggregate like above we have to consider only the individual parameter values returned by the method. Due to this, the sum of these values may not be one since the pairwise second order factors receive non-zero importance values. To fix this we normalize the individual importance values to sum to one.

### 3.1.4 Parameters considered

The DBMS parameters we took into account were chosen from a variety of sources to ensure a good starting point for optimization. We want to start with a set of parameters that we suspect have an impact on the database performance. Although this approach is counter intuitive to the point of utilizing parameter importance methods in the first place to select a set of knobs that are most efficient, we still want to save time and avoid having an initial search space that is too large. If the initial optimization process considered all possible parameters, it would take forever in the case of a database like PostgreSQL to properly explore its search space with hundreds of parameters. The chosen parameters for the databases we have studied are thus based on previous research, online documentation and blogs that have presented impacting parameters. Since we specifically study the throughput in this thesis, we have selected sources that have studied the throughput as a metric. If information relating to the throughput is missing, other metrics like latency were considered. Below we present the parameters considered for each database in our study.

## RocksDB

The parameters chosen for RocksDB were largely based on a paper by Alabed and Yoneki [2] that explored tuning RocksDB parameters to optimize a problem with multi-task objectives. The authors sought to maximize the database performance indirectly through three smaller objectives. A total of 10 knobs were used and they were all chosen using prior expert knowledge and then manually clustered into the multi-objective categories. An additional parameter observed by Ouaknine et al. [30] in a performance analysis as well as two categorical parameters mentioned in the RocksDB online documentation were deemed important and will be considered. In total this makes 13 knobs. Besides choosing the parameters, their value ranges were also inspired from the sources mentioned, either directly if they are explicitly defined, or indirectly from an intuition about previous results obtained using the parameters. Table 3.1 lists all the selected knobs in this thesis.

**Table 3.1:** The RocksDB knobs we consider in our optimization and parameter importance analysis.

| Knob (total 13) | Value range | Default value |
|---|---|---|
| block_size | $1, 2^x, x \in [2, 19]$ | $2^{12}$ |
| cache_index_and_filter_blocks | {false, true} | false |
| compaction_readahead_size | $x \cdot 10^4, x \in [0, 10]$ | 0 |
| compression_type | {snappy, zstd, lz4} | snappy |
| level0_file_num_compaction_trigger | $2^x, x \in [0, 8]$ | $2^4$ |
| level0_slowdown_writes_trigger | $0, 2^x, x \in [0, 10]$ | 0 |
| level0_stop_writes_trigger | $2^x, x \in [0, 10]$ | $2^5$ |
| max_background_compactions | $2^x, x \in [0, 8]$ | 1 |
| max_background_flushes | $[1, 10]$ | 1 |
| max_bytes_for_level_multiplier | $[5, 15]$ | 10 |
| max_write_buffer_number | $2^x, x \in [0, 7]$ | 2 |
| min_write_buffer_number_to_merge | $2^x, x \in [0, 5]$ | 1 |
| write_buffer_size | $1, 2^x, x \in [2, 30]$ | $2^{26}$ |

## PostgreSQL

The list of configuration parameters that form the initial search space for PostgreSQL optimization is shown in table 3.2. For this DBMS, we managed to receive expert advice from developers at DBtune [11], a company that provides database parameter optimization services with a lot of experience optimizing PostgreSQL. From them we were able identify a list of relevant parameters that can impact the throughput performance as well as their valid value ranges. For the purpose of protecting intellectual property rights, the value ranges for the parameters are not shown in table 3.2.

**Table 3.2:** The PostgreSQL knobs we consider in our optimization and parameter importance analysis. The value ranges are not provided due to intellectual property rights.

| Knob (16 in total) | Default value |
|---|---|
| bgwriter_delay (ms) | 200 |
| bgwriter_lru_maxpages | 100 |
| checkpoint_timeout (mins) | 5 |
| deadlock_timeout (ms) | 1000 |
| default_statistics_target | 100 |
| effective_cache_size (kB) | 2016383 |
| max_parallel_workers | 4 |
| max_parallel_workers_per_gather | 2 |
| max_worker_processes | 4 |
| max_wal_size (GB) | 8 |
| default_statistics_target | 100 |
| random_page_cost | 1.1 |
| shared_buffers | 4096 |
| temp_buffers (kB) | 1024 |
| wal_buffers | 8192 |
| work_mem (kB) | 6990 |

## Neo4j

We were able to contact a developer at Neo4j with extensive knowledge of their platform that provided us with parameters that were interesting to tune. On top of that, Neo4j itself offers a performance tuning guide [27] with focus on memory management. This guide is also part of a larger operations manual [26] that goes into depth on configuring different parts of the system, such as garbage collection. Neo4j runs in a Java Virtual Machine (JVM), a Java compiler system that takes care of the memory management of class instances and arrays belonging to the application it runs. Table 3.3 shows the list of parameters we considered. All parameters were inspired by the suggestions of our contact at Neo4j with the exception of the JVM parameters that we had to choose ourselves. A JVM itself has a lot of configuration parameters that can be tuned but in order to not complicate the optimization we chose two parameters that seemed influential in regards to garbage collection. The value ranges for most parameters were arbitrarily as plausible distributions but the parameters *dbms.memory.heap.max_size* and *dbms.memory.pagecache.flush.buffer.enabled* required some manual testing first. The two parameters use up different parts of the RAM so they cannot together add up to more than there is RAM available in the system. If an error relating to this occurs, we consider that specific configuration (i.e. sample in the optimizer search space) as invalid.

**Table 3.3:** The Neo4j knobs we consider in our optimization and parameter importance analysis.

| Knob (10 in total) | Value range | Default value |
|---|---|---|
| checkpoint.interval.time (mins) | $[5, 25]$ | 15 |
| checkpoint.interval.tx | [1] linspace$(10^4, 2 \cdot 10^5, 10^4)$ | $10^5$ |
| [2]jvm.additional - GC algorithm | {G1, Parallel, Serial}GC | G1GC |
| [3]jvm.additional - Survivor Ratio | {none, 2, 4, 6, 8} | [4] none |
| heap.max_size (GB) | linspace$(4.1, 37.1, 4)$ | 24.1 |
| off_heap.max_size (GB) | linspace$(0.5, 5, 5)$ | 2.00 |
| pagecache.flush.buffer.enabled | {false, true} | false |
| pagecache.flush.buffer.size_in_pages | $2^x, x \in [0, 9]$ | $2^4$ |
| pagecache.size (GB) | linspace$(4, 38, 4)$ | 50% of (RAM) $\approx$ 28 |
| tx_state.memory_allocation | {OFF, ON}_HEAP | OFF_HEAP |

[1] Means an evenly spaced sample distribution with arguments *(start value, stop value, number of samples to draw in between start and stop values)*.
[2] Parameter for choosing the garbage collection algorithm.
[3] Parameter for specifying the JVM survivor ratio. This is the ratio between the survivor space and eden space. The eden space is the pool with memory that is allocated to most application objects and the survivor space is the set of objects that have survived garbage collection from the eden space. If the ratio is too small, we get an overflow into an 'older generation space'. If it is too large, the survivor space could be mostly empty.
[4] 'none' meaning this parameter is left alone to be set by the operation system.

# Chapter 4
# Evaluation

## 4.1 Setup

### Optimization

The discussed CAVE methods were applied using optimization data provided as CSV files by DBtune, after first being converted into a valid format. During optimization, the number of samples we choose to evaluate is hindered by a limitation in computational resources. The more samples the better, but the process should be done within a reasonable amount of time for a user. In our case, we chose D+1 samples during the warm-up phase and 30D samples in the main phase, where D is the number of parameters. The first number is low because we want to spend a lot less time during warm-up than doing actual optimization.

### Hardware

All experiments for database optimization were run using using cloud instances on SNIC [36] with 4 CPU cores and 8 GB RAM. Every DBMS was dedicated its own instance so as to not interfere with each other and minimize the activity produced by background applications. A hard drive disk (HDD) is attached to each instance with a capacity such that it can hold all the data stored in a DBMS for benchmarking.

### Software versions

RocksDB version 6.22 and PostgreSQL 14.1 were used. The benchmarking tool db_bench was already provided together with RocksDB, while for PostgreSQL the latest version of the OLTP-Bench repository on Github was used (commit reference *9279ce2ed9aa130afd08c0-de29b81bd99a9af008*). For Neo4j, we used the version of LDBC SNB provided by the commit reference *1333d4cf94eabf79ebb8a50316ab143fa2496769* that utilizes version 4.3 of the Neo4j community edition.

### Benchmarking

A benchmark has several properties that can be considered. Since we used different benchmarking tools we have to acknowledge the differences in how we benchmark and measure the performance of the DBMS we study. Here is a list of things we configured for our benchmarks:

1. **Size of data**: If the size of data we store in the database is too small then it is trivially handled by the DBMS. We also want to consider the case when users do not have sufficient RAM in their system to store all their data so we want data sizes that are larger than the RAM available (in our case this means > 8 GB). At the same time, the size of data cannot be so large that it would take too long to refresh the state of the DBMS between consecutive benchmark runs.

2. **Workload sensitivity**: We tried to explore different workloads where it is possible to gather insights on what impact it has on the parameter importance results. Specifically, this entails changing the mixture of read and write queries to make the DBMS more read or write heavy. For RocksDB and Neo4j this is trivially handled by the benchmarking tools *db_bench* and LDBC SNB due to the presence of a parameter that specifies the ratio of read to write queries we want to use. In the case of PostgreSQL, we chose to look at different workloads offered by OLTP-Bench and described in the original paper [12]. These workloads work a bit differently, where although the amount of read-only queries can be calculated as presented in the paper, the workloads still rely on what so called *transaction profiles*. Transaction profiles define a specific sequence of actions that model real-world behaviors and usually contains multiple different queries executed in succession.

3. **Number of threads**: Both *db_bench* and LDBC SNB allow users to run queries in parallel by specifying the number of connection threads we should have open. We chose to use a constant number of 8 threads where possible. We do not explicitly study different number of threads as we believe as long as it is kept consistent the result we get would be valid.

4. **Benchmark execution time**: Evaluation of a sample during optimization was done by running a benchmark for 10 minutes. In between benchmarks runs the state of the database is restored, which takes a few varying number of minutes depending on the benchmark tool that is used.

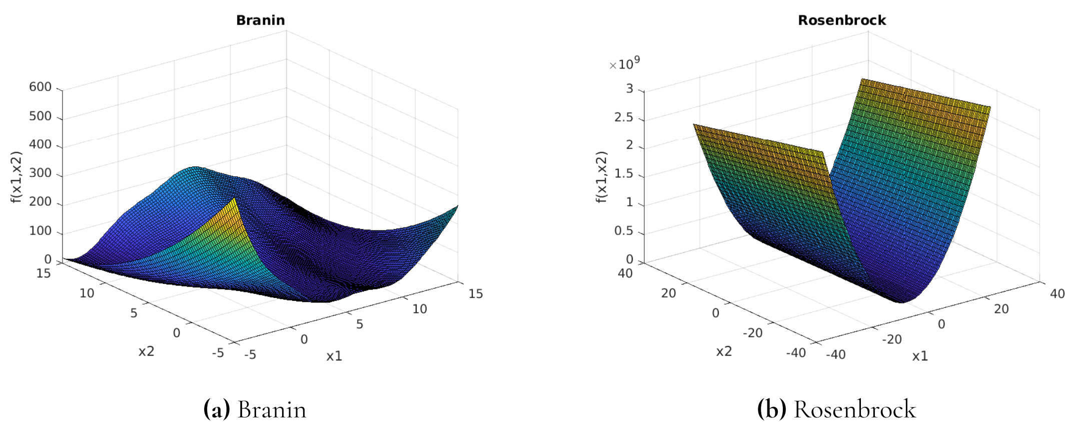
## 4.2 Parameter Importance Study

### 4.2.1 Test functions: Branin and Rosenbrock

#### Results

To gain an understanding of how the parameter importance methods work in practice, we applied them on two well known optimization test functions called Branin and Rosenbrock. Figure 4.1 shows how both functions look like in two dimensions. These functions were

already supported by DBtune and we optimized them using 100 warm-up iterations with random sampling and 1000 iterations in the main optimization phase. The results obtained are summarized in table 4.1. Note that the method AA considers both the default and best performing configuration in order to calculate parameter importance. Since the default configuration can be arbitrarily set by the user for both Branin and Rosenbrock (i.e., we can choose the starting point ourselves), we have chosen to use the worst performing configuration as the default configuration in AA. This would at least perhaps cover a large difference in performance between the default and the incumbent configurations, which could be our best bet in analyzing parameter importance for these functions. Finally, CAVE creates some insightful plots for the LPI and AA methods, which we show in figures 4.2-4.3.

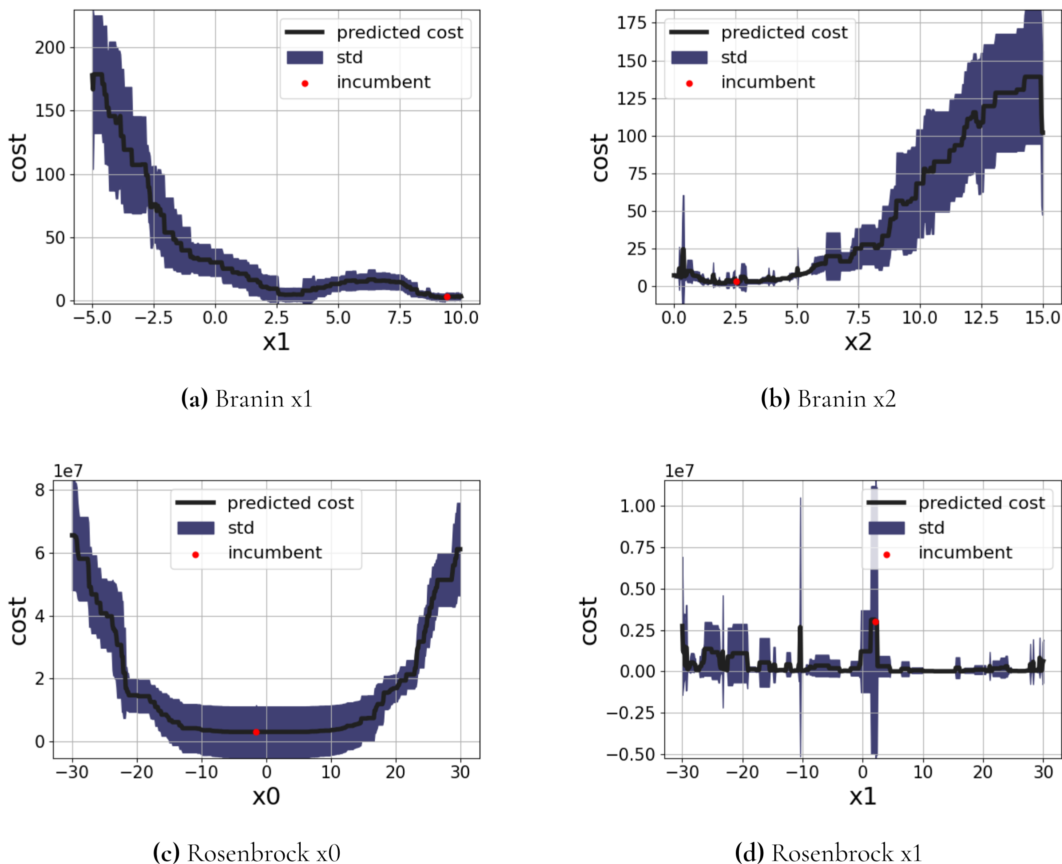



**(a)** Branin **(b)** Rosenbrock

**Figure 4.1:** 3D surface plots of two optimization test functions. The expressions for the functions Branin and Rosenbrock can be found in [7] and [8] respectively.

| Parameter | RF | LPI | AA | fANOVA | Ranking {RF, LPI,AA,fANOVA} |
|---|---|---|---|---|---|
| Branin $x_1$ | 0.665 | 0.509 | 0.921 | 0.111 | {1,1,1,3} |
| Branin $x_2$ | 0.335 | 0.491 | 0.079 | 0.255 | {2,2,2,2} |
| Interaction $\{x_1,x_2\}$ | - | - | - | 0.634 | {-,-,-,1} |
| Rosenbrock $x_0$ | 0.940 | 0.952 | 1.003 | 0.896 | {1,1,1,1} |
| Rosenbrock $x_1$ | 0.060 | 0.048 | -0.003 | 0.015 | {2,2,2,3} |
| Interaction $\{x_0,x_1\}$ | - | - | - | 0.089 | {-,-,-,2} |

**Table 4.1:** Parameter importance results for Branin and Rosenbrock. The second order parameters (or terms) in fANOVA have been included.




**(a)** Branin x1

**(b)** Branin x2

**(c)** Rosenbrock x0

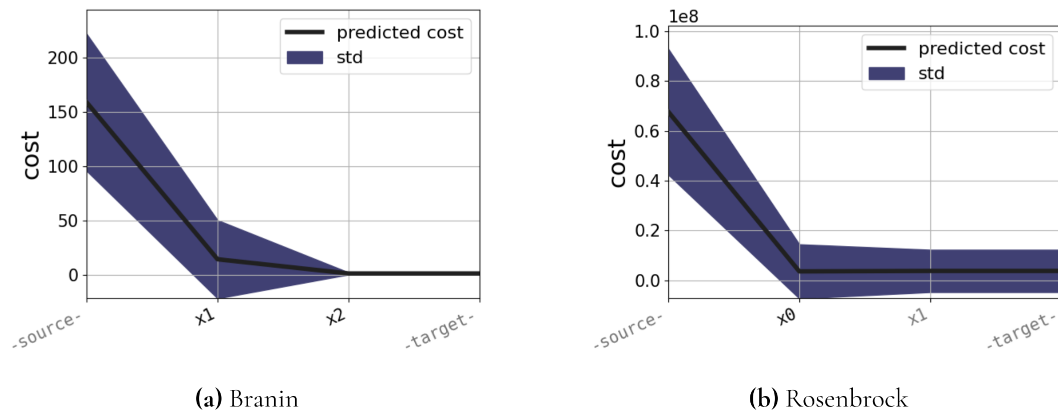
**(d)** Rosenbrock x1

**Figure 4.2:** Predicted cost of evaluated Branin (top) and Rosenbrock (bottom) configurations in Local Parameter Importance with uncertainty as the standard deviation.

## Discussion

The results show some interesting differences between the methods. For Branin, the importance appears almost equally distributed using RF and LPI. The plot in figure 4.2 shows that while the minimum is similar, the maximum of cost values in $x_1$ comes close to 200 while in $x_2$ to 150. This slightly larger variation in cost values means that the variance calculated for cost values due to $x_1$ is higher, but also explains why the parameter importance values calculated using LPI are almost equal. In contrast, AA identified $x_1$ as drastically more important, which can be explained by the fact that it was enough to use only $x_1$ to obtain a function value that is substantially lower (close to local minimum). This means that modifying the remaining $x_2$ caused only a small decrease in the objective. On the other hand, fANOVA complements these results with an interesting insight showing that the interaction between $x_1$ and $x_2$ is much more important than the individual contributions of the two parameters. Essentially this means that for Branin, fANOVA tells us that it is better to tune both parameters at the same time rather than have a narrower focus by analyzing singular parameters. On the contrast, AA identified values of $x_1$ that lead to a very close minimum which means that perhaps a user would be satisfied with only tuning that one parameter by itself.

The results for Rosenbrock show that $x_0$ is the clearly dominant parameter by all meth-




**(a)** Branin **(b)** Rosenbrock

**Figure 4.3:** Ablation Analysis plots for Branin and Rosenbrock with the cost (function value) obtained along a path from the default (source) configuration to the incumbent (target) with uncertainty as the standard deviation. A decrease in cost means a decrease in function value.

ods. Despite this, AA shows rather peculiar importance values. We know that the assigned importance weights always sum to one, but it appears strange that $x_1$ is assigned a negative value, forcing $x_0$ to have a value larger than one. The complementary plot in figure 4.3 for Rosenbrock helps us explain these results. We see that after only modifying $x_0$ we get a sufficiently low value such that modifying $x_1$ afterwards to reach the incumbent (denoted as 'target' in the plot) may well lead to increase in cost. Hence these results tell us that for Rosenbrock it is enough or perhaps even better to only be tuning $x_0$ to reach configuration with very low cost. Despite this, we could try to discuss the uncertainty in the EPM predictions made using the Rosenbrock optimization data. Looking at the LPI plots in figure 4.2, we see that while the plot for $x_0$ seems feasible, the one for $x_1$ appears to be at certain points highly unreliable. The uncertainty is very high within a narrow band around the incumbent. To further validate the obtained results, one could try to reduce the discussed uncertainty by e.g. providing more optimization data.

The parameter importance methods from CAVE ultimately depend on the accuracy of the EPM predictions. The EPM's accuracy in turn is dependent on the amount of training data that we have provided it, so it is likely in the case where we obtain highly contradicting results that we have simply not provided enough training data. Now that we have studied the two test functions and hopefully better understand the parameter importance methods, we are ready to do similarly for database parameters.

### 4.2.2 RocksDB

A parameter importance study was conducted on RocksDB optimization data for three workloads with different read to write query ratios: 1:9, 1:1, and 9:1 in the benchmarking tool *db_bench*. An analysis was performed on each workload so that we can compare them and investigate if any drawn insights can be generalized. The resulting parameter rankings are thus specific for the workload they are associated with. The throughput results for default

and best found configurations are shown in table 4.2. We have also plotted the best found configuration found so far at every iteration as a means of drawing a convergence curve in figure 4.4. Every optimization run is repeated twice (for a total of 3 runs) in order to plot an error region. The curve represents the mean best performance found so far. The shared error region represents the difference between the maximum and minimum best performance found so far among all three optimization runs that were performed.

| Workload (r:w) | op/s at default | op/s at incumbent (best) | Improvement |
|---|---|---|---|
| 1:9 | 142,195 | 161,622 | 13.7% |
| 1:1 | 124,853 | 199,193 | 59.5% |
| 9:1 | 149,040 | 333,824 | 124.0% |

**Table 4.2:** Throughput results from optimizing RocksDB using three different workloads with specific read to write (r:w) ratios.

### Results

The feature importance results for RocksDB are summarized in figures 4.5-4.7. Each figure has two plots for every workload studied, where the top plot shows the weights (values) returned by the parameter importance methods and the bottom one shows the values returned by fANOVA specifically. Note that we have only included the fANOVA terms that show a non-zero importance value, which means that not all terms are shown in the plot. Regarding plots from CAVE itself, we chose not to include them due to visual errors; for the test functions we studied above there are no problems but it appears that with many input parameters as in this case with RocksDB, the plots were not able to be visualized coherently. As of the time of writing this thesis, the CAVE repository is not actively maintained.

### Discussion

With three different workloads and four parameter importance methods, there is a lot to unpack in the results. Firstly, for the write-heavy workload in figure 4.5, we see that *write_buffer_size* appears to be a dominant parameter according to LPI and fANOVA. AA has a more equal distribution between the parameters while RF instead highly regards *max_background_compactions*. The write-heavy workload shows the least improvement attained at 13.7%, which suggests that the default configuration could already be highly optimized for write queries. If the improvement gained is very low, analyzing parameter importance would not be practical but it may say something about the worse configurations attained during the optimization process. The distribution of importance weights could say something about how changing certain parameters leads to worse performance. The parameter importance methods like LPI and fANOVA in our case end up measuring the variance of set of throughput values, which means that essentially even if we do not obtain a significant improvement gain after optimization, it is the contributions from the worse configurations that can have a large influence on the parameter rankings.

For the other workloads we see much better improvement gains. The read-heavy workload with a 9:1 read to write ratio has the highest improvement at 124.0%, which suggests that perhaps the RocksDB parameters we tuned are more sensitive to read queries. Therefore the

default RocksDB configuration could be said to not be directly optimized for read-heavy operations. For the 9:1 workload, the method AA finds that *level0_file_num_compaction_trigger* is most important with *block_size* and *write_buffer_size* receiving a considerable share. LPI agrees with this to a large extent but DBtune's RF shows a different perspective. It finds that the parameter *max_background_compactions* is significant while having a more equal distribution around multiple other parameters. The distribution is similar in the case of workload 1:1 in figure 4.6 but with all methods except RF placing more emphasis on *write_buffer_size*. This seems reasonable since the workload has a larger proportion of write queries that takes advantage of more optimal values for this parameter.

The bottom plots in figures 4.5-4.7 show that some second order terms are highly valued by fANOVA. However, the terms already consist of parameters that were highly valued by themselves as singular terms. What we draw from this is that interaction effects between the parameters in RocksDB are possibly influential but perhaps not dominant enough to raise an issue with using the ensemble method that we discussed in section 3.1.3 based on first order contributions that we will use to form a reduced search space.

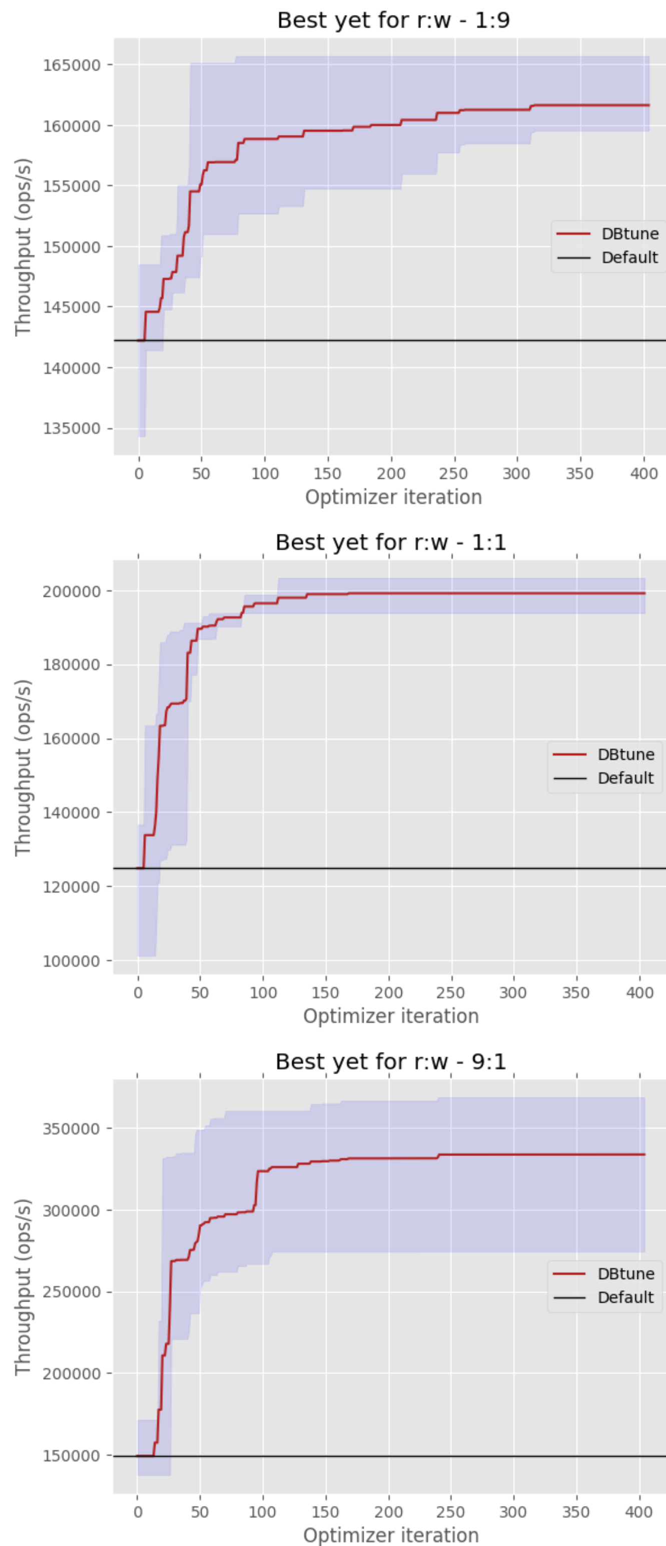



**Figure 4.4:** Three plots for the three workloads we studied showing the best achieved throughput during optimization so far using the initial set of RocksDB parameters we considered. DBtune represents the best found configuration so far by the DBtune as the optimizer.

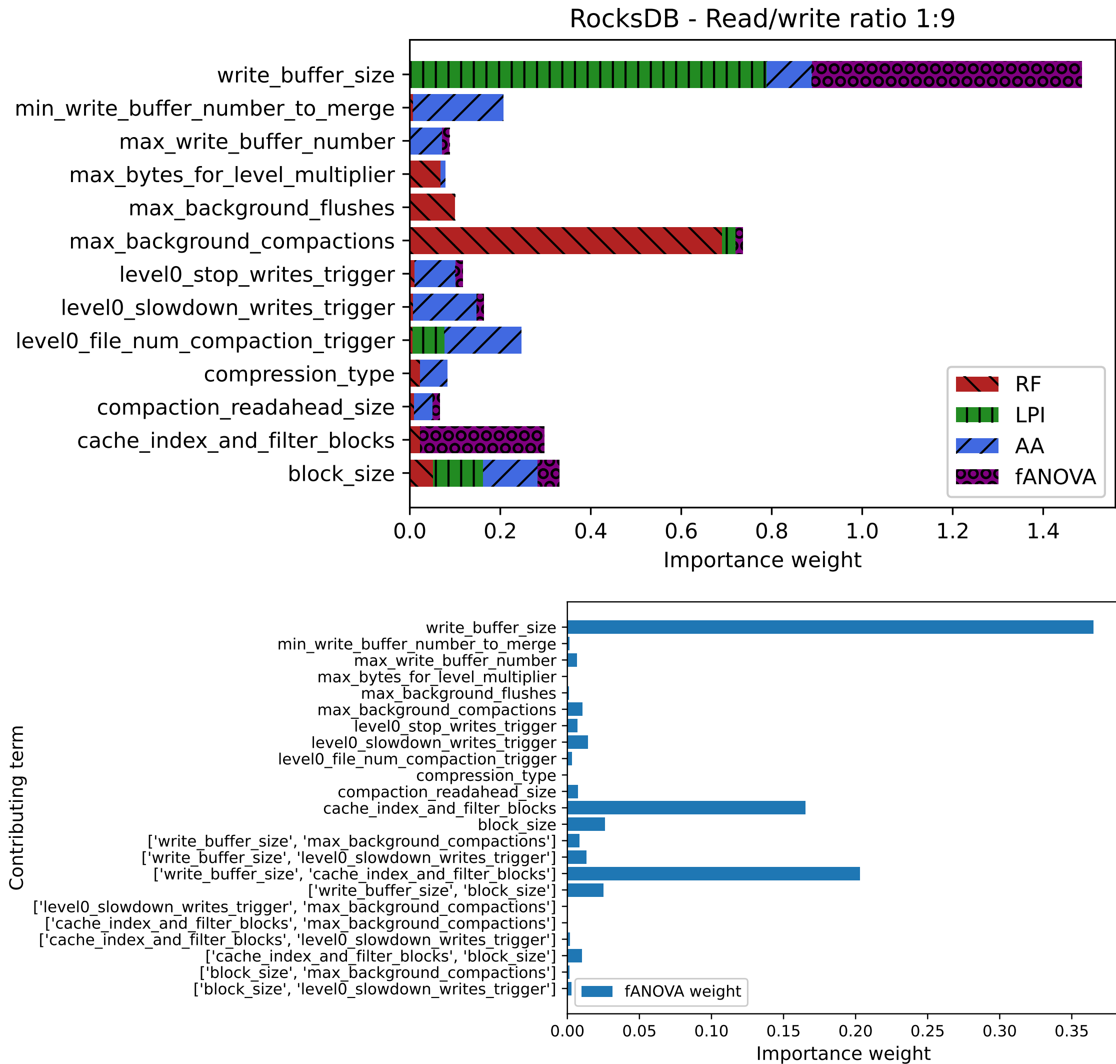



**Figure 4.5:** Results for RocksDB using a workload with a read to write ratio of 1:9. The stacked horizontal bar chart (top) shows the individual importance weights given to each parameter by the discussed importance methods. The bottom plot shows the values returned by fANOVA to showcase the importance of both singular parameter terms and the pairwise second-order terms.

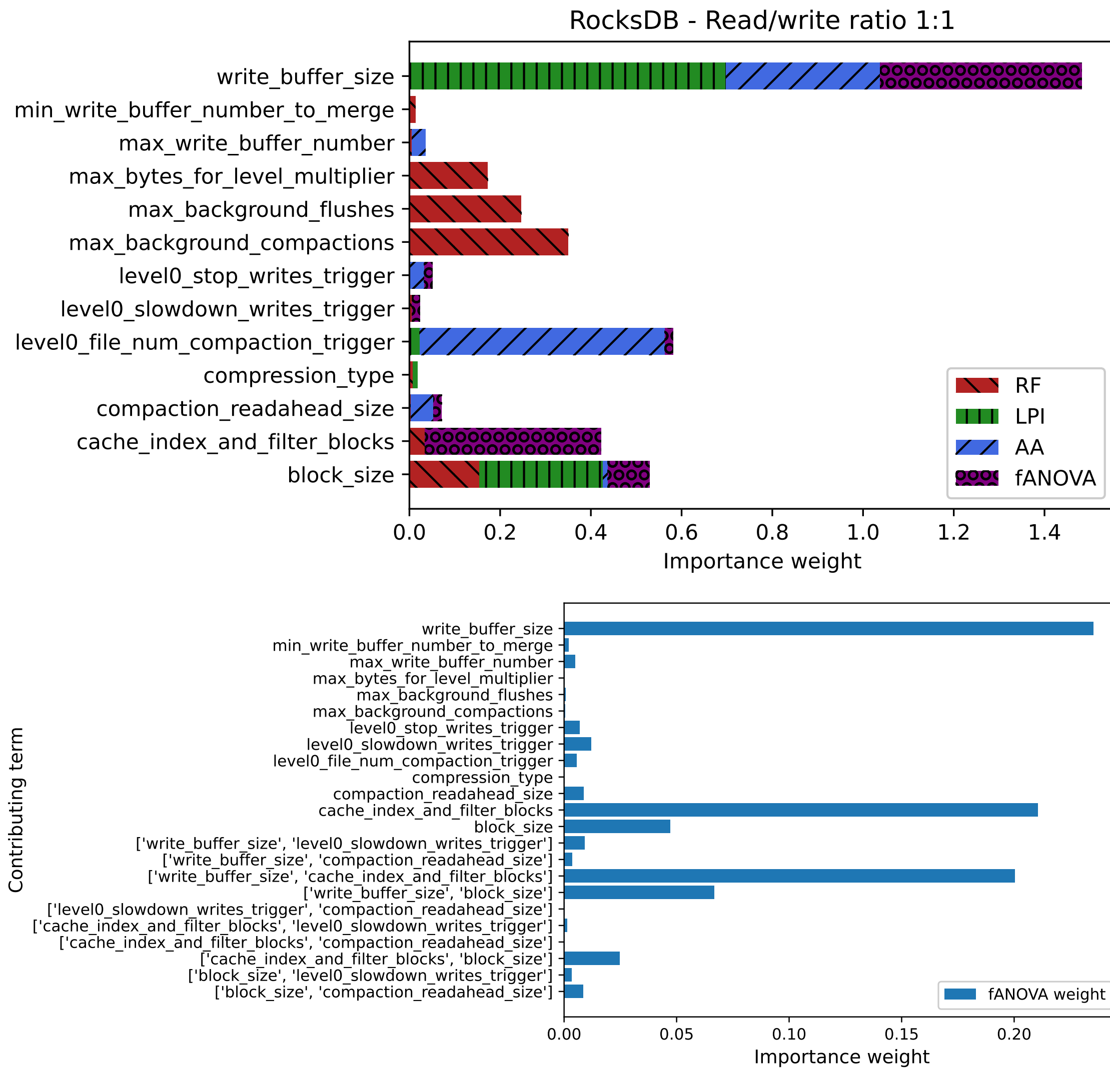



**Figure 4.6:** Results similar to figure 4.5 but using a workload with a read to write ratio of 1:1.

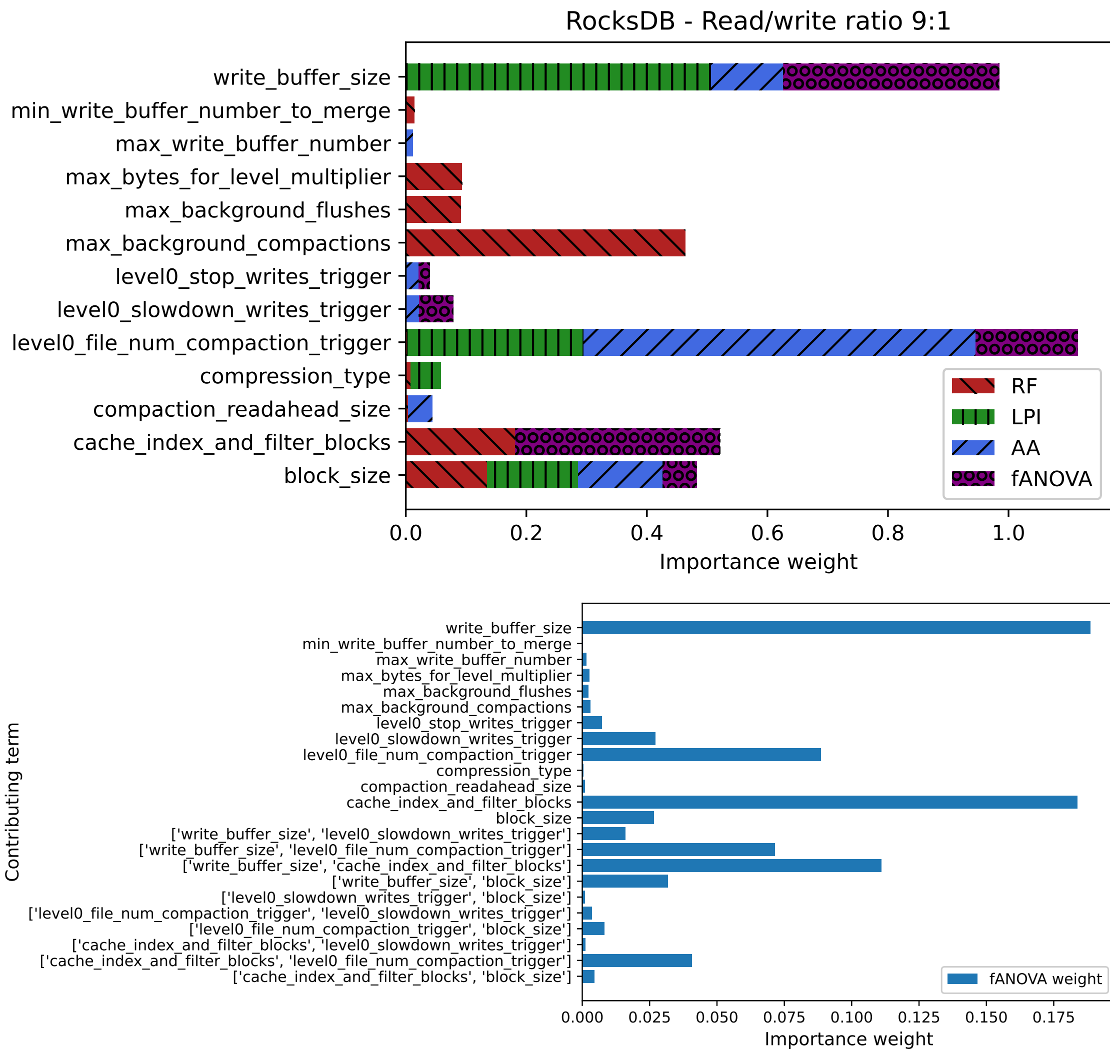



**Figure 4.7:** Results similar to figure 4.5 but using a workload with a read to write ratio of 9:1.

## Optimization with top RocksDB parameters

Using the ensemble method we defined in equation 3.9, the most important parameters we identify are listed in table 4.3. We can see that although the rankings differ, all three workloads are very similar, with the 1:1 workload including an additional parameter. A new search space consisting of these parameters were then used in a new optimization run to get new results, which are shown in table 4.4. The graphs showing the best achieved throughput so far are presented in figure 4.8

| Rank | r:w - 9:1 | r:w - 1:1 | r:w - 1:9 |
|---|---|---|---|
| 1 | level0_file_num_compaction_trigger | write_buffer_size | write_buffer_size |
| 2 | write_buffer_size | level0_file_num_compaction_trigger | max_background_compactions |
| 3 | cache_index_and_filter_blocks | block_size | block_size |
| 4 | block_size | cache_index_and_filter_blocks | cache_index_and_filter_blocks |
| 5 | max_background_compactions | max_background_compactions | level0_file_num_compaction_trigger |
| 6 | - | max_background_flushes | - |

**Table 4.3:** The selected most important parameters after studying workloads with different read to write ratios for RocksDB. The parameters form smaller search spaces in new optimization and the performance gains will be reported.

After determining a smaller set of parameters consisting of only the most important parameters, we see in table 4.4 that for all workloads we match at least 60% of the improvement gained from using the entire set of parameters we initially considered. Matching only at around 60% can be seen as rather unimpressive when comparing to the work done by Kanellis et al. [19] that consistently show at least 99%. The plausible reason for this is that the threshold we use in our ensemble is too high. Using the threshold value of 0.2, we were able to roughly halve the number of parameters. Using a more generous (lower) threshold would likely lead to better improvement numbers at the cost of a more complex search space due to the inclusion of more parameters.

It still appears that the more read-heavy a workload is, the higher the improvement gain that can be achieved. The improvement gain for the write-heavy workload is still small, which could potentially cause an issue surrounding the efficiency argument with using parameter importance methods. Since tuning a database overall takes time in order to properly explore a search space, it can be seen as infeasible for the user to spend a lot of computational resources in order to gain a few percentage points of improvement that might as well be attributed to noise. Nonetheless, this depends on the context in which the DBMS is used and the performance requirements for it. A few percentage points may deliver a large impact towards end users.

## Comment on the default RocksDB performance variation

The heavy-write (9:1) workload shows a large variation in performance for the default configuration (our starting point) in both figures 4.4 and 4.8. Why this occurred is unclear and our

| Workload (r:w) | $\text{Gain}_{Initial}$ | $\text{Gain}_{Refined}$ | (%) of $\text{Gain}_{Initial}$ |
|---|---|---|---|
| 1:9 | 13.7% | 10.1% | 73.7% |
| 1:1 | 59.5% | 38.4% | 64.5% |
| 9:1 | 124.0% | 86.5% | 69.8% |

**Table 4.4:** A comparison of the improvements gained from optimizing RocksDB for the three workloads we studied using the initial set of parameters from table 3.1 and the refined set from table 4.3.

best guess is that there was a lot of noise that influenced the CPU performance. Considering that we also obtain the least amount of improvement using this workload, this variation plays an even bigger role. Repeating the optimization procedure many more times would likely shed more light on this problem but due to time constraints we were not able to.

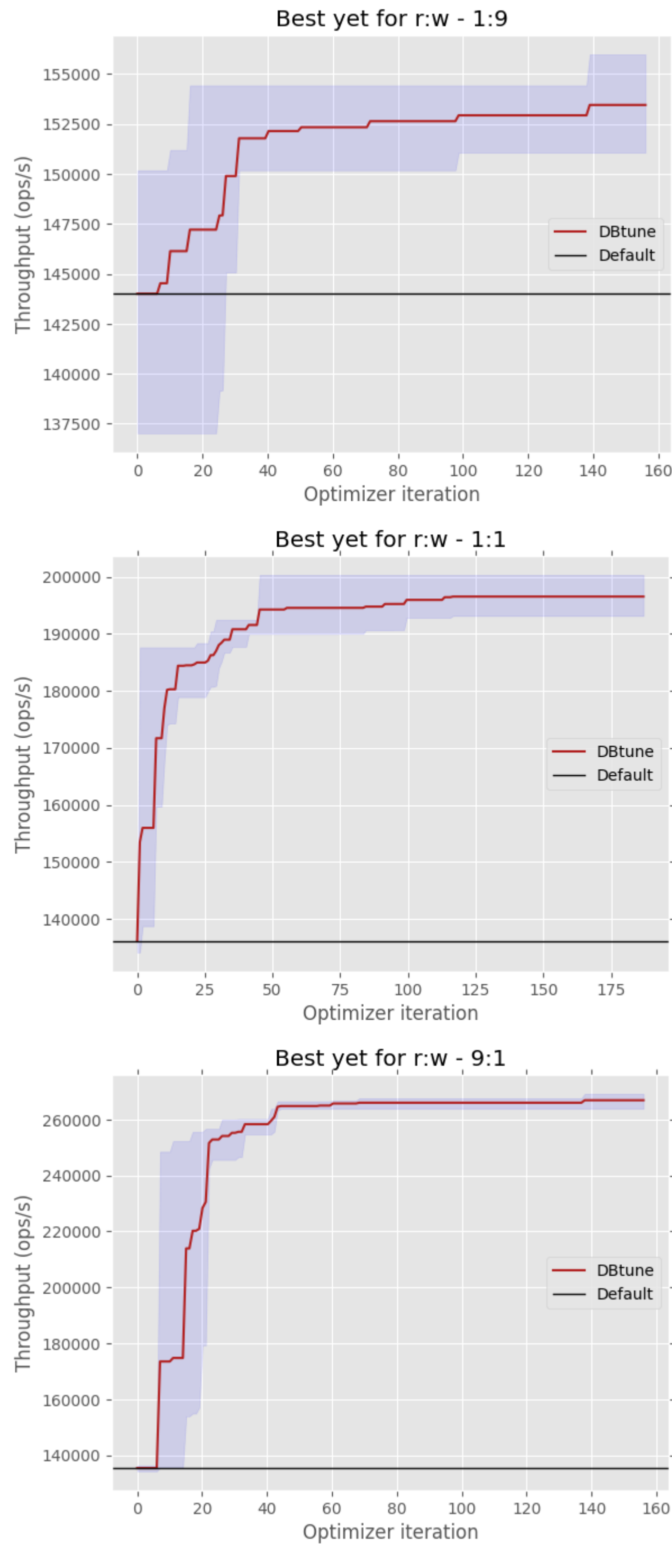



**Figure 4.8:** Three plots for the three workloads we studied showing the best achieved throughput so far using the reduced set of RocksDB parameters consisting of the identified most important parameters.

### 4.2.3 PostgreSQL

For PostgreSQL we explored two different workloads called *TPC-C* and *Wikipedia* that represent two very different sets of queries. As highlighted in the paper for OLTP-Bench [12], TPC-C consists of 8.0% read-only transactions by default, while for Wikipedia that number is 92.2%. Both workloads appear different in the types of transactions they execute, with the authors in [12] classifying TPC-C as a transactional workload while Wikipedia as a Web-oriented workload. With TPC-C being described as a classic industry standard for evaluating OLTP systems and Wikipedia representing the largest online encyclopedia, we thought the stark contrast between the two makes them good study subjects. TPC-C is a collection of five transaction profiles where some are read-only but others a mixture of read and write queries [37]. The significance of each transaction profile also differs, where some transactions are described by TPC-C to be more frequently occurring and requiring stringier response times as to not cause problems for users. While we could change the amount of each transaction profile we execute, we feel that it would distort the purpose of the benchmark, which emulates a complex retail setting. Therefore we use default values as specified by OLTP-Bench, where it appears that more weight is placed on the more important transaction types. The Wikipedia workload, as the name suggests, is based on the online encyclopedia *Wikipedia* and developed by the OLTP-Bench authors using information such as data dumps and browser access patterns. Essentially the workload expresses the most common type of operations found in Wikipedia.

The initial optimization runs using the parameters we listed in the chapter 3.1.4 in table 3.2 leads to the graph we see in figure 4.9 and the comparison between the default and best configuration found in table 4.5. Every iteration took around 30 minutes to evaluate, which means the entire optimization run took around between 10-11 days. Unfortunately due to time constraints in the project this pressured us to not repeat the optimization run in order to include an error area around a calculated mean curve as we showed previously for RocksDB in figure 4.4.

| Workload | op/s at default | op/s at incumbent (best) | Improvement |
|---|---|---|---|
| TPC-C | 351 | 528 | 50.4% |
| Wikipedia | 1,107 | 1,184 | 7.0% |

**Table 4.5:** Throughput results from optimizing PostgreSQL using two different workloads, TPC-C and Wikipedia.

#### Results

After applying the parameter importance methods on optimization data using the TPC-C and Wikipedia workloads we obtain the plots in figures 4.10 and 4.11.

#### Discussion

To start off, we see that the optimizing PostgreSQL for read-heavy workloads such as Wikipedia appears to be inefficient, given the much smaller improvement gain in comparison to TPC-C as we see in table 4.5. From this may we reason that PostgreSQL is already optimized well for

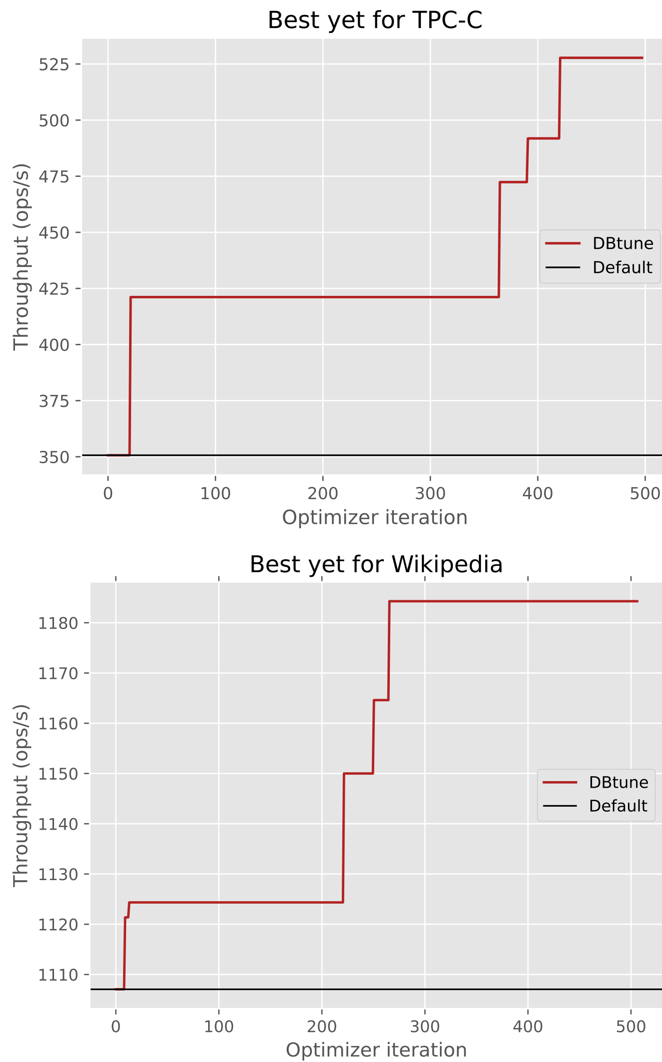



**Figure 4.9:** Two plots showing the best achieved throughput so far using the initial set of parameters we considered for PostgreSQL. The top plot is for the TPC-C workload and the bottom is for Wikipedia. This graph is similar to the one for RocksDB in figure 4.4 but includes only a single optimization run due to time constraints.

read-heavy applications by default and write queries made to the DBMS are more sensitive to changes in the parameters we considered.

Concerning the parameter importance results, the TPC-C workload shows us that some parameters like *shared_buffers* and *work_mem* dominate according to most methods. LPI and fANOVA seem to agree with each other the most. AA assigned some high values for parameters like *max_wal_size* and *deadlock_timeout* that other methods thought were insignificant. Since AA in practice creates a path of configurations from the source to the incumbent, we can perhaps say something about the change in performance along that path being hard to predict. This can occur if the EPM in CAVE does not have sufficient training data that makes it difficult to identify patterns in the search space to allow for better predictions. It could also be that along the path there exist influential interaction effects between certain parameters

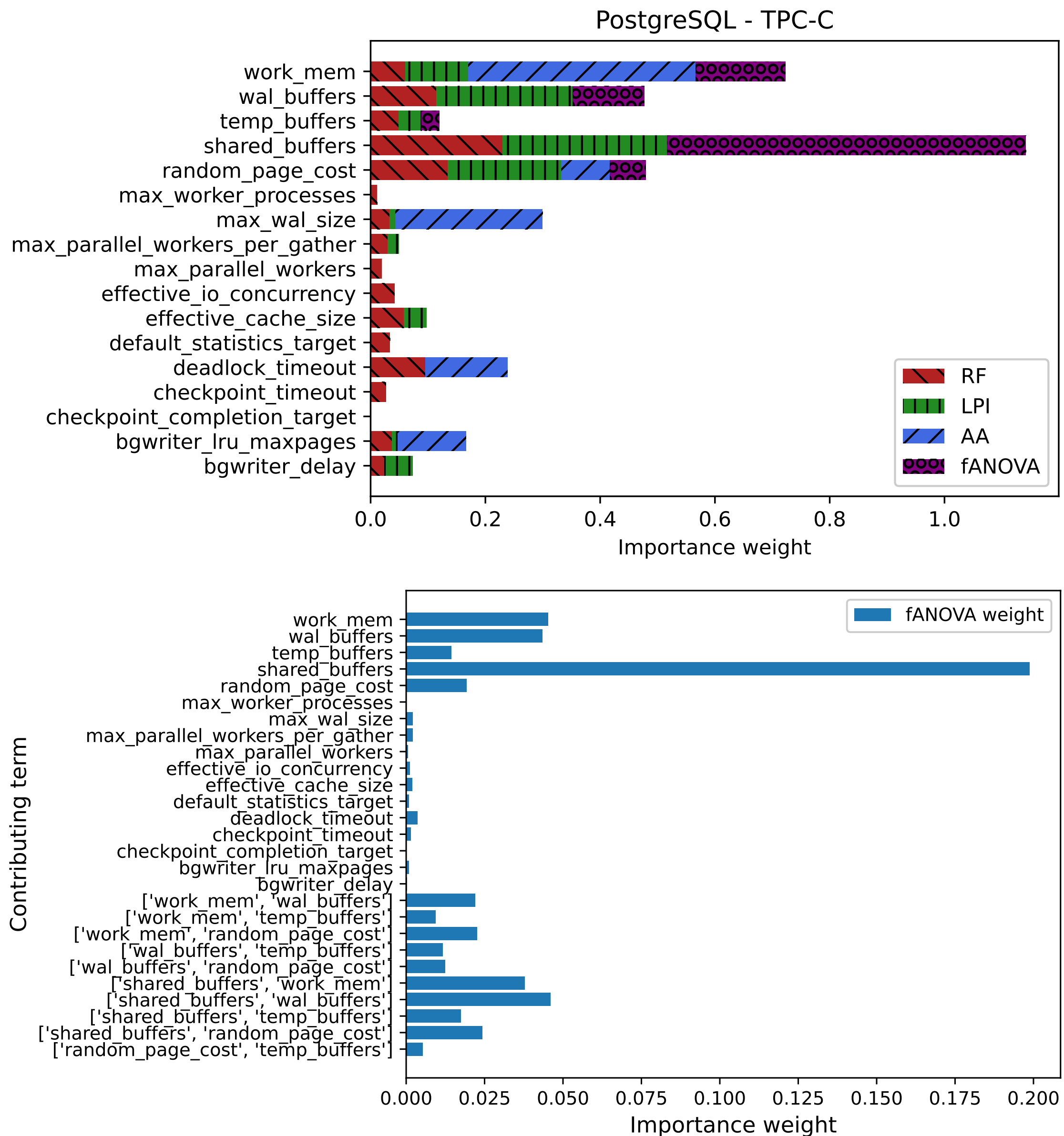



**Figure 4.10:** Feature importance results for PostgreSQL using the TPC-C workload in a similar fashion as to what was shown for RocksDB in figure 4.7.

that impacts the performance. These interactions effects are ideally captured by fANOVA, but at least for second-order terms this doesn't appear to be the case when analyzing the weights in the bottom plot of figure 4.10. The more important second order terms consist of parameters that were seen as highly important as singular terms already. The method we called RF from the original model in DBtune shows a more equal distribution among the parameters, while generally agreeing that the memory-related parameters like *shared_buffers* and *work_mem* matter most. To contrast TPC-C with Wikipedia, the latter workload shows us more dominant parameters as see in figure 4.11. The weight distribution is less equal and the parameter importance methods seem to agree with each other a lot. Interestingly enough, Wikipedia shows us that the parameter *wal_buffers* plays a much bigger role in the DBMS

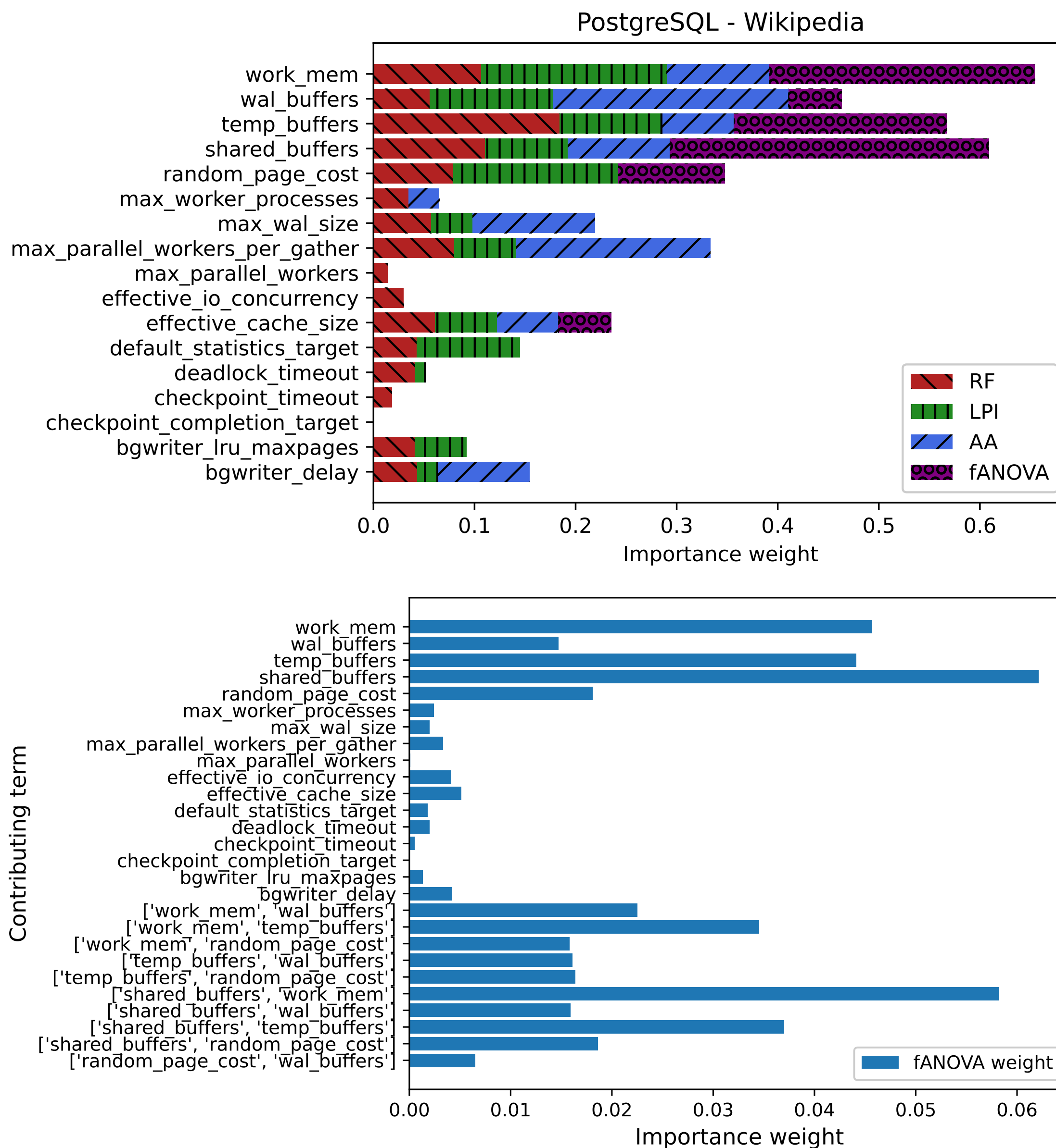



**Figure 4.11:** Feature importance results for PostgreSQL using the Wikipedia workload similar to figure 4.10.

performance in comparison to what TPC-C showed.

To compare with previous literature, we have the paper by Kanellis et al. [19] that has also identified parameter rankings for PostgreSQL. In their study the authors utilized YCSB, which is a different benchmarking tool from the one we used (OLTP-Bench). In YCSB [9], users can specify the proportion of read and write queries they want to use directly, but this means they cannot use well-defined and popular workloads such as TPC-C and Wikipedia like we have done in our experiment. Due to this, direct comparisons are hard to make - not to mention differences in hardware and software setups that we would have to account for.

## Optimization with top PostgreSQL parameters

Similar to what we did for RocksDB, we use the ensemble method to identify the parameters with importance values whose sum is greater than the pre-defined threshold value that we set at 0.2. This leads us to the list of ranked parameters shown in table 4.6. A new search space consisting of these parameters were then used in a new optimization run to get new results, which are shown in table 4.7. The graphs showing the best achieved throughput so far are presented in figure 4.12

| Rank | TPC-C | Wikipedia |
|---|---|---|
| 1 | shared_buffers | work_mem |
| 2 | work_mem | shared_buffers |
| 3 | wal_buffers | temp_buffers |
| 4 | random_page_cost | wal_buffers |
| 5 | max_wal_size | random_page_cost |
| 6 | deadlock_timeout | max_parallel_workers_per_gather |
| 7 | - | effective_cache_size |
| 8 | - | max_wal_size |

**Table 4.6:** The selected most important parameters after studying both workloads for PostgreSQL (TPC-C and Wikpedia). The parameters form smaller search spaces in new optimization runs and the performance gains will be reported.

Just like when we used the initial set of parameters, the improvement gain is once again a lot higher for TPC-C in comparison to Wikipedia. However, we note that we observe a problem in the default performance using the TPC-C workload. Comparing the TPC-C graph in figure 4.12 with 4.9, we see that the default performance is very different (approximately 225 versus 350). This variation is large and signals that we need more repetitions to find out why the default configuration showed lower performance after tuning a smaller set of parameters. Unfortunately due to time constraints, we can only speculate that some type of noise (such as an active running application) may have used up some of the CPU. As we see can in table 4.7, the results from optimization shows that we match the initial improvement by an enormous 239.3% . Instead, we can consider using the default performance value from figure 4.9, meaning that the improvement gain using the refined set of parameters is approximately 38.9%. We thus match the improvement gain by $\frac{38.9\%}{50.4\%} = 77.2\%$ when comparing the initial and reduced set of parameters. This result would make more sense, though the issue we raised should be investigated further with more optimization repetitions.

As for Wikipedia, while we see that we obtain a higher improvement gain using a refined set of parameters, we also should keep in mind that the absolute gains are rather small, meaning that optimizing PostgreSQL for workloads like Wikipedia could be regarded as inefficient depending on the user's resource availability.

| Workload | $\text{Gain}_{Initial}$ | $\text{Gain}_{Refined}$ | (%) of $\text{Gain}_{Initial}$ |
|---|---|---|---|
| TPC-C | 50.4% | 120.6% | 239.3% |
| Wikipedia | 7.0% | 10.3% | 147.1% |

**Table 4.7:** A comparison of the throughput improvements gained from optimizing PostgreSQL for the two workloads we studied after using the initial set of parameters from table 3.2 and refining the set to include only the most important parameters.

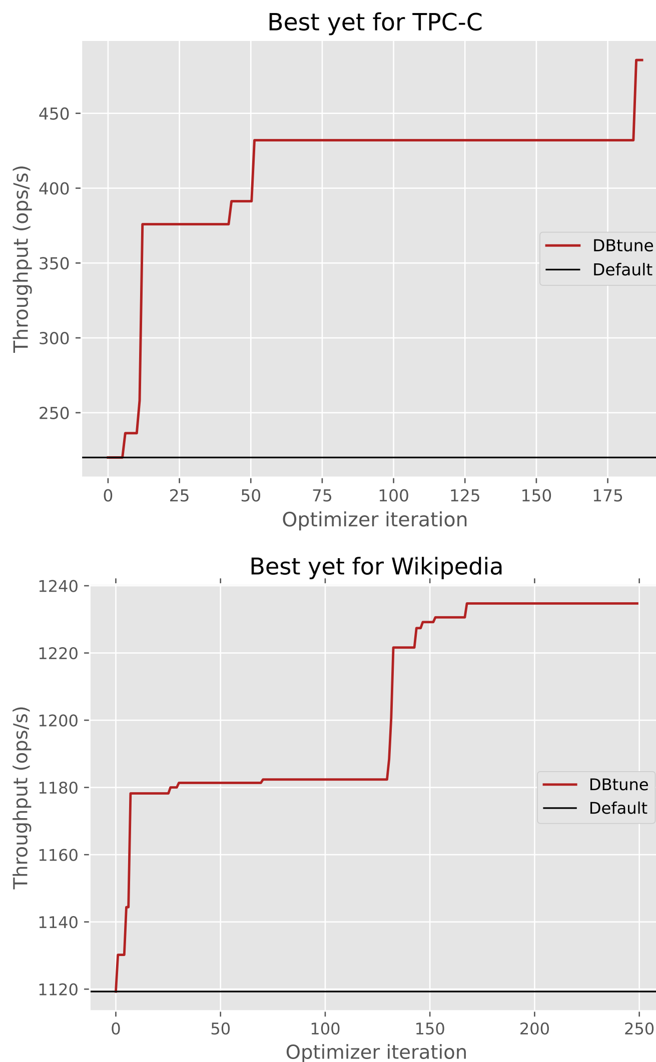



**Figure 4.12:** Plots similar to the ones in figure 4.8 for RocksDB where we show the best obtained throughput so far during optimization for the two PostgreSQL workloads TPC-C and Wikipedia using a reduced set of parameters. Here we do not have have an error area due to lack of multiple repetitions.

### 4.2.4 Neo4j

Benchmarking Neo4j using LDBC SNB to evaluate a configuration proved to be difficult as we could not obtain any reliable results. When using data sizes that were larger than the RAM (i.e. not fitting into memory), the measured overall throughput kept shrinking over time. This was observed in the output that is produced by the benchmark every second, which reports the throughput during the last second and the overall throughput since the beginning. When we used low data set sizes like 3 GB (scale factor of 3) that is lower than the 8 GB of RAM available, we managed obtain a stable throughput over time. However, the problem with using small data sizes is that incoming queries are then trivially handled by the DBMS. The parameters we had chosen were mostly connected to the java virtual machine that is run by Neo4j with heavy focus on memory management. For small amounts of data, these parameters have little to no impact, which means that the stable throughput that we obtained would be consistent no matter how we tuned the configuration parameters of Neo4j. Using larger data sizes as described above, the throughput drop over time also occurred no matter how we tweaked the benchmark set, e.g. by allowing only read queries, disabling complex queries, or changing the number multiple threads.

Whether the issues are related to the benchmark itself or if they are due to limitations in the current versions of Neo4j (and likely, other graph databases) is unclear. In the future we hope that the LDBC benchmark can be demonstrated to work with any size of data that user has regardless of the computer RAM so that we can effectively tune the vital memory management parameters. An alternative would be to explore other types of parameters, though in our case there hasn't been any meaningful online documentation or other references showing how other parameters than the ones we considered impact the performance of a Neo4j instance.

Apart from failure in performing proper evaluations of the performance, our long period of debugging has led to fruitful discussions with both Neo4j developers as well as the active LDBC SNB developers. The problems we faced led to weekly online meetings with both parties involved over a period of two months to discuss issues relating to the benchmark and future work. For example, one of the LDBC SNB developers recommended we try another smaller and unofficial benchmark called *Labelled Subgraph Query Benchmark* (LSQB) [23] that is setup in a similar fashion to LDBC SNB. It executes a set of 9 different queries and returns the response times for each query, which means we do not have the throughput as the objective but rather the latency. An overall performance metric could for example be the geometric mean of the response times obtained for each query. However, we faced another issue where the maximum data size we could use was around 0.3 GB - a very trivial size considering our hardware setup. Any larger and multiple queries return a timeout due to the DBMS taking too long to process them. Even for that data size, we did not see any impact of tuning Neo4j parameters on the performance. Though this can be regarded as reasonable because as discussed earlier, we are mostly tuning memory-related parameters, which in this case the memory is not even close to its limit. An interesting insight we gained from experimenting with LSQB is that changing neo4j license to an enterprise edition allowed us to obtain much faster response times for each query. It enabled us use a data size of 3 GB, but it was still very small. The reason behind the difference between the normal community edition and the enterprise edition was explained to us by both LDBC SNB and Neo4j developers to be due to the difference in how queries are executed. The Neo4j documentation in [28] explains

that the normal edition uses an *interpreted* runtime, while the enterprise edition uses either a runtime that is *slotted* or *pipelined*, essentially consisting of additional optimizations to boost query execution performance.

Another insight we arrived at from our frequent meetings was that initially it was difficult to configure the *Time Compression Ratio* (TCR), a variable in the benchmark to control the scheduling of query executions. As briefly explained in the specification [10], this variable is configurable if a user wants to set a specific target throughput to test. The goal would then be to measure another objective like the latency of each query. Since in our case we are only interested in *thorughput* as the performance metric, we need to have a TCR value that makes incoming queries execute quickly in order to ensure that the system is being stress-tested. In the benchmark setup, we tried using a TCR value that is as low as possible since a lower value means less time between query executions. We also saw that the ratio could not be too low, as that would strangely enough cause severe throughput drops (over time) regardless of the data size we used. It takes more effort for the user (us, in this case) to take into account another parameter that needs to be tuned before benchmarking our system of interest in the first place. After discussing this with the benchmark developers, they have added scripts to tune this parameter beforehand. Finally, an additional highlight from our discussions is that because we want to run an optimization sequence on this system by benchmarking every optimization sample, we needed a convenient way of backing up and restoring data between iterations. The benchmark did not have such a solution but it was also something that the developers added to make LDBC SNB more pleasant to use. Overall, we find that both developers at Neo4j and those maintaining the benchmark are positive and ambitious towards creating ideal testing environments for graph databases.

## 4.3 Summary

Optimizing databases for the right workload can be an expensive process and we have seen differences in how workloads with different mixtures of read and write ratios have impacted the results. The experiments for RocksDB have shown that the improvement we expect from optimizations appears to be higher with a higher read to write ratio in the workload. The identified most important parameters have shown that a smaller search space created using these parameters leads to at least 60% of the performance improvement gained by tuning the initially large set of parameters. What these experiments have shown is that using as few as 5-6 parameters for RocksDB instead of the initial 13, we ultimately face a trade-off. The trade-off is between spending resources during optimization due to requiring more iterations in order to properly explore a larger search space versus using a smaller set of important parameters to decrease these resources and still obtain a 'good enough' improvement. The parameter importance methods in some cases show drastically different priorities, which raises interesting questions on the validity of each one and the overall ensemble method we defined and used (see section 3.1.3).

In contrast to the results for RocksDB, the PostgreSQL experiments have shown that the more heavy-write workload TPC-C has a much larger potential of reaching higher performance values than the read-heavy Wikipedia. Ideally the results we obtained for PostgreSQL should be complemeted with more optimization repetitions, as that would make the results more clear.

Finally, while it was unfortunate to not be able to successfully tune Neo4j for any parameter importance experiments, we are still optimistic about any efforts put into optimizing GDBMS in general. These systems appear to have a lot of configuration parameters that play a vital role in how the GDBMS performs (e.g. Neo4j Java virtual machine) that can be tuned for some specific use case.

# Chapter 5
# Conclusion

Applying and comparing parameter importance methods in the context of database parameter optimization has provided us with lessons about the complexity of the problem as well as future possibilities. The aim of the thesis was to learn about quantifying parameter importance using the different methods we presented and use them in an ensemble method to establish a subset of the parameters containing only the ones that were identified to be most important. We compared using the initial set of the parameters and this subset to determine the difference in improvement obtained during optimization. If the subset manages to obtain a high enough improvement it could mean that a user could use a smaller search space during optimization (hence faster optimization procedure) and still be satisfied with the optimal configuration they reach.

We could say that the results for both RocksDB and PostgreSQL reflect a trade-off between using a smaller search space and achieving a smaller improvement gain from optimization, though in cases where the absolute improvements were very small, this does not seem to apply. Furthermore, we have seen that there may be large differences in the parameter importance results depending on the workload used. We cannot forget that an assumption made about pursuing the parameter importance study was that a user has a workload that they deem is relatively consistent over time. If users find themselves in situations where the database changes often and unpredictably then this type of study does not carry a lot of meaning. Essentially the users would have to find out the most important parameters every time their workload changes. Whether there is a DBMS with parameters that are insensitive to the workload exists or not is an interesting question to pursue. To this end, we might find interesting insights from optimizing graph databases like Neo4j, which is not studied in the literature. In the meantime, these experiments assume workload insensitivity in order to make parameter importance appropriate to study.

As future work, the options that can be further explored are almost endless. Firstly we would recommend that the experiments we performed be validated with many more optimization repetitions. This would add clarity surrounding the issues we faced regarding off performance numbers around the default configuration. Otherwise there are many parts to

this type of study that one can spend more time on to see if results generalize and find new insights: the benchmarking tools, the optimization algorithm itself, utilizing more workloads, comparing the results to even more database systems in each category (SQL, NoSQL, graph), modifying the ensemble method we defined in section 3.1.3, and more. In addition, we would encourage future researchers to dive deeper into graph databases like Neo4j as applying automated methods on them has very few traces in the research community and it could open up lucrative opportunities to make graph databases more widely known as data storage solutions.

# References


[1] Takuya Akiba et al. *Optuna: A Next-generation Hyperparameter Optimization Framework.* 2019. arXiv: 1907.10902 [cs.LG].

[2] Sami Alabed and Eiko Yoneki. "High-Dimensional Bayesian Optimization with Multi-Task Learning for RocksDB". In: *Proceedings of the 1st Workshop on Machine Learning and Systems*. EuroMLSys '21. Online, United Kingdom: Association for Computing Machinery, 2021, pp. 111–119. ISBN: 9781450382984. DOI: 10.1145/3437984.3458841. URL: https://doi.org/10.1145/3437984.3458841.

[3] AutoML. *ParameterImportance*. 2021. URL: https://github.com/automl/ParameterImportance.

[4] Freiburg-Hannover AutoML. *AutoML*. 2021. URL: https://www.automl.org/.

[5] James Bergstra et al. "Hyperopt: a Python library for model selection and hyperparameter optimization". In: *Computational Science & Discovery* 8.1 (July 2015), p. 014008. DOI: 10.1088/1749-4699/8/1/014008. URL: https://doi.org/10.1088/1749-4699/8/1/014008.

[6] A. Biedenkapp et al. "CAVE: Configuration Assessment, Visualization and Evaluation". In: *Proceedings of the International Conference on Learning and Intelligent Optimization (LION'18)*. 2018.

[7] Derek Bingham. *Branin Function*. 2013. URL: https://www.sfu.ca/~ssurjano/branin.html.

[8] Derek Bingham. *Rosenbrock Function*. 2013. URL: https://www.sfu.ca/~ssurjano/rosen.html.

[9] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

[10] LDBC (Linked Data Benchmark Council). *The LDBC Social Network Benchmark (version 0.3.3)*. 2021. URL: https://arxiv.org/pdf/2001.02299.pdf.

[11] DBtune. *DBtune is an AI-powered database tuning service*. 2022. URL: https://www.dbtune.ai/.

[12] Djellel Eddine Difallah et al. "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases". In: 7.4 (Dec. 2013), pp. 277–288. ISSN: 2150-8097. DOI: 10.14778/2732240.2732246. URL: https://doi.org/10.14778/2732240.2732246.

[13] Katharina Eggensperger et al. *Efficient Benchmarking of Algorithm Configuration Procedures via Model-Based Surrogates*. 2017. arXiv: 1703.10342 [cs.AI].

[14] The PostgreSQL Global Development Group. *What is PostgreSQL*. 2021. URL: https://www.postgresql.org/about/.

[15] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. "An Efficient Approach for Assessing Hyperparameter Importance". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, June 2014, pp. 754–762. URL: https://proceedings.mlr.press/v32/hutter14.html.

[16] Frank Hutter et al. "ParamILS: An Automatic Algorithm Configuration Framework". In: *J. Artif. Int. Res.* 36.1 (Sept. 2009), pp. 267–306. ISSN: 1076-9757.

[17] Hypermapper. *Json Parameters File*. 2022. URL: https://github.com/luinardi/hypermapper/wiki/Json-Parameters-File.

[18] ISO. *ISO/IEC AWI 39075 Information Technology — Database Languages — GQL*. 2021. URL: https://www.iso.org/standard/76120.html.

[19] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. "Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs". In: *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020. URL: https://www.usenix.org/conference/hotstorage20/presentation/kanellis.

[20] TPC Kim Shanley. *Origins of the TPC and the first 10 years*. 2021. URL: http://tpc.org/information/about/history5.asp#.

[21] LDBC. *Introduction*. 2022. URL: https://ldbcouncil.org/introduction/.

[22] Ashraf Mahgoub et al. "Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads". In: *Proceedings of the 18th ACM / IFIP / USENIX Middleware Conference*. Middleware '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 28–40. ISBN: 9781450347204. DOI: 10.1145/3135974.3135991. URL: https://doi.org/10.1145/3135974.3135991.

[23] Amine Mhedhbi et al. "LSQB: A Large-Scale Subgraph Query Benchmark". In: GRADES-NDA '21. Virtual Event, China: Association for Computing Machinery, 2021. ISBN: 9781450384773. DOI: 10.1145/3461837.3464516. URL: https://doi.org/10.1145/3461837.3464516.

[24] MongoDB. *What is NoSQL?* 2021. URL: https://www.mongodb.com/nosql-explained.

[25] Luigi Nardi, David Koeplinger, and Kunle Olukotun. "Practical design space exploration". In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2019, pp. 347–358.

[26] Neo4j. *Neo4j Operations Manual*. 2022. URL: https://neo4j.com/docs/operations-manual/current/performance/.

[27] Neo4j. *Neo4j Performance Tuning*. 2022. URL: https://neo4j.com/developer/guide-performance-tuning/.

[28] Neo4j. *Query tuning*. 2022. URL: https://neo4j.com/docs/cypher-manual/current/query-tuning/.

[29] Neo4j. *What is a Graph Database?* 2021. URL: https://neo4j.com/developer/graph-database/.

[30] Keren Ouaknine, Oran Agra, and Zvika Guz. "Optimization of RocksDB for Redis on Flash". In: *Proceedings of the International Conference on Compute and Data Analysis*. ICCDA '17. Lakeland, FL, USA: Association for Computing Machinery, 2017, pp. 155–161. ISBN: 9781450352413. DOI: 10.1145/3093241.3093278. URL: https://doi.org/10.1145/3093241.3093278.

[31] RocksDB. *RocksDB Basics*. 2022. URL: https://github.com/facebook/rocksdb/wiki/RocksDB-Basics.

[32] RocksDB. *RocksDB Benchmarking Tools*. 2021. URL: https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[33] RocksDB. *RocksDB Users*. 2022. URL: https://github.com/facebook/rocksdb/blob/master/USERS.md.

[34] Thomas Schmied et al. *Towards a General Framework for ML-based Self-tuning Databases*. 2021. arXiv: 2011.07921 [cs.DB].

[35] Bobak Shahriari et al. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: vol. 104. 1. 2016, pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.

[36] SNIC. *SNIC Science Cloud*. 2022. URL: https://www.snic.se/allocations/ssc/.

[37] TPC. *TPC Benchmark C: Standard Specification*. 2022. URL: http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[38] Dana Van Aken et al. "Automatic Database Management System Tuning Through Large-scale Machine Learning". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. 2017, pp. 1009–1024. URL: https://db.cs.cmu.edu/papers/2017/p1009-van-aken.pdf.

[39] Ji Zhang et al. "An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning". In: June 2019, pp. 415–432. DOI: 10.1145/3299869.3300085.

# Appendices

**EXAMENSARBETE** Efficient optimization of databases using parameter importance methods.
**STUDENT** Osama Eldawebi
**HANDLEDARE** Luigi Nardi (LTH)
**EXAMINATOR** Pierre Nugues (LTH)

# Effektiv optimering av databasparametrar

POPULÄRVETENSKAPLIG SAMMANFATTNING **Osama Eldawebi**

Optimering av databasparametrar har uppmärksammats allt mer under den senaste tiden i samband med att datorkraft ökar och blir mer lättillgänglig. Detta arbetet gick ut på att undersöka ifall det går att göra optimeringen av databasparametrar mer effektiv genom att endast fokusera på de mest viktiga parametrarna.

Inte sällan har databaser ett tiotal eller hundratal parametrar som påverkar olika delar av databasen (t.ex. hur minneseallokering ska gå till, val av komprimeringslalgoritm, och hur ofta databasen säkerhetskopierar) som i sin tur påverkar databasprestandan. Arbetet som en databasadministratör utför till att försöka ställa in bra värden i parametrarna kan underlättas genom att utnyttja optimeringsalgoritmer inom maskininlärning. En specifik metod, *Bayesian optimization*, används ofta inom forskingsämnet som en kostnadseffektiv algoritm för att undersöka och utvärdera olika parametervärden i syfte att hitta värden som leder till bäst prestanda. Eftersom inte alla parametrar bidrar till prestandan på samma sätt kan det finnas vissa som har lite betydelse och därför kan bortses från optimeringen. Att hantera ett färre antal parametrar innebär mindre kostnad eftersom algoritmen inte behöver undersöka lika många olika kombinationer av värden men samtidigt kan det innebära att vi inte når samma prestanda som om vi hade försökt optimera med fler parametrar.

I detta arbetet har prestandan definierats som antalet transaktioner en databas kan hantera per sekund (s.k. *throughput*). Olika parameter importance-metoder har använts för att kvantifiera bidraget varje parameter har på prestandan och därmed utvärdera vilka parametrar är mest viktiga och vilka som kan uteslutas från optimeringen. Totalt har fyra metoder undersökts som bygger på olika beräkningsmetodik. En övergripande *ensemble* metod har introducerats som ett sätt att slå samman resultaten och komma fram till ett slutgiltigt uppsättning av de parametrar man anser är mest viktiga. Vad vi åstadkommer med detta är att framtida optimeringsrutiner kan spara på resurser i och med att man nu kan endast fokusera på de viktiga parametrarna.

Resultatet visar att genom att kapa ner antalet parameter med hjälp av ensemble-metoden för både databaserna RocksDB och PostgreSQl har man kunnat nå cirka 60-70% av den förbättringen man annars hade fått genom att inte utesluta någon parameter. Det visar sig även att vissa parametrar har en större eller mindre roll beroende på den typen av arbetsbelastning som databasen tar emot. Dessutom visade det sig att man kan optimera RocksDB och PostgreSQL bättre för vissa arbetsbelastningar än andra. Fler repetitioner och mer hänsyn till arbetsbelastningen i databasen hade önskats som framtida arbete för att validera och generalisera resultaten.