LTH — LUND UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
Degree Project in Machine Learning

Workload detection and Continuous Automatic Bayesian Optimization in Database Management Systems

Authors
Jonas Boström(jo5773bo-s)
&
Viktor Olsson(vi1146ol-s)

LU-CS-EX: 2022-50
ISSN 1650-2884

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-50

Workload detection and Continuous Automatic Bayesian Optimization in Database Management Systems

Arbetsidentifiering och Kontinuerlig Automatisk Bayesisk Optimering i Databashanterare

Jonas Boström, Viktor Olsson

Lunds Universitet

Degree Project in Machine Learning

Workload detection and Continuous Automatic Bayesian Optimization in Database Management Systems

---

Authors

Jonas Boström(jo5773bo-s) & Viktor Olsson(vi1146ol-s)

Email: jonas.oskar.bostrom@gmail.com & violsk@protonmail.com

Supervisor

ASSOCIATE PROFESSOR IN COMPUTER SCIENCE

Dr. Luigi Nardi

Examiner

ASSOCIATE PROFESSOR

Christoph Reichenbach

Abstract

The goal of this thesis is to investigate the possibility of multi-workload optimization in Database Management Systems, and the workload detection implied by it. Data was collected for the popular workloads TPC-C, CH-benCHmark and Wikipedia for two different types of metrics. The first was hardware-based metrics, consisting of values such as CPU and memory usage, which was tested using several detection methods. It was found that hardware-metrics excelled in separating data for the chosen workloads in non-optimizing circumstances, and in optimized situations. The second group of metrics were query-based metrics, consisting of the query types that were executed by the Database Management System. This metric is tested with the use of the DBSCAN clustering algorithm. This method could also separate all workloads that were chosen, without a misinterpreted workload shift. A benchmarking framework was successfully designed to allow for multi-workload testing and data aggregation. The performance gain when optimizing using this project did not seem to match the optimizing performance obtained during testing within the single-workload framework. This was found to most likely be due to performance degradation in the storage, caused by the storage drive being almost full.

Keywords: databases, postgres, bayesian optimization, dbms, throughput, optimization

Preface

With this thesis finished, we conclude our studies at LTH. The project was executed during the winter-spring of 2022. We would like to thank our supervisor Luigi Nardi for providing excellent guidance and knowledge for the past six months. As well as the team at DBtune for being a great resource to have at hand, with particular thanks given to Muhammed Umair.

Finally, thanks to Christoph Reichenbach for taking on the role of examiner for this thesis.

To see author contributions view appendix 6.8 table 45.

Table of Contents

1 Introduction **1**

1.1 Research Questions 1

2 Background & Related Work **2**

2.1 Background 2

2.2 Theory 2

2.2.1 Terms 2

2.2.2 Regarding Objectives 3

2.2.3 Bayesian Optimization 4

2.2.4 Expected Improvement 5

2.2.5 Random Forest 6

2.3 Workload Detection 6

2.3.1 Clustering 6

2.3.2 Classification 6

2.3.3 Anomaly Detection 7

2.4 Tools 8

2.4.1 PostgreSQL 8

2.4.2 OLTP-Bench 9

2.4.3 HyperMapper 9

2.4.4 DBtune 10

2.4.5 rsync 11

2.4.6 Data Collection Tools 11

2.4.7 Amazon Web Services 12

2.5 Related Work 12

3 Methodology **14**

3.1 Setup 14

3.1.1 Hardware 14

3.1.2 DBtune Implementation 14

3.1.3 Workloads 16

3.2 Data Collection 19

3.2.1 Query Collection 19

3.2.2 Hardware Collection 20

3.3 Data Processing 25

3.3.1 Query Statistics Processing 25

3.3.2 Hardware Statistics Processing 25

3.4 Multi-Workload Testing 35

3.5 Workload Detection 35

3.5.1 Detection methods 36

3.5.2 Performance Measurement 41

3.5.3 Implementation for queries 43

4 Results **45**
4.1 Experimental Settings 45
4.1.1 Optimal Configurations 45
4.1.2 Multi-Workload Optimization 45
4.1.3 Queries 46
4.1.4 Hardware Statistics 46
4.2 Analysis 47
4.2.1 Optimal configurations 47
4.2.2 Multi-Workload Optimization 50
4.2.3 Hardware Statistics 51
4.3 Workload Detection 63
4.3.1 Query-based Statistics 63
4.3.2 Hardware Statistics 65
4.4 Discussion 70
4.4.1 Regarding credibility of results 70
4.4.2 Optimal Configurations 70
4.4.3 Multi-workload Optimization 70
4.4.4 Query-based Statistics 71
4.4.5 Hardware Statistics 72

5 Conclusions & Future Work **74**
5.1 Conclusions 74
5.2 Future Work 74

6 Appendix **81**
6.1 Knobs search space 81
6.1.1 Knobs with priors 81
6.1.2 Knobs without priors 82
6.2 CH-benCHmark 83
6.3 Hardware metrics 83
6.4 Multi-workload Optimization 86
6.5 Hardware analysis 86
6.6 Hardware Detection 95
6.7 Code Listings 102
6.8 Author Contribution 113

1 Introduction

In a world further impacted by digitalization, the demands for servers have risen over the past decades. As a result, companies spend more money maintaining and expanding their server infrastructure, which has an economic and environmental impact. To minimize the impact of these, the server performance can be optimized by modifying the server configuration. If tuned in just the right way, this means that some optimal configuration will be found, which could potentially lead to a large increase of user requests being handled by one server, therefore making fewer servers necessary.

To optimize a server, you must have an objective to optimize for, such as Throughput, Latency or CPU-usage, as well as some server configuration parameters that can be modified and has an impact on the servers performance.

A key issue that defines server optimization is that each evaluation of the objective is very costly. The server can be referred to what's known as a "black-box function", a function that we can give inputs and be given some output, but without knowing the analytical form of the function. Therefore, no gradient is available, and since each evaluation takes a significant amount of time to test, optimization of the server which requires many evaluations takes a long time.

For this reason, it is naturally interesting to optimize the server with some method that requires the fewest number of iterations to obtain good performance, since evaluating an input takes quite a long time. One method that is capable of efficient server optimization is Bayesian Optimization (BO), which attempts to gather data in such a way to make sure that each iteration provides a mix between gathering data and achieving high performance.

Unfortunately, since Bayesian Optimization assumes a never-changing black-box function, it fails to optimize correctly if the hidden function inside were to change. In server optimization, this can be represented as a workload shift, meaning that the tasks that are sent to the server has started varying to such a degree that a new optimal configuration is required. Therefore, it can be beneficial to identify when the workload changes.

To combine the aspects of Bayesian Optimization with workload detection is a challenging research project, with potentially great results from an environmental and economical perspective. This thesis will tackle this problem described by using single-objective BO and workload detection using database and hardware metrics.

1.1 Research Questions

- Can workloads be adequately characterized by hardware statistics?
- Can workloads be adequately characterized by query statistics?
- How can workloads be classified automatically?
- What is the possible performance gain from continuous optimization?

2 Background & Related Work

2.1 Background

As a problem, database optimization has been a topic of discussion since at least the early 1990s [25]. Since then, the focus areas have partially shifted away from lower-level query-optimizations, to higher-level automatic server configuration optimization, which can be seen with the recent rise of papers that looks into finding the best server configuration [1][55][30]. Before this recent uptick in automatic optimization, most DBMS were configured by human experts in the field. Unfortunately, as time passes, DBMS has grown more complicated and is now at a point where hundreds of knobs can be modified, with many of them being co-dependent on each other [1]. The complexity of the problem only compounds when you add the fact that each specific hardware configuration greatly impacts the optimal configuration [3]. Assuming this trend holds, it will only become more important to have reliable tuning software. For this reason there has been a need in the industry to automate this process.

While many methods such as reinforcement learning and random search are capable of finding optimal configurations for multi-dimensional problems, they suffer from the same deficiency: They require a significant number of samples to obtain reliably good performance [3]. A solution to this issue comes in the form of Bayesian Optimization, which will be further detailed in section 2.2.3. Put shortly, this method generates a simulated model of the black-box function, and then probes it for data points with a probability of achieving great performance. This allows it to maximize the information and performance gain from each iteration.

This method has grown very popular in recent years, being used in many different areas, from server optimization to robotics to computer vision [7].

As mentioned previously, this method does not take a modification of the underlying workload into account, which is one of the key aspects this project seeks to solve. The idea is to identify workloads based on some feature vector that is collected during a workload execution. While the idea of workload detection in DBMS contexts is relatively new, from a more generic classification and clustering perspective, algorithms such as k-nearest neighbour have existed since at least the 1950s [19]. Due to the fact that this area of research has existed for over half a century, many different algorithms and ideas of clustering have been invented over the years, providing a wide range of tools for the DBMS workload detection problem.

2.2 Theory

2.2.1 Terms

- **transaction**: A transaction contains several tasks, each of which is performed on the database. An example would be the placement of a new order. These consist of one or more queries.

- **query**: A request send to a DBMS that is related to the retrieval or storing of data. These typically consist of a command, such as SELECT, INSERT or UPDATE.
- **terminal**: An operator that sends transaction requests to the database.
- **warehouse**: Each warehouse contains one part of the data available in the database. When data is retrieved or written by a transaction, it interacts with one or multiple warehouses.
- **black-box function**: The function that the system will optimize for. Has some input, and provides some output but no other information is known. Hereafter referred to as BBF.
- **surrogate model**: An estimation of the true model hidden inside the BBF.
- **iteration**: An iteration is the act of running a workload in OLTP-Bench one time.
- **repetition**: A repetition is the act of running the whole optimization program. Essentially, this consists of a large number of iterations.
- **hardware utilization**: Hardware utilization is referring to hardware related metrics, such as "bytes read" from the disk and CPU-usage.
- **data point**: A single measurement value for some metric, like CPU percentage.
- **OLTP**: On-Line Transaction Processing (OLTP) transactions. Typically quite small and easy for the DBMS to execute.
- **OLAP**: On-Line Analytical Processing (OLAP) transactions are more analytical and complex then OLTP transactions. These types of transactions require far more processing power to execute. For this reason, far fewer of these can be performed per second as compared to OLTP transactions.

2.2.2 Regarding Objectives

Since a BBF is defined to be optimized after an objective, the choice of objective is important. Some possible objectives include

- Energy Efficiency
- Throughput
- Average latency
- 99th percentile latency

Since this project uses single-objective optimization, some objective must be chosen to evaluate database performance. For most research in database performance, either throughput or latency is used [20]. For the purposes of this thesis, throughput will be the optimizing objective, which is defined as the number of transaction requests processed per second (TPS).

2.2.3 Bayesian Optimization

Bayesian Optimization (BO) is a method used for the efficient optimization of a BBF. Put shortly, BO intelligently explores the available search space, so over time, the best configuration can be found. This exploration is done by utilizing what is called a surrogate model. The surrogate model is the statistical machinery, giving information on how likely a configuration is to achieve a throughput value. The surrogate model achieves this by using something called the prior. The prior contains information about the results from using previously tried configurations. The surrogate job is then to incorporate this information to create a better estimate of the BBF. By combining the surrogate with an acquisition function one achieves BO. The acquisition function's job is to decide the next configuration to be explored. See for example [21] for a further description of these subjects.

The surrogate model is highly useful in resource-intensive applications. By using a surrogate model one can avoid calculating the real BBF. One would search the surrogate model for the next optimal value and once found the BBF would be evaluated using the parameterization of the found value. Using a surrogate model has great importance when the BBF is expensive to evaluate, i.e when the BBF is resource-intensive somehow. For example, in database optimization, one test run can take several hours.

There are many types of surrogate models, with two such models being the Gaussian Process [8, page. 303] and Random Forest, maybe the most renowned. The key aspect of the surrogate model is that it should follow the Bayesian model, i.e incorporate information as it becomes available in order to improve the surrogate in resembling the real function.

To acquire the believed optimal points from the surrogate, a utility function is needed. In BO, the utility function is called the acquisition function. The acquisition function searches the surrogate model to maximise the likelihood of finding the next optimal parameterization. This is important since just taking the believed next optimal point can get the search stuck in a local minimum. Thus, both exploitation and exploration are needed. Exploitation is when the acquisition function explores the believed best point, i.e evaluates it by running the BBF. Meanwhile, for the exploration part, the goal is to look into unexplored areas where no evaluation has been executed on the BBF. Shahriari et. al. identify four acquisition function categories [46].

1. Improvement-based Policies: In this category, the functions assign high values to parameterizations that are expected to improve the target value, which is usually set as the latest known optimum. This category includes functions like the probability of improvement and the expected improve-

ment (EI). These functions utilise statistics, such as the expected mean to propose the next point to be explored. By utilizing the functions in this category the user has some statistical foundation that the next point to be explored is the expected best. These functions are usually exploiting too heavily, to remedy this a heuristic is often used. The heuristic often used, is to give the acquisition function some probability of exploration, instead of exploitation. Once exploration is chosen, some random point is selected for evaluation by the BBF.

2. Optimistic Policies: These acquisition functions give very optimistic estimates of the values in the surrogate. For example, the acquisition function, upper confidence bound, take the upper confidence interval of the surrogate for each parameterization. Utilizing such exploration tactics favours uncertain points, which could delay the exploration of a minimum. The benefit is that some sort of exploration is built within the method itself.

3. Information-based Policies: Here entropy functions come into play like entropy search. These functions also exploit quite heavily.

4. Portfolios: Several acquisition functions in a portfolio. Some meta-function selects the believed best acquisition function for the current selection of the believed optimal parameterization.

From the above list, one understands there are many choices of acquisition functions and these fall into different categories. There is no defined best function, a function is usually selected according to previous trials or heuristics. Though there are some functions which seem to perform well in many test scenarios, such a function is, for example, EI [46].

2.2.4 Expected Improvement

Expected improvement (EI) models the expected gain from choosing parameterization x. Expected improvement comes from the improvement-based policies acquisition functions category, and as previously described the target value w_{best} is usually the currently known minimum for the BBF. This is also the case for EI. The function $I(x)$ refers to the actual improvement one would get using parameterization x, see equation 1, where w represents the return value from using such a x. Since the BBF is an unknown function one can **not** know what w will be produced from using x. Therefore the improvement must be evaluated as the *expected improvement* (EI), see equation 2 and [26, page, 471]. Where the expected value of w using x is derived from the surrogate.

$$I(x) = max(w_{best} - w, 0) \quad (1)$$

$$E[I(x)] = E[max(w_{best} - w, 0)] \quad (2)$$

2.2.5 Random Forest

Random Forest is a machine-learning method capable of regression as well as classification. It consists of a large amount of decision trees, where each decision tree traverses its leaves to classify some input. After an input has been given to a random forest, this input is then distributed to each tree. All the individual tree outputs are then ensembled, meaning they are combined and averaged, to produce a final output. The benefit of using a random forest is that the model tends to avoid overfitting due to the "wisdom of the crowd" effect. Due to the way Random Forests are constructed, they can use continuous, discrete and any other type of features that may be desired. Since many knobs in DBMS:s are discrete, Random Forest fits database optimization well.

2.3 Workload Detection

A key aspect of this project is workload detection. This consists of gathering input data, and then matching it to some pre-existing data clusters. Primarily the matching techniques are divided into two types of algorithms: clustering and classification algorithms. We provide a general description of these algorithms in the rest of the section.

It is important to note that the description below of both of these different algorithms are only general implementations. There exists many algorithms that break these rules in some way.

2.3.1 Clustering

Clustering refers to the idea of classifying data points by dividing them into a number of groups, i.e clusters. This method is unsupervised, meaning that no prior data is required to train the model, and instead the model solves this itself. Depending on the exact algorithm the prerequisite knowledge varies, but in general they need very little information. A very common unsupervised algorithm, DBSCAN, only requires a single hyperparameter, the *eps*, to be defined, which specifices how far apart data points can be while still being considered in the neighbourhood of one another [17].

Another common, and very simple, clustering algorithm is k-means clustering, which requires knowing the number of clusters to divide the data into before doing so unsupervised. The methods works by dividing the number of data points n into k clusters, in such a way that within any one cluster the total euclidean distance is minimized. The downside of this method is that it relies on clusters being of similar size and shape, as well as knowing the number of clusters beforehand.

2.3.2 Classification

Unlike clustering, classification algorithms require some information about each type of data, so it can then classify exactly which one a new data point belongs to. Classification methods do not change overtime, and therefore need to be

completely predetermined upon use. These tend to be supervised methods, and therefore need some labels for what data they are classifying. Classification algorithms have many different implementations of varying complexity, from simple Euclidean distance to advanced neural networks that are trained to detect microscopic features in the input data. The fact that these tend to be supervised means they are unsuitable for this project, due to the possibility of workloads shifting over time in real-world scenarios it cannot be assumed that they stay constant over time.

2.3.3 Anomaly Detection

There are many possible candidates for workload detection in the field of anomaly detection. To give a few examples, there are the more traditional functions like: CUmulative SUM (CUSUM)-chart, Pearson's Chi-squared test (χ^2) and Mutual Information (MI) [6, Chapter 1] [28]. There are also other more complex modern functions, such as X-iForest [18] which utilizes an isolated random forest and the X-mean as a clustering method. Other modern examples are biological algorithms [38] and deep learning models [54]. The ones investigated in this thesis are CUSUM-chart and MI.

CUSUM chart is an easy to implement algorithm for detecting a diverging mean. There are many implementations of CUSUM chart and the authors of this thesis follow Cuadra-Sánchez and Arcails' definition as defined in chapter 1, [6]. In Arcails' implementation, the mean and standard deviation (STD) is usually derived from previous observed data. These values will then work as a reference point for further anomaly detection. For example, the threshold value is the allowed divergence from the specified reference mean, and the threshold value itself is usually set to be some factor of the reference STD. The threshold value itself is then used for the accumulation of the detection bounds. The detection bounds contain the accumulated breaches of the threshold, with a lower and an upper bound. The processes then function as the following: When a value is received it is compared against the lower and upper bound. If the new value is below the mean minus the threshold, then the lower bound value will increase and conversely an upper bound value will increase when a given value is above the mean plus threshold. These two bound values then continue to accumulate for every divergence and decrease for every correction, see equation 3. Detection is then made when the bound values have diverged too much, which is a hyperparameter.

$$\begin{aligned} lower_{bound}(x+1) &= max(0, (mean - threshold) - next_{value} + lower_{bound}(x)) \\ upper_{bound}(x+1) &= max(0, next_{value} - (mean + threshold) + upper_{bound}(x)) \end{aligned} \tag{3}$$

Mutual Information (MI) comes from the field of information theory and in simple terms tries to set a value on how similar two distributions are. MI is based on entropy, which is the uncertainty of a random variable (RV). If a RV

has maximum entropy it is equivalent to knowing nothing about the variable, i.e completely random, and the resulting sampling could be whatever. If the entropy is 0 then everything is known, it is known exactly what each sample will be at every point [49][Chapter. 1], from here on H(X) will define the entropy of RV X. If we then ask, how random is RV X given that one knows Y, $p(X|Y)$, this is what MI tries to describe. How much information is gained from knowing RV Y when determining X or how **equal** are the two distributions. If the two RVs are independent then MI is 0, opposite a large value negative or positive indicates high dependence. If one defines MI by entropy then one has the following formula $I(X;Y) = H(Y) - H(Y|X)$.

Continuous MI is not as simple as discrete MI, usually there is no analytical formula and an estimate must be used. One method to estimate MI between continuous RVs is to use the k-nearest neighbour method. If one takes the averages for each point to its k neighbours then the distances can discern information about the distribution of the RV. If this distance stays the same for all points it is hard to say anything about the distribution except that it is random, if the distances are closer in certain regions this indicates a higher probability mass in that zone, i.e this is an estimate for $H(X)$. Similarly one could estimate $H(Y)$ and now MI can be estimated as $I(X;Y) = H(Y) - H(Y|X)$. This is usually not how it is done, since this can create systemic bias, but this proves the concept. The actual k-neighbours algorithm as discussed by Alexander et. al. uses something more similar to binning, but this will not be touched further upon, to read further see [28].

2.4 Tools

The project used the following four tools to handle BO, PostgreSQL [41], OLTP-Bench [4][15], HyperMapper [33] and DBtune [7]. PostgreSQL is the DBMS, OLTP-Bench is the benchmarking tool, HyperMapper is the optimizer and DBtune the wrapper that combines all the tools. In the following sections these tools will be described further.

2.4.1 PostgreSQL

PostgreSQL [41] is the database software used for this project. A database is described by oracle as ”an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a DBMS.” [11]. The DBMS used in this project is PostgreSQL, which is a relational DBMS initially released in 1996 and implements the SQL [5] format. Since its creation it has become one of the most popular DBMS in the world. Currently it is the fastest growing one and in the top five DBMS overall [12]. Since its release, PostgreSQL has become known for being highly reliable [22], as well as being one of few open-source DBMS in a world populated by closed-source DBMS such as MySQL, Oracle and MongoDB.

PostgreSQL has a large number of knobs that can be used to tune performance of the database, with at least 169 knobs available to tune [55]. These knobs are

of many different types. Some are in the form of a boolean, whilst others require some discrete or floating point value. Further, these knobs are divided into so called "Non-restarting" and "Restarting" knobs. This specifies whether or not the database must be restarted for any change to the knob to take effect, with the majority of knobs being non-restarting.

2.4.2 OLTP-Bench

OLTP-Bench is an open-source testbed for database performance and exists under the Apache 2.0 license [15]. Among OLTP-Bench's workloads are some of the most universally utilized database tests, such as TPC-C [31]. One key feature of OLTP-Bench is the ability to easily customize workloads through a configuration file. In this file, the scale-factor and number of terminals can be modified, which affects the size of the database and number of clients sending transactions to the DBMS respectively. As well as these, OLTP-Bench also allows you to set the distribution of the transactions in the workload, which is set by assigning a weight to each transaction type. The weights resembles the likelihood that the transaction will be selected and should sum to 1.

Each workload in OLTP-Bench consists of an initial warm-up of the database, followed by an execution. The "warm-up phase" is OLTP-Bench's efforts to accurately simulate a long-running DBMS, where the components are "warmed up". Otherwise the performance at the beginning of a workload could differ significantly from the remainder of the workload execution, so a warm-up is necessary to obtain a reliable value for TPS.

As of 22-8/2021, OLTP-Bench has been deprecated and replaced with BenchBase [4]. BenchBase was intended to be this projects benchmarking software, but testing found that it was unfortunately too unstable and bug-ridden to be used. The authors of this project have therefore chosen to stick to OLTP-Bench, as it is a well-tested and known benchmarking software.

2.4.3 HyperMapper

HyperMapper is an open-source program developed by Luigi Nardi et. al., and is available under the MIT License [34]. The program is describe as " a multi-objective black-box optimization tool based on Bayesian Optimization ", where the black-box function (BBF) is defined by the user. Whilst the program is based upon Bayesian Optimization, it is capable of other methods, such as Evolutionary Optimization and Local Search [35]. The latter two options are meant to be used in situations were the BBF isn't particularly hard to evaluate, something which does not apply to this project. Aside from the BBF, the user also needs to define the values of different knobs that HyperMapper will navigate in order to find the optimal configuration. Henceforth this will be referred to as the search space.

An optimization in Hypermapper contains two primary steps. First, a Design of Experiment (DoE) phase where a number of configurations in the search space are sampled. This is done to build up the optimizer's knowledge of the

search space. Afterwards, the optimization phase starts where HyperMapper uses its knowledge to update the surrogate model, which can be either a Gaussian Process or a Random Forest. The chosen surrogate model is then examined by the acquisition function. The acquisition function produces the configuration with the largest Expected Improvement. This configuration is then sent to the BBF to obtain the actual throughput. This second phase is repeatedly done until the desired amount of iterations has been reached.

While other automatic BBF optimizers exist, HyperMapper is very flexible and is one of very few that is capable of handling prior knowledge, a concept that is vital for retaining information when an already optimized workload is detected.

For Bayesian Optimization HyperMapper supports user-defined a priori knowledge [33], something that can be used to retain information when previously optimized workloads are detected. For the a priori, Hypermapper accepts a list of probabilities for each parameter, and then uses a beta-distribution to truncate these. Note that HyperMapper is considered robust regarding priors, and even if very bad values are given, Hypermapper is capable of recovering and finding an optimal configuration [33].

HyperMapper also supports non-continuous values, such as discrete, boolean and ordinal parameters. This is very helpful for server configuration tuning, as many knobs that can be tuned are not continuous.

2.4.4 DBtune

DBtune is a project built around OLTP-Bench and HyperMapper. DBtune creates a BBF that updates the server configuration before running an iteration of OLTP-Bench. The resulting throughput is then fed to HyperMapper, which uses this updated information to provide the next recommended server configuration. DBtune also provide a number of options, such as the number of optimization iterations to run, what optimization model to use, which knobs to optimize and much more.

DBtune also solves an important aspect of using OLTP-Bench for database optimization. As OLTP-Bench wasn't designed for continuous server optimization or for multiple consecutive iterations. One issue that occurs is due to the fact that OLTP-bench continually writes data to storage, leading to an increase in the database size. As the database grows, it also loses performance in many metrics, including throughout. Over time, this decrease in performance means that optimizing a database based on throughput becomes difficult and unreliable as the optimizer is unable to tell that a configuration encountered in a later iteration with lower throughput may actually be far superior than one encountered early on in the repetition when the database size was smaller. To deal with this, rsync is used within the DBtune BBF in conjunction with OLTP-Bench in such a way that after each iteration the database is reset into its original state. By doing this, performance is maintained over multiple iterations of DBtune. It could be argued that restoring the database to maintain performance is unrealistic, but so is the massive amount of extra storage OLTP-Bench would require over

multiple iterations, as well as the significant performance drop. Since this project is limited to using OLTP-Bench, rsync must be used.

2.4.5 rsync

Rsync is an well-known utility program provided by default in the Ubuntu distro [10] Rsync is used to generate a backup of some source folder into a destination folder. Whenever desired, rsync is then capable of comparing the two directories and replacing files that differ in the source directory with the equivalent file in the destination. For this project, rsync is used to create a back-up folder of the database. This directory is then used by DBtune to restore the database after each iteration. The reasoning for why this is done can be found in section 2.4.2.

Whilst any program with a similar functionality could be used to reset the database, this one was picked due to being a standard inclusion in most Linux distributions.

2.4.6 Data Collection Tools

There are many different collection tools for PostgreSQL, too numerous to explain in detail. A few of these were explored but remain unused. A list of many tools including the explored ones can be found here [2]. Some of the items from the list are pgmetrics [40], pgSCV [29] and pgwatch2 [27]. These tools extract similar information, with the difference between them being how the end user is presented the information and how the tool is set up. pgmetrics uses a command line interface to set up output files, connections to the database and so on. The designated output file can be text, json or csv. pgSCV uses configuration files instead of directly using the command line like in pgmetrics. The information is returned directly to the calling process and displayed in the Prometheus format. pgwatch2 lets the user use a graphical interface to look at data. After setting up the configuration files the user directly interacts with this graphical interface in the web browser to collect and download information into csv files. The type of information these tools gather is very similar. Most information is collected by querying PostgreSQL's own collection tools or log files which can be activated by modifying PostgreSQL configuration files. Other information, such as hardware utilization, can optionally be collected at a system-wide level. Collecting metrics provided by the database will naturally affect the overall performance of the DBMS, but it seems like not too much. For example, the statistics collector [48], "the statistics collection shouldn't take more than a few percent of CPU time" [23], according to a quote on Oriely. It is hard to find a definitive answer since PostgreSQL does not provide any. Since these three explored tools remain unused in the project, no further time will be spent investigating the performance impact of these.

What most of these tools have in common is that they rely on the data collection module pg_stat_statements, which is one of the most widely used modules in PostgreSQL. This module tracks all queries that are executed by

the database and records a wide variety of information regarding these, such as the number of times a query is executed, the average latency and much, much more. This module is the one that is relied upon to collect data about queries in this project. Whilst the activation of this module will result in some performance impact, it is generally agreed upon to be negligible, using an extra 1% of CPU-time [39].

Another tool is psutil [45]. psutil is a python module which provides statistics from over 50 different metrics related to hardware utilization, including CPU, virtual memory, swap memory, disk, network and more and is the go-to tool for hardware statistics gathering in the Python world. This tool can produce both system-wide and process isolated utilization statistics. It's noteworthy to mention that psutil seems to mainly take its data from the registers provided by the OS, which contain the information that is then forwarded to the user. A downside of using psutil is the lower abstraction level. The end-user must handle process identification numbers and the collection of actual metrics by themselves. Furthermore, the psutil module is not platform-independent and many collection methods depend on the underlying operating system used.

2.4.7 Amazon Web Services

Amazon Web Services (AWS) is a system created by Amazon that provides cloud computing services. For this thesis, the Amazon Elastic Compute Cloud (EC2), provides access to many different types of servers, depending on the users' demands. Each server allocated to a user is referred to as an "instance". These instances can be configured in a multitude of ways, such as changing the Operating System, type of storage, amount of memory and CPU model. The user can launch as many of these instances as they want. Each instance can be accessed through Amazon's EC2 Instance Connect, or via ssh, the latter of which is used for this project. EC2 provide two main storage alternatives in the form of Elastic Block Storage (EBS) or dedicated storage. The EBS is virtualized, which means it's not stored locally on the allocated server instance, and instead is stored on one or more devices. The other type of storage is dedicated storage, where the data is stored on one single, non-virtual device directly connected to the server itself.

2.5 Related Work

Whereas there have been multiple previous projects that implement automatic database tuning, doing it from a continuous perspective is something that seems entirely new. The Ottertune paper [1] does something that is somewhat related to continuous tuning, but isn't quite the same. In the paper, they utilize previous tuning information to make future optimization efforts more efficient. While this does mean that data is collected and stored to improve optimization, the area of improvement is spread over future optimization sessions on new hardware, rather than benefiting an already optimized server.

Whilst this project has opted to investigate hardware-utilization and query-based statistics to distinguish workloads, the Ottertune paper has opted for a different approach. The data that is collected is instead based on internal DBMS run-time metrics, whilst this project uses query-statistics and hardware-utilization.

In terms of DBMS workload detection, there was a paper written previously that investigate the detection of TPC-C against CH-benchmark workloads [56]. This paper utilized the C5 algorithm, which is based upon the supervised decision tree based model. This resulted in fantastic performance, achieving over 99% correctly classified data points, using DBMS metrics as their feature vector. This C5 algorithm was not used in this project due to the supervised nature of it.

The area of automatic DBMS optimization contains papers with a range of different solutions. The previously mentioned Ottertune [1] utilizes simple regression of a Gaussian Process model. The performance that was obtained for PostgreSQL was far beyond that of default server configuration, and beat systems optimized by experts by $\sim 12\%$. Following papers use more complex methods of optimization, such as neural networks based on reinforcement learning [30] [55]. These obtain performance that greatly beat Ottertune, at the cost of requiring an offline training phase as well as significant previous data collection.

There has been one previous paper regarding DBMS optimization that utilizes BO [3]. This paper does not optimize any SQL or relational database, and instead is based around RocksDB. Regardless, the paper found that their method resulted in significantly higher performance, with BO allowing the method to converge to the optimum at a much faster pace than other state-of-the-art methods.

While all these previously mentioned papers are also related to automatic tuning of server configurations, they all rely on a simple start-to-finish optimization session. It is just a matter of how this optimization is done, what DBMS is used and how prior information is included that change among them. The thing that makes this project unique is the idea of continuous tuning, where the tuning session never completely ends but instead remains in the background, ready to start optimizing again if a workload shift is detected.

3 Methodology

3.1 Setup

3.1.1 Hardware

Testing was to be done on the AWS cloud. To have realistic testing, the AWS m5d group was chosen, since these have dedicated storage on the local machine, instead of relying on the virtual EBS. The dedicated storage is needed for repeatability, i.e comparability. To see the exact specifications used during testing, please see section 4.1.

3.1.2 DBtune Implementation

As DBtune is quite limited in terms of workload switching, modifications had to be made to support this. This included changes within DBtune as well as Hypermapper.

As for DBtune, the system had to be changed to allow for loading multiple databases into storage, and the capability to switch which workload configuration to execute next.

For the workload switching, it was designed so the user can set which workloads will be included, and the system will then divide the total count of permitted iterations evenly between the workloads. For example, if the user sets 60 iterations with 3 workloads, then each workload would be executed for 20 iterations. When the allocated amount of iterations is reached for a workload, DBtune will then change what workload configuration OLTP-Bench will run. By doing this, DBtune simulates a live database system where data is collected over a 10 minute period for each iteration, followed by a detection cycle that determines whether or not this data should be classified as a workload shift. The 10 minute measurement cycle coincides with findings from Cuadra-Sánchez and Aracil [6, chapter 2] where they found that aggregating data into 10-15 minutes intervals produced the best detection results when doing anomaly detection on network data.

As for the Hypermapper modifications, significant reworks had to be done to the code to allow for a workload detector that executes in between iterations, and the capability for the workload detector to reset optimization when a shift is detected. Aside from this, functionality remained the same.

Out of the previously discussed optimization strategies within Hypermapper, Bayesian Optimization (BO) was chosen by the authors to solve the database optimization problem, due to the efficient nature of this optimization-scheme. More specifically, the model to be used is a single-tasked BO with a prior for the optimum [7] using random forest as a surrogate model and EI as the acquisition function.

The server optimization is done by HyperMapper. HyperMapper relies on a predefined search-space, meaning that the initial server configuration as well as the values to be investigated needs to be known. This also means that what server configuration knobs that are to be tuned need to be known. In table 1 a

list shows the name, how many different values that existed for the optimizer to select, the smallest and largest values for each knob that was used.

Knob name	# of values	Smallest value	Largest value
shared_buffers	20	640	8192
work_mem	25	4096	51200
random_page_cost	18	0.1	8
effective_io_concurrency	5	1	400
max_wal_size	7	4	64
max_parallel_workers_per_gather	5	1	16
max_parallel_workers	2	4	8
max_worker_processes	2	4	10
checkpoint_timeout	2	5	10
checkpoint_completion_target	1	0.9	

Table 1: *Table showing a summary of the search space*

Previous literature, such as the paper by Osama Eldawebi [16], found that these are some of the most important knobs for optimization performance in PostgreSQL. The authors of this project has also had the fortune of having these recommended by experts working at DBtune [13].

For each of these knobs, a search-space needed to be defined. Therefore, a number of points were selected for each of these 10 knobs. A complete list of the search space can be found in the appendix under section 6.1.

There were some knobs that had to be predetermined to allow for proper optimization. What these are, and why they need to be set is expanded upon in the last paragraph of section 3.1.3.

A visualization of one repetition can be seen in figure 1

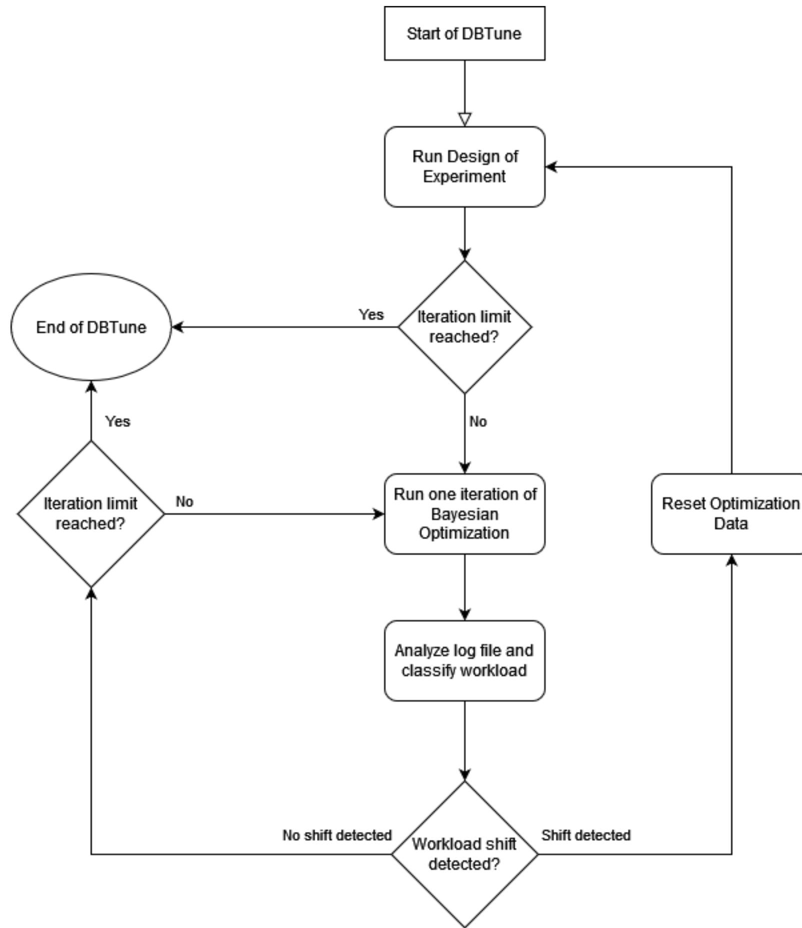

Figure 1: *Flowchart depicting the DBtune process for one repetition*

A final note, even though Hypermapper is capable of optimizing for a wide range of parameters, all the knobs used in this project use discrete integer values. This was not intended, and is by complete chance that the knobs deemed to be the most important all use this type of value.

3.1.3 Workloads

OLTP-Bench has a wide range of workloads. The three used for this project are TPC-C, Wikipedia and CH-benCHmark. Information about other workloads can be found at the GitHub page for the OLTP-Bench project [4].

As previously discussed in section 3.1.2 the running time of one iteration is 10 minutes. The running time is configured through OLTP-Bench's warm-up and execution time, and is set to a 150 second warmup period, followed by an execution period of 450 seconds.

- **TPC-C:** A multi-faceted workload which simulates a large retailer. TPC-C uses more complex transactions than what is regularly seen in workloads for DBMS, providing a difficult workload [43]. Since its release in 1992, TPC-C has gone through multiple versions and become widespread in DBMS benchmarking. TPC-C is a write-heavy workload, this means that the queries TPC-C generate writes a large amount of data to storage. To simulate contention the TPC-C workload uses three components: the warehouse, the district and the customer. Many customers live in one district, and many districts are served by one warehouse. By using this structure many customers simultaneously try to access one point, creating contention [51].

  The weights chosen for the TPC-C workload is following the minimum weight percentage mix from the Transaction Processing Performance Council (TPC), which is the non-profit corporation responsible for establishing TPC-C. The weights can be found on page 70 in their latest official requirements document for the TPC-C workload [51, page. 70] and are also listed below. Compared to the original TPC documentation, the weights of New Order has been set to 45 %, whereas TPC does not established a recommended percentage:

  - New Order: A customers submits a new order to a warehouse, 45%.
  - Payment: Updates customer balance and sale statistics, 43%.
  - OrderStatus: Status of customers latest order, 4%.
  - Delivery: Processes 10 non delivered orders, 4%.
  - StockLevel: Items in stock below threshold, 4%.

  The scale factor represents the number of warehouses in the TPC-C workload. For this project, it is set to 750, which generates a 77 GB large database. The amount of terminals is set to 450.

- **Wikipedia:** This workload is based on real transactions from Wikipedia. The authors of the OLTP-Bench paper state that this workload is "invaluable to test novel indexing, caching and partitioning strategies" [15]. Unlike TPC-C, this workload has very few writes overall and instead is very read-heavy, roughly 92% of transactions being read-only.

  The weights of Wikipedia are chosen directly from the observed percentage mix of transactions from the trace data from the Wikipedia database as reported by OLTP-Bench. The transaction types are not documented by OLTP-Bench, but the names of the transactions seem to be quite self-explanatory. The names and weights are the following:

  - AddWatchList: 0.07 %
  - RemoveWatchList: 0.07 %
  - UpdatePage: 7.6725 %

- GetPageAuthenticated: 91.2656 %
- GetPageAnonymous: 0.9219 %

For Wikipedia, the scale factor represents the amount of Wikipedia pages multiplied by 1000. The scale factor was set to 380 resulting in a 82 GB large database. The amount of terminals was set to 280.

Unfortunately, there are some problems with the Wikipedia workload, either due to poor implementation in OLTP-Bench or it being too stressful for the system. The workload maxes out Java's default maximum heap size of 8GB, crashing the system. The solution to this problem was to increase the maximum and initial heap sizes to 16GB and 2GB respectively, which is done by modifying Java's runtime configuration files. Another issue is that the workload's transactions will sometimes deadlock, to which no solution has been found. However, since deadlocks results in reduced throughput, the optimizer will avoid such configurations.

- **CH-benCHmark:** A complex, hybrid workload that bridges the gap between OLTP-based workloads and OLAP based workloads [9]. CH-benCHmark uses a combination of TPC-C and an OLAP based workload called TPC-H [52]. This provides a workload that is a mixture of fast OLTP based transactions and complex analytical OLAP based transactions. This combination results in mixture of read and writes.

  The weighting scheme of CH-benCHmark is divided into its two components: TPC-C and TPC-H. For the TPC-C part, the same weighting scheme is used as in the TPC-C workload. As for the TPC-H part, it is slightly more complicated. From the latest TPC-H requirements document [52], TPC-H uses "streams of query[transaction] sequences" instead of weights. Streams do not seem to be implemented in OLTP-Bench, and therefore can not be used. Instead the used weighting scheme is close to the original sample configuration from OLTP-Bench, but not exactly the same. The reason is that the transaction 15 is broken in OLTP-Bench [32], there are in total 22 transactions. Therefore, the four percentage chance of transaction 15 have been redistributed. To see the exact weight distribution see table 29 in appendix. For more information on TPC-H and to investigate what each query does, see the TPC-H requirements document [52, page. 29].

  The scale factor was set to 600, generating a 61 GB large database. As in TPC-C, the scale factor represents the number of warehouses. For the terminals CH-benCHmark is defined in a different way. The terminals are divided into two sets, one for TPC-C and one set for TPC-H. TPC-C is set to 80 terminals, whilst TPC-H is set to 200 terminals.

  Just like Wikipedia, CH-benCHmark has some issues. When running the CH-benCHmark workload, it can result in a crash on rare occasions. The solution to the problem was a simple try-except-clause installed into DBtune. The try-except-clause will catch when such a crash happens, and

just re-run the BBF function without notifying Hypermapper. When a run finally succeeds, the throughput value is reported back as usual.

The scale factor and amount of terminals were set to tax the hardware. Each configuration file was carefully crafted to assure that the system would be throttled in some aspect. For example, the TPC-C workload's throttle will usually be IO based, since this is a very write-heavy workload. For the Wikipedia workload, due to the read-heavy nature, it will more likely be performance-bound by the CPU instead. Similarly, due to consisting of many writes, as well as process-demanding queries, CH-benCHmark can be bound by either IO or CPU. Another important aspect to consider is that the database should not be able to be stored in memory alone, since this could give very misleading performance. To achieve this, the scale factor was set so the database was at least twice the size of the memory, forcing the database to be stored mostly on cold storage, much like a real server situation.

As previously discussed in section 3.1.2 the project is using some predetermined knobs. One of these knobs is "max_connections" which determines the maximum amount of concurrent connections to the PostgreSQL database. Such a knob must be predetermined, since the workload will not be able to execute properly if this value is set too low, and performance will be hampered if set too high. For example, when running Wikipedia, too small a value will lead to either deadlocks, or complaints of a lack of connection slots. To avoid these issues, the connections were set to 800, 850 and 800 for TPC-C, Wikipedia and CH-benCHmark respectively. Another knob is the "dynamic_shared_memory_type" which is set to "mmap" during execution of CH-benCHmark. "mmap" tells PostgreSQL to simulate shared memory by using memory-mapped files within the database, which must be used by CH-benCHmark to avoid database errors.

3.2 Data Collection

3.2.1 Query Collection

As the databases are created at the start of a DBtune repetition, the pg_stat_statements (pss) extension is installed. Then after the databases have been filled with data, the pss_reset() command is executed to clear irrelevant data that was collected during the loading phase.

As the system starts running each optimization iteration of OLTP-Bench, pss will collect data during the workload execution. When the iteration is finished, the query types are collected. Each of these consists of text that represents the full query sent to the database. From the query type, the leading query statement is collected, such as SELECT, UPDATE, INSERT, and is combined with the total number of times the statement was executed during the iteration. This data is collected via a simple query to the database, and saved for future use. After this step, and before the next iteration is started, pss_reset() is executed yet again to ensure that data is reset between iterations.

3.2.2 Hardware Collection

Hardware-utilization data was collected in two parts: (1) A set of workload runs, where the database would not be optimized. (2) A set of workload runs, where optimization would be running while data was collected. This was done to compare the hardware-utilization metrics for non-optimized and optimized data. This is an important section since the collection process is the foundation of the analysis of hardware metrics and workload detection in the later results sections. This section will start with a general explanation of the collection process and then end up with an explanation of how (1) and (2) were collected.

The data was divided into the several categories depending on which system a metric mainly affected:

- CPU: Contains metrics like CPU utilization in per cent, the number of active programs, ...
- Virtual Memory: Contains metrics related to random accesses memory (RAM). Information like the amount of RAM available immediately, RAM used, ...
- Swap Memory: Contains metrics like used swap memory, percentage of available memory, ... Gnome a famous Linux open-source interface describes swap memory in the following way. “Swap memory or swap space is the on-disk component of the virtual memory system. It is pre-configured as a swap partition or a swap file when Linux is first installed, but can also be added later.” [42].
- Disk: Contains metrics like the number of reads, time spent reading on disk, ...
- Net: Contains metrics like the number of bytes sent, bytes received, ...

Each category contain many more metrics, see appendix section 6.3, table 30.

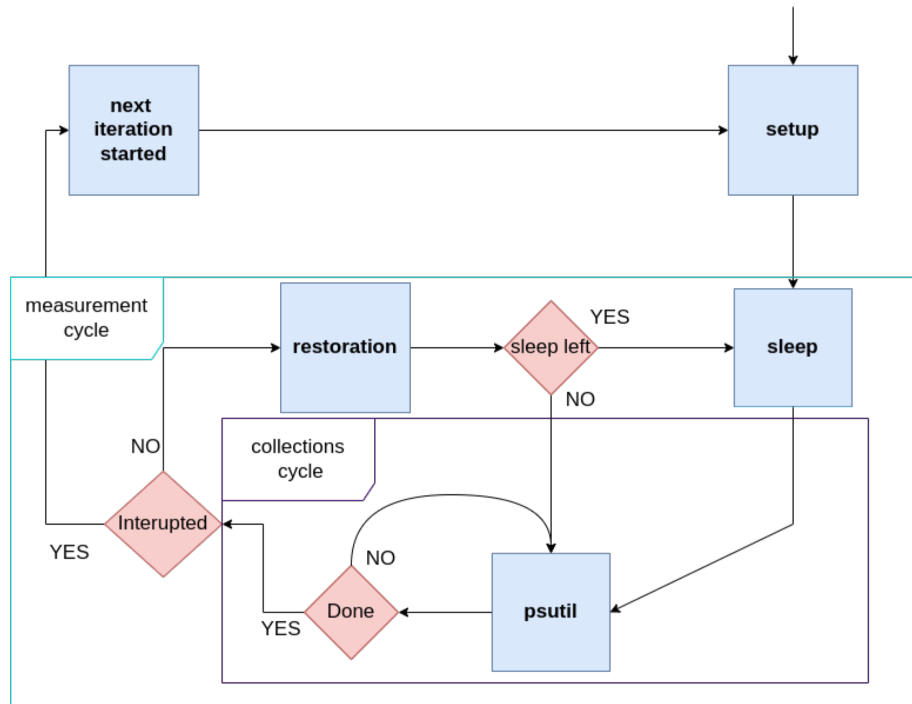

Figure 2: *Shows basic properties of the data collection on hardware-utilization.*

In figure 2 one can see the basic properties of the hardware statistics collection process. The figure does not show every single step of the process, but it shows how the parts hang together. The recommendation to the reader is to visit the image before and after reading the section.

The collections program was developed as an extension of DBtune. To then run both DBtune as normal and the measurement program at the same time, the measurement program was given its own thread, by utilizing python's threading module [50].

To acquire the relevant hardware data, the previous mentioned tool psutil was used. This is not straightforward though, as most data is collected by querying the OS registers. This is a problem due to how most of the metrics are generated by the OS. Most metrics are generated via accumulation, such as the number of reads from the disk, which returns the number of reads since boot to when the register was queried. The solution to this is to simply restore many of the hardware utilization metrics. How this is done will be elaborated on later in the section.

Other tasks such as having a consistent calling rate to psutil were also difficult since each collection cycle is slightly different in time. The collection cycle is the act of gathering the data points related to the hardware metrics, by utilizing psutil, i.e this is when psutil's functions are called to return the metric values. The collection cycle should always be started after some predefined time has

passed, this predefined time is always the same. For example, if the predefined time is set to 5 seconds, then the collection cycle should start at 5, 10, 15, ... seconds. The total period, including the waiting time for the collection to start and the collection time itself, is called a measurement cycle. The measurement cycle assures that the collection cycle is started at the correct time point. To maintain the measurement cycle, the collections thread would sleep, so that the sleep would always end when the predefined time had passed. This was achieved by simply subtracting the predefined period, with the time that already had passed since the last sleep, see equation 4. The code can be viewed in the appendix at page 113. The used predefined time can be found in the results section at 4.1.4.

$$sleep_time = max(predefined_time - (current_time - last_time), 0) \quad (4)$$

For the plotting data, some sort of alignment must be used to compare the workloads in time. To achieve this, a counter was started in the beginning of every iteration, and a timestamp based on this counter was appended to every measurement cycle. Every cycle would then have a time stamp that corresponds to the amount of time the current iteration had been executing.

Further, some of the metrics are process-based and some are system-wide. For example, the CPU utilization in percent is a system-wide metric. Another metric, "aws", which represents the actual used memory by the process, is instead a process-isolated metric. In PostgreSQL each connection is given its own process identification (ID) and ran as its own process, these ID remain over the lifetime of the connection. Moreover, there are a few more PostgreSQL processes that are non-connections but still active processes, for example, the postmaster. These and the connection processes are found by using psutil's function "process_iter", which iterates through all active processes on the host system. Then by filtering for PostgreSQL , the wanted processes are found.

As mentioned previously, some metrics will need to be periodically restored, with how often depending on the metric itself. Restorations can be divided into several categories, with the main ones being restoration every iteration and restoration between each measurement cycle. Restoration can be as simple as saving value x before the measurement cycle, where x is some hardware utilization metric. Then when it is time to acquire the data point, value y will be saved and the resulting data point printed to storage would be $z = x - y$. In summary, this measures how a metric was affected within one measurement cycle. For example, how many reads happened in five seconds. This is the per measurement cycle restoration, called "cycle". All the metrics related to statistics about disk fall into this category. This basic restoration principle is also the same for the PostgreSQL isolated metrics, the key difference is that each process must be restored individually. Even though connection processes remain the same over the lifetime of the connection, this is not the case for all PostgreSQL processes, which will be referred to as "dying processes". Further complicating things, once the dying processes die, the process ID can be reused

by any newly spawned process in Linux. Therefore one can not simply sum all processes and use the last sum of the metrics as the x and the new sum as y. Instead, each process' ID with its last collected metrics is stored and then used as the restoration. If a process ID is previously unseen then the metric is simply discarded, something which never occurred in testing. For the other main category, iteration-based restoration, there is only one set of metrics belonging to this type of restoration, the measurement category swap memory. In this case, the computer's swap memory is flushed in between the iterations. This is done to try to reset each measurement to the same starting state. In the same vein, the caches are also restored between the iterations by utilizing the Linux command "`sudo sh -c \"echo 3 > /proc/sys/vm/drop_caches\"`" [47]. A small list describes the restoration operations further:

- **never** : Restoration is never done.
- **iteration** : Restoration is done per iteration.
- **cycle** : Restoration is done for every measurement cycle.
- **flushed** : Restoration is done every iteration by flushing swap memory.

One problem with using process-based metrics is the instability of the total amount of running processes at a given time point. For example, at one time point, there may be 58 PostgreSQL processes and at another time point 57. This is a problem in Ubuntu, since parents seem to accumulate their children's metrics once they die. This means that if the system measures process A and process B and B is a child of A, then when B dies A will accumulate for example all B's disk writes into A's disk writes. This becomes a problem when using restoration, due to how it will suddenly seem like A wrote a huge amount of data in between the five-second collection period. For this reason, all parent processes are banned. From testing, this amounts to two PostgreSQL processes. Exactly which these two processes are was not investigated, but most likely one of them is the back-end process generator which creates all connection processes. The other is unknown. Doing this parent pruning removed the instability and the process count became stable throughout the collections process.

Previously it has been described that many metrics are treated differently in terms of restoration, some are system-dependent and some are system-wide or PostgreSQL isolated, there are two main tables sharing this information. The most comprehensive table is located in the appendix section 6.3, and table 30, a small selection is described here in table 2. The appendix table also shows the corresponding function used to collect the data, otherwise, the tables share the same information. The information in the tables are which psutil function call was used or a description of the metric collected, how often the metric is restored, and if the values of the function are system-dependent (SD) which is represented as true (T) and false (F) and lastly if collected just for PostgreSQL represented by a (P) in the metric name.

Table 2: *The table shows some of the metrics collected for hardware utilization statistics under the collections module psutil [45], the full table can be found in appendix at table 30. In more detail the table shows; a small description of the metric, when it is restored and if system dependent (SD). If the metric is a PostgreSQL isolated metric then a (P) is shown in the function name. Headers with only a name is the collection category. The units are milliseconds (ms) if time is involved and bytes (b) if storage or memory.*

function	collects	restored	SD (T/F)
- CPU -			
idle	time idling (ms)	cycle	F
- Virtual Memory -			
slab	kernel cache (b)	never	T
vms (P)	virtual memory, total used (b)	never	F
data (P)	non text memory (b)	never	T

Another problem of the measurement program was performance issues. For example, using the collection function "memory_info_full" instead of "memory_info" for PostgreSQL dependent metrics would increase the collection time a 100-fold, likely overshooting the measurement cycle by a large margin. Unfortunately the extra metrics included in "memory_info_full", psutil claims to be the most important to accurately measure process memory usage. Another non-program-related performance issue was the flushing of the swap memory. If the flushing operation occurs while the workload is running, it can take so long that the measuring thread does nothing else but wait for this operation to finish. Thus a special semaphore blocks DBtune in the setup phase of an iteration until finished, the setup phase being the flushing and other setup operations as restoration. Other performance aspects were also taken into account, for example, the measurement thread would be given the highest CPU and IO priority [44], [24]. Code was also built with performance in mind. For example, the measurement thread runs three "setup cycles", and these setup cycles correspond to five seconds each, i.e 15 seconds. Then after the last cycle, all PostgreSQL processes will be collected and stored in a list for later use, exploiting the fact that a PostgreSQL connection process remains the same over the lifetime of the connection, the removal of parent processes is also done during this setup phase.

The collection of the non-optimized hardware data(1) was done simply by running each workload in a cycle. The measurement program measuring on the non-optimized database would run OLTP-Bench for a whole iteration then the workload would be shifted to the next workload. For the last workload, the cycle would start over and the first workload would run again. During a workload's running time the measurement thread would collect its data and store it locally. To avoid measuring the measuring program's writes, data was only written once the whole iteration had finished and the measuring temporarily stopped. For each repetition, the AWS instance would be rebooted.

Collection of data in optimized circumstances(2) worked very similarly, the only difference being that the cycle was removed and optimization turned on.

DBtune would be allowed to optimize the database whilst the measurement program was running, which meant that the database configuration file would change on a per iteration basis, updating the database configuration, and possibly altering the hardware statistics. Another change is that instead of switching workloads for every iteration one workload would run the whole course of the repetition. This change was made to allow the optimization program to properly optimize for the given workload. After one repetition was done the AWS instance would be rebooted, and then the next workload would run or the same repeated.

3.3 Data Processing

3.3.1 Query Statistics Processing

After the query data has been saved from the collection phase, the data is read into memory. The data is normalized so that each type of query is now represented by the percentage of that query statement in its iteration, rather than being the total count of that query statement in the iteration. As to not lose information, the total query count for the iteration was appended to the iteration data.

For each workload type, the average number of queries and average percentile of each query is aggregated, together with their respective standard deviation.

When testing the workload detector, each iteration data will be forwarded to the workload detector, where the query count will undergo further processing as will be described in section 3.5.3.

3.3.2 Hardware Statistics Processing

The hardware utilization data was aggregated in many different ways to produce plots and statistics. To achieve this a processing tool was created, the tool can be found at GitHub [53]. In the rest of this section, a two workload example will be used throughout, since there are always only two workloads in one comparison. The comparisons are expanded exhaustively, i.e TPC-C versus Wikipedia, TPC-C versus CH-benCHmark and TPC-C versus Wikipedia to cover every comparison scenario.

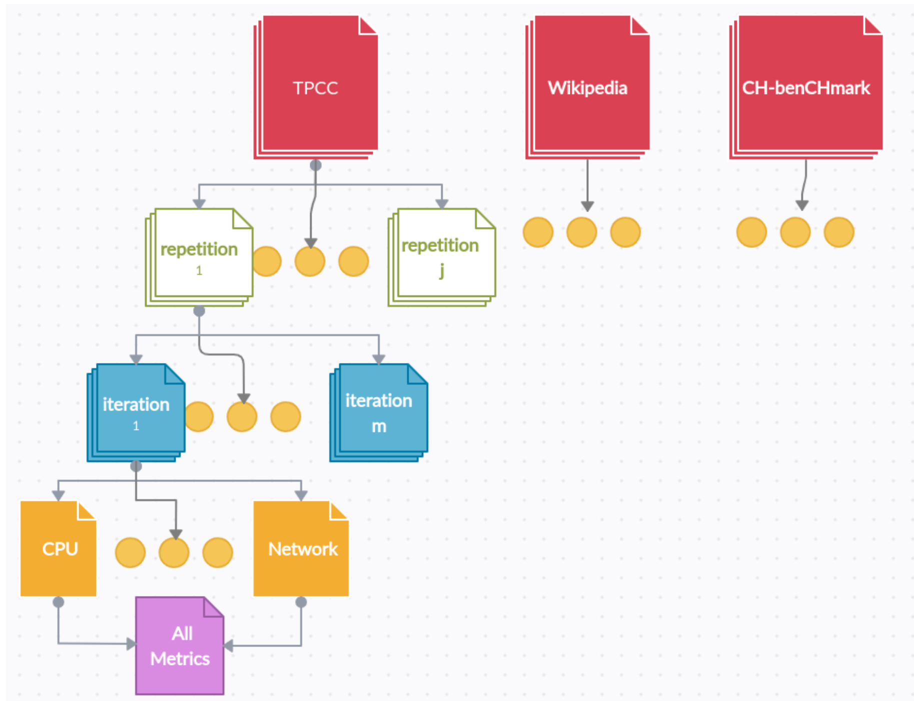

Figure 3: *Showcasing the file structure of the data collection processes for each workload.*

The file structure will work as a mental overview of how the data was selected for aggregation. The file structuring had the following structure: (1) At the top, the workload folder. (2) Within each workload folder the per repetition sub-folders. (3) Within the per repetition sub-folder, per iteration sub-folders. (4) Lastly within the per iteration sub-folders, the files storing the hardware utilization data. See figure 3. Each hardware utilization file represents one hardware utilization category, which is CPU, Virtual Memory, ... see the specification in section 3.2.2. These hardware utilization files are then concatenated into one file.

The concatenated hardware utilization file was represented as a matrix, the "main matrix", containing n columns and t rows. The columns representing the metrics and the rows representing the data points collected within the same measurement cycle. If one takes the Virtual Memory category as an example, there are 12 metrics: buffers, cached, ..., see table 30. In this example, one would have 12 columns in the main matrix from the category Virtual Memory, one for each metric. If one would select the same row over all these 12 columns, one would have 12 different metrics that were collected within the same measurement cycle. If t represents the number of rows, then t is equal to the number of measurement points. If n represents the number of columns, then n is equal to the sum of the number of metrics within all the hardware utilization categories.

First, the general aggregation process will be described. Afterwards what the data was used for, and how the data is useful.

The data between non-optimized and optimized data is aggregated a little bit different. Instead of one-to-one comparisons between the iterations, only every fifth iteration is taken out from the optimized hardware utilization data, accounting for the different iteration lengths. See the result section for the specification 4.1.4.

The warm-up and termination data were discarded. Warm-up data is data collected during the "warm-up phase". Termination data is just an end-of-experiment phase where the system is shut down.

The data was aggregated in the following ways by the processing tool:

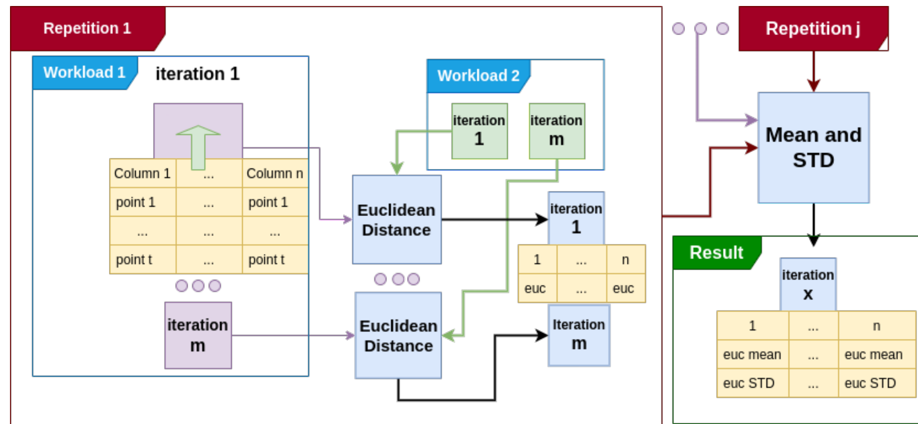

Figure 4: *Shows how the Euclidean Workload data was generated.*

- **Euclidean Workload**: The mean and STD of the Euclidean distance (ED) between workload 1 and workload 2. The ED would be calculated between the corresponding iteration metrics under some repetition. For example, workload 1 iteration x column z would calculate its ED against workload 2 iteration x column z, using all rows, see equation 5. Then the same procedure but for the next column $z + 1$. The process would then continue for iteration y against y and so on. Now a value of the ED between the workloads for every column in every iteration exists. The same calculation would then be done for every repetition. One would now have several values of the ED in every column in every iteration, one value for each repetition. For example, before calculating the ED the two compared iteration files would look like [[x1....xt], ..., [n1...nt]], after ED only one file looking like [x, ..., n], after adding the repetitions only one file like this [[x1...xj], ..., [n1...nj]]. From this last file, i.e the repetition data, mean and STD would be derived for each column, i.e [[x1...xj], ..., [n1...nj]] $\rightarrow$ [[x_mean, x_std], ..., [n_mean, n_std]]. So in the end, one would have the same amount of iterations as one began with, but the data reduced to one repetition and one of the two workloads, see figure 4.

$$
\begin{aligned}
W_{X1z} = & \text{ The column vector z of all data points from iteration X} \\
& \text{in workload W1} \\
W_{X2z} = & \text{ The column vector z of all data points from iteration X} \\
& \text{in workload W2} \\
C = & W_{X1z} - W_{X2z} \\
\text{Euclidean Distance} = & \left(\sum_{a \in C} a^2 \right)^{1/2}
\end{aligned} \tag{5}
$$

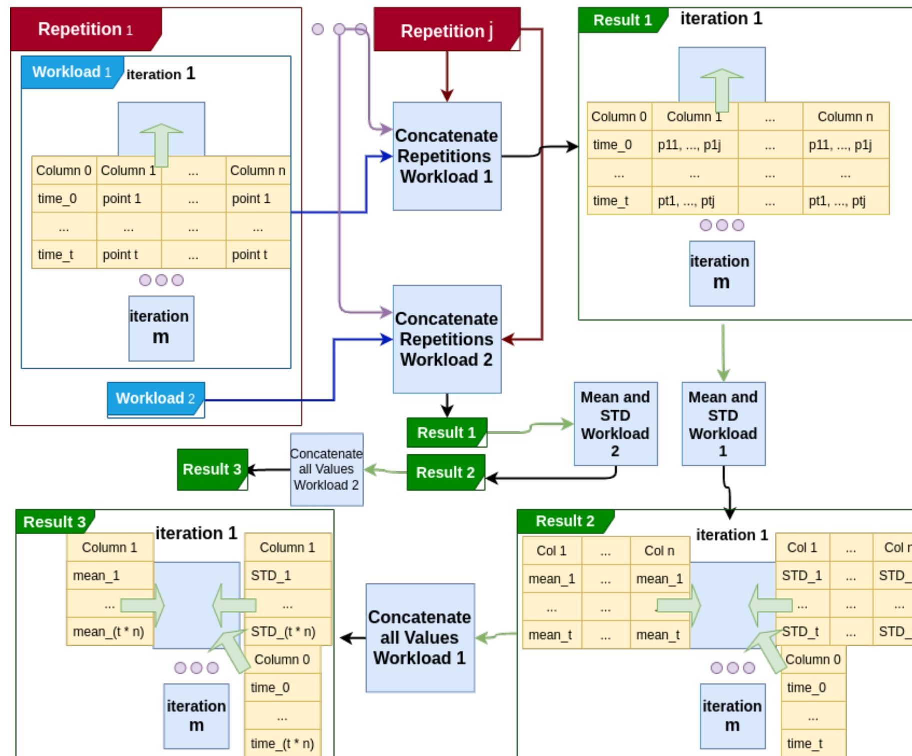

Figure 5: *Shows how the timeline data was generated.*

- **Timeline**: Mean and STD for each data point on a timeline. By combining the repetitions of each workload, the data points would be vectorized to contain the number of values, as there are repetitions of the workload. For example, for three repetitions of the same workload, a column in the respective repetition would from the beginning look like [x1, ..., xt], [y1, ..., yt] and [z1, ..., zt], after the vectorization they would share the same

vector, looking like [[x1, y1, z1], ..., [xt, yt, zt]]. From these vectorized data points the mean and STD could be calculated. After the means and STD were generated, all iterations would be concatenated into one big line, the time values accumulated separately for each workload, see figure 5. In summary, timeline houses two workloads, the mean and STD data for every measurement cycle.

For the statistics processing divisions by zero could occur. All such divisions by zero are handled by just keeping the nominator.

Another issue was length differences. When comparing different workloads they could differ in the amount of collected data points, the solution was to drop the tail of the longer one so they became equal.

- **Euclidean Workload**: Several metrics are generated from the Euclidean Workload data: "outer STD", "growth" and "growth variance".
  - **"outer STD"**: helps in identifying the metric's stability in the iteration-based viewpoint, an unstable metric increases the risk of false classifications. "outer STD" is the normalized mean-STD taken over all iterations for a metric. Each iteration in the Euclidean Workload data contains the STD and the mean for every metric column. By selecting STD and the same column in all iterations one has a list of standard deviations, then taking the mean of these standard deviations one gets the mean-STD. Similarly by taking the mean of all means one gets the mean-mean. To normalize the mean-STD it is divided by the mean-mean, the normalized value is called "outer STD". Equation 6 shows the procedure for one metric, where m is the amount of iterations. The whole procedure is then to be repeated for every metric, i.e every column in the iterations. The "outer STD" shows in percentage how much the average STD makes up the average distance between the two workloads for a metric. "outer STD" is categorized as: $very_high >= 0.5, high < 0.5, medium < 0.3, low < 0.1$.

$$\begin{aligned}
&ewd = \text{Euclidean Workload data} \in \mathbb{R}^{m\times 2\times n}\\
&metric_{idx} \in \{1, 2, ..., n\}\\
&\text{mean-mean} = \frac{\left(\sum_{iteration_x \in ewd} iteration_x(1, metric_{idx})\right)}{m}\\
&\text{mean-STD} = \frac{\left(\sum_{iteration_x \in ewd} iteration_x(2, metric_{idx})\right)}{m}\\
&\text{"outer STD"} = \frac{\text{mean-STD}}{\text{mean-mean}}
\end{aligned} \tag{6}$$

  - **"growth"**: displays if the optimized data is diverging from the non-optimized data. The information is important since a diverging metric increases the risk of self-detection. When comparing data between a

non-optimized and an optimized workload scenario, the "growth" can be calculated by the Euclidean Workload data. "growth" is calculated by linear regression. The linear regression line is fitted between all points from iteration 1 to last, by utilizing NumPy's function lstsq [36]. Then the growth coefficient is the leaning of this line, the growth coefficient is then multiplied by the number of iterations, but since the metrics are of varying magnitude normalization is needed. "growth" is then normalized by dividing the multiplied growth coefficient by the first Euclidean iteration value, $growth = \frac{growth_{cof} \times m}{mean_1}$. This is the process for one metric, the procedure for the rest is the same. Equation 7 shows the process for one metric, the ":" indicate that every value is selected in that dimension. "growth" is categorized by negative and positive growth as: $+very_high >= 2, +high >= 1, +medium >= 0.5, +low >= 0, -low >= -0.5, -medium >= -1, -high >= -2, -very_high < -2$.

$$
\begin{aligned}
&ewd = \text{Euclidean Workload data} \in \mathbb{R}^{m \times 2 \times n} \\
&metric_{idx} \in \{1, 2, ..., n\} \\
&ones = \begin{pmatrix} 0 & 1 \\ 1 & 1 \\ \vdots & \vdots \\ m & 1 \end{pmatrix} \\
&mean_{values} = ewd(:, 1, metric_{idx}) \\
&\text{growth coefficient}, _ = lstsq(ones, mean_{values})(1) \\
&\text{"growth"} = \frac{\text{growth coefficient} \times m}{mean_{values}(1)}
\end{aligned}
\tag{7}
$$

– **"growth variance"**: shows how trustworthy the "growth" value is. If high "growth variance" then it seems that there is no stable trend in any direction and the growth may be represented by just high variation or outliers. To calculate the "growth variance" the residual of the linear regression is used. The residual was then divided by the number of iterations, calculating the average distance from the regression line. The average distance was then divided by the unnormalized growth, generating a divergence percentage on average from the line. Equation 8 shows the process for one metric . "growth variance" is categorized as: $very_high >= 0.5, high < 0.5, medium < 0.3, low < 0.1$.

$$
\begin{aligned}
&ewd = \text{Euclidean Workload data} \in \mathbb{R}^{m \times 2 \times n} \\
&metric_{idx} \in \{1, 2, ..., n\} \\
&ones = \begin{pmatrix} 0 & 1 \\ 1 & 1 \\ \vdots & \vdots \\ m & 1 \end{pmatrix} \\
&mean_{values} = ewd(:, 1, metric_{idx}) \\
&(\text{growth coefficient}, _), residual_{squared} = lstsq(ones, mean_{values})(1 \wedge 2) \\
&residual = \sqrt{residual_{squared}} \\
&\text{average dist from line} = \frac{residual}{m} \\
&\text{"growth variance"} = \frac{\text{average dist from line}}{\text{growth coefficient} \times m}
\end{aligned}
\tag{8}
$$

- **Timeline**: From the timeline data, several more metrics are derived: "STD", "distance", "inner STD" and "intermingling".
  - **"STD"**: tells how much a metric varies. Though workload selection was done to generalize the results, there is no evidence that the selected workloads cover every case or even the general. Therefore it is interesting to know how stable a metric is, on its own, since when the "STD" is low the likelihood increases that the metric will generalize well. "STD" is calculated by simply taking the mean over the STD values of the time-points for each workload and metric. Equation 9 displays this process for one metric and one workload. "STD" is categorized by: $low < 0.1, medium < 0.3, high < 0.5, very_high >= 0.5$.

$$
\begin{aligned}
&v = m \times t \\
&td_{STD} = \text{Timeline data STD} \in \mathbb{R}^{2 \times v \times n} \\
&workload_{idx} \in \{1, 2\} \\
&metric_{idx} \in \{1, 2, ..., n\} \\
&STD = td_{STD}(workload_{idx}, :, metric_{idx}) \\
&\text{"STD"} = \frac{\sum_{p=1}^{v} STD(p)}{v}
\end{aligned}
\tag{9}
$$

  - **"distance"**: shows how separate two workloads are for a given metric. A high value means that there is a large separation between the workloads under the metric, making the metric more likely to

be able to be used in detection methods to separate the workloads accurately. To calculate "distance", the time-point mean values were extracted for both workloads, and averaged. The highest mean value of these two workloads was then divided by the smallest mean value, creating "distance". "distance" is a value always bigger or equal to one. Equation 10 shows the process for one metric. "distance" is categorized as: $low < 1.5, medium < 1.8, high >= 1.8$.

$$\begin{aligned}
&v = m \times t \\
&td_{mean} = \text{Timeline data mean} \in \mathbb{R}^{2 \times v \times n} \\
&metric_{idx} \in \{1, 2, ..., n\} \\
&td_{means1} = td_{mean}(1, :, metric_{idx}) \\
&td_{means2} = td_{mean}(2, :, metric_{idx}) \\
&mean_1 = \frac{\sum_{p=1}^{v} td_{means1}(p)}{v} \\
&mean_2 = \frac{\sum_{p=1}^{v} td_{means2}(p)}{v} \\
&\text{"distance"} = \begin{cases} \frac{mean_1}{mean_2} & \text{if } mean_1 > mean_2 \\ \frac{mean_2}{mean_1} & otherwise \end{cases}
\end{aligned} \tag{10}$$

- **"inner STD"**: just like "outer STD" shows how trustworthy the "distance" value is but from the point-based perspective. A high "inner STD" indicates an increased chance of self-detection for point-based detection methods. "inner STD" is calculated by taking the distance between workload 1 and 2 for each time-point and deriving the STD. The STD is then divided by the mean distance between the same points. "inner STD" now shows how the distance varies as a percentage of the mean distance. Equation 11 shows the process for one metric. "variance inner" is categorized by $low < 0.1, medium < 0.3, high < 0.5, very_high >= 0.5$.

$$\begin{aligned}
&v = m \times t \\
&td_{mean} = \text{Timeline data mean} \in \mathbb{R}^{2 \times v \times n} \\
&metric_{idx} \in \{1, 2, ..., n\} \\
&td_{means1} = td_{mean}(1, :, metric_{idx}) \\
&td_{means2} = td_{mean}(2, :, metric_{idx}) \\
&diff = td_{means1} - td_{means2} \\
&\text{"}innerSTD\text{"} = \frac{STD(diff)}{mean(diff)}
\end{aligned} \tag{11}$$

- **"intermingling"**: shows how much two workloads overlap for a given metric by calculating intersecting points. High intermingling

increases the chance of false classifications in point-based detection methods. Intersection points are defined in this thesis as; the points that belong to one workload but which are within the other workloads zone of STD. To expand on the definition; each point in the Timeline data comes with an STD value. If one takes a point from workload 1 and a point from workload 2, at the corresponding time-interval, if the STD then overlaps between these two points, it is defined as one intersection point. The number of found intersection points is then divided by the number of total points creating "intermingling". Equation 12 shows the process for one metric. "intermingling" is categorized as: $non = 0, low <= 0.01, medium <= 0.05, high <= 0.1, very_high > 0.1$.

$$
\begin{aligned}
&v = m \times t \\
&td_{mean} = \text{Timeline data mean} \in \mathbb{R}^{2\times v\times n} \\
&td_{STD} = \text{Timeline data STD} \in \mathbb{R}^{2\times v\times n} \\
&metric_{idx} \in \{1, 2, ..., n\} \\
&td_{means1} = td_{mean}(1, :, metric_{idx}) \\
&td_{means2} = td_{mean}(2, :, metric_{idx}) \\
&td_{STD1} = td_{STD}(1, :, metric_{idx}) \\
&td_{STD2} = td_{STD}(2, :, metric_{idx}) \\
&intersections = \sum_{p=1}^{v} \begin{cases} m1 = td_{means1}(p) & m2 = td_{means2}(p) \\ s1 = td_{STD1}(p) & s2 = td_{STD2}(p) \\ 1 & \text{if } m1 < m2 \wedge m1 + s1 > m2 - s2 \\ 1 & \text{if } m1 > m2 \wedge m1 - s1 < m2 + s2 \\ 0 & \text{otherwise} \end{cases} \\
&\text{"}intermingling\text{"} = \frac{intersections}{|td_{mean1}|}
\end{aligned}
\tag{12}
$$

- **"bias"**: tries to quantify if the metrics are affected by being in the first iteration. This is more of a verification value than a performance metric. If the bias is high for the starting iterations then it is hard to compare that data to later iterations. "bias" was generated by collecting the mean from every iteration and then combining all other repetitions of the same workload type and then once again deriving the mean. For example, m iterations, each with t data points would be made into m iterations with 1 data point by the mean value. Then, for example, j repetitions would be combined to create m iterations with j data points each. Then once again mean taken, creating m iterations with 1 data point each. The bias could then be calculated by comparing the mean value of iteration first against every other iteration. In practice, it would work like this. The distance between iteration first and all other iterations would be

calculated on a point-by-point basis, then the mean taken, generating the mean distance from iteration first. Then this mean distance would be divided by iteration first's variance, telling in percentage how much bigger the mean distance is compared to iteration first's variance. The intuition behind this is that if the variance of iteration first is much larger than the mean distance, then it is not any bias at all. "bias" is categorized as: $high >= 1, medium >= 0.5, low < 0.5$.

Several plots were produced: Euclidean Distance plots, timeline plots and 3D plots. The plotting information is useful to realise what the generated statistics are representing in the actual data.

- **Euclidean Workload**: This plot directly uses the data generated in Euclidean Workload. The plot shows the per iteration Euclidean distance between two workloads, per metric. From viewing the plot one can see how the workloads vary against each other on an iteration-basis in the Euclidean distance. This information for example can help to discern which metrics separate the workloads well from an iteration perspective.

- **Timeline**: The data would directly be used in the plot. The plot shows workloads 1 and 2 in one continuous time-plot per metric. In the plot, one can see how a metric is affected over time and iterations. Also one can view, for example: bias, intermingling, variance, separation and so on. This plot also, and the only one to do so, have added the warm-up data.

- **Partitioned**: Is data generated only for plotting. The matrix containing every hardware utilization category was split, separating the hardware utilization categories into 5 matrices. Each containing g_i columns, these were then subdivided down to contain only three columns and then the three columns were plotted together in 3D scatter plots. The subdivision worked in the following way for an array where $g_i = 5$; [1, 2, 3, 4 ,5] $\rightarrow$ [1, 2, 3], [2, 3, 4], [3, 4, 5]. This procedure would be performed on each row, a total of t times. The procedure allows the plotting of some codependencies, even though it misses most. If one looks at the given example of the subdivision, one can see that 1 and 2 are never plotted with 5, so one can imagine by looking at the example that the longer the array becomes the more metrics will not be compared. In these 3D plots light to dark colour means newer to later.

- **Partitioned Mean**: Similar to the Timeline plot but just the mean from each data point. In summary a 3D scatter-plot of workloads 1 and 2 generated by taking the mean of the repetitions. Useful to find codependencies.

- **Partitioned Raw**: Just a 3D scatter-plot of the three first repetitions on the "Partitioned" data, gives insight into the variance of "Partitioned Mean".

3.4 Multi-Workload Testing

To see if the modified version of DBtune works for multiple workloads and to show the benefit of a multi-workload system, a test had to be devised that could run all workloads, and then be compared to some baseline value.

It was decided that all three workload databases would be loaded into storage initially, and then the workloads would be executed sequentially, one after the other. To demonstrate that the system itself works, an oracle was used in place of a workload detector. This oracle will perfectly predict workload shifts. Hardware and query data was still collected, and later used to examine how well detector based upon queries and hardware-utilization would have performed in this multi-workload system.

Each workload was divided into the following three phases:

- Design of Experiment - Gather random data points for BO
- Optimization - BO tries to find the best server configuration
- Post-optimization - After a number of iterations, the best server configuration is maintained until the next workload shift

Each phase will consist of a number of iterations. The exact values can be found later in section 4.1.2.

To compare the performance of this multi-workload optimization, a baseline was obtained by running a naive optimization session that completely ignores any workload shift. In essence, this will be the same as the multi-workload system, except it has a non-functional detector that cannot identify workload shifts. This means that the baseline will optimize for the first workload, switch over to the optimal configuration found after the Optimization phase and then maintain this for the duration of the repetition, i.e the duration of the experiment.

3.5 Workload Detection

Put simply, workload detection is just a matter of data classification. For this type of problem, many possible solutions exist, depending on what problem needs to be solved. What all the methods have in common is that some feature-vector needs to be constructed that can be used to classify the data. When accounting for database performance, there are many different methods that could be taken into account when doing workload detection.

One alternative was to utilize hardware utilization measures, such as CPU, RAM, and disk usage, measured during the execution of OLTP-Bench. However, these measures have some potential issues. The optimization goal of the project is throughput, which is likely linked to the usage of these metrics. Put simply, as the search for the optimal PostgreSQL configuration progresses during the optimization phase, the hardware utilization may change as the throughput changes. Therefore these metrics could change over time during the optimization of the database, even for the same workload.

The other option that was investigated was the usage of the database transaction statements themselves, as described in section 3.2.1.

As the queries themselves won't be affected by the optimization of the database, discovering a way to detect trends in queries would provide a reliable way to separate workloads over time.

As described by previous work [1], queries do not seem to fully characterize the workload. Two systems may have databases structured in very different ways, while still having workloads containing similar query data. This means that that over multiple optimization sessions, these queries fail to provide information regarding optimal configurations. This is not a hindrance to this project however, as optimization is only done for one DBMS instance at a time. As the database and hardware stay the same, it means that the workload could be characterized well enough that different workloads can be separated by only using query-based statistics.

3.5.1 Detection methods

The following section will describe the implementation of the methods tested for workload detection. Most of these methods are unused in the actual detection process and are mainly implemented to compare the performance between them.

The workload detection on hardware metrics is handled by a main method, the main method handles the presentation of the data to the detection methods, see the pseudocode 2. The detection is then done in the main method by utilizing a reference to one of the detection methods, each, of which follows the interface Detector, see interface 1.

The implementation of this main method is easiest described by its functions. The main detection handling function, *do_detection*, receives a feature matrix. The feature matrix contains all the collected data from an iteration, the data as discussed in 3.2.2, but with warm-up and termination data removed as discussed in 3.3.2. The function *do_detection* then divides the feature matrix into feature vectors, each vector containing collected values of a hardware utilization metric. These vectors are then sent to the *detect* function together with needed reference values. The reference values are either the STD and mean of previous feature vectors or just the previous feature vector itself. After then receiving the *detect* function's return value, the algorithm will store if detection was reported.

Furthermore, each detection method has hyperparameters. These are best described in the individual detection methods themself, but they are given through the main method via the function *local_setup*.

Function `detect`(*type: feature_matrix, shape:* $N \times 1$ *;; type: reference_values, shape:* $N \times 1$ *OR* 2×1)**:**
| Return(type: Boolean)

Function `extract_reference_values`(*type: feature_matrix, shape:* $N \times M$)**:**
| Return(type: reference_values, shape: $N \times M$ OR $2 \times M$)

Function `local_setup`(*type: hyperparameters, shape:* 1×1 *OR* 2×1)**:**
| Return(type: null)

interface, Detector. 1: *Displays the interface called Detector, which all the hardware detection methods follow. The interface specification uses some definitions that need to be explained. (1) shape: is the shape of the expected parameter. (2) type: is a short explanation of the expected value. All code is implemented in python, meaning no static types exist, therefore sometimes a function has several types, or shapes. This is defined by adding OR to the description. ";;" delimits the parameters, for example, if a function receives 3 parameters, then one would have; (param1 ;; param2 ;; param3).*

```
Function setup(type: Detector, name: detector ;; type:
 hyperparameters, shape: 1 × 1 OR 2 × 1, name: hyperparameters):
    detector.local_setup(hyperparameters) ;
    first_iteration ← True ;
    reference_values ← None ;

Function do_detection(type: sampled metrics, shape: N × M, name:
 feature_matrix):
    detection_found? ← False ;
    if first_iteration then
        first_iteration ← False ;
        reference_values ←
         detector.extract_reference_values(feature_matrix) ;
        Return(no workload detected)
    else
        for index, feature_vector ← feature_matrix do
            /* A feature_vector is an extracted column from the
               feature_matrix, the index corresponds directly
               to the index of the extracted column.  For
               example, first column 1 is extracted then 2, 3,
               ..., the corresponding indexes for each
               extraction will be 1, 2, 3 ...                 */
            reference_value ← reference_values[index] ;
            detected? ← detector.detect(feature_vector, reference_value) ;
            if detected? == True then
                detection_found? ← True ;
            end
        end
        if detection_found? == True then
            reference_values ←
             detector.extract_reference_values(feature_matrix) ;
        end
        Return(detection_found?)
    end
```

pesudocode 2: *Displays the pseudocode of the main method running the detection methods. The pseudocode uses some definitions that need to be explained. (1) name: is the given name to a parameter in the local function. (2) shape: is the shape of the expected parameter. (3) type: is a short explanation of the expected value. All code is implemented in python, meaning no static types exist, therefore sometimes a function has several types, or shapes. This is defined by adding OR to the description. ";;" delimits the parameters, i.e if one function receives 3 parameters then one would have; (param1 ;; param2 ;; param3).*

One wants to assert that the used complicated methods are necessary. One way to do this is to use a simple and naive method. The naive method should then be used as a baseline and compared against the non-naive methods. Since

workload detection logically can be divided into distances between the workloads, the Euclidean distance was chosen as a naive and simple function.

The implementation of the Euclidean Distance method is described by the pseudocode at 3, which implements the previously discussed interface Detector. The method starts in the local setup, where the distance hyperparameter is given, dictating the maximum allowed distance between two workloads, for them to not be specified as a workload shift. The Euclidean implementation returns the entire feature matrix as its reference values, the columns being the feature vectors. When the main method then requests a detection, the feature vectors are compared against the reference vectors. The comparison is done by unifying both vectors, then calculating the infinite norm to normalize the vectors, and then finally calculating the Euclidean distance. The distance is then compared against the given hyperparameter, if bigger, a workload shift has been detected. Euclidean Distance is an iteration-based detection method.

```
Function local_setup(type: int, name: threshold):

Function extract_reference_values(type: doubles, shape: N × M,
 name: feature_matrix):
 |  Return(feature_matrix)
Function detect(type: doubles, shape: N × 1, name: v1 ;; type:
 doubles, shape: N × 1, name: v2):
    /* The reference value, v2, is the vectors from the
       reference feature matrix                                  */
    v1_n, v2_n ← Normalize(v1, v2) : Returns (v1,v2)/|v1∪v2|_inf ;
    distance ← Calculate_Euclidean(v1_n, v2_n) ;
    if distance > threshold then
     |  Return(workload shift detected)
    else
     |  Return(no shift detected)
    end
```

pesudocode 3: *Pseudocode of the Euclidean implementation*

In the implementation of CUSUM-chart the authors mainly follow the implementation as mentioned in section 2.3.3, the pseudocode can be viewed at 4. As previously described, the standard implementation of CUSUM-chart leaves the interpretation of what is a detected anomaly open. The definition of when the lower bound or upper bound has breached the threshold is left out. In this project a simple factor of the STD was used as the threshold. A breach of this threshold would mean that an anomaly had been detected. When such a breach had happened, lower and upper bound would once again be restored to 0. To increase the stability, an additional hyperparameter was added, called "outlier_threshold". The additional hyperparameter is the needed percentage of anomalies for a hardware metric to be classified as a workload shift. This percentage is derived from the feature vector length as $percent = \frac{nbr_anomalies}{length}$. For the reference values, they were simply the mean and STD from each column in the feature matrix. CUSUM-Chart is a point-based detection method.

```
Function local_setup(type: double, name: distance_threshold ;; type:
 double, name: outlier_threshold):
Function extract_reference_values(type: doubles, shape: N × M,
 name: feature_matrix):
 | Return : get_mean_STD_from_each_column(feature_matrix)
Function detect(type: doubles, shape: N × 1, name: v ;; type: doubles,
 shape: 2 × 1, name: mean_std):
   mean, STD ← mean_std ;
   detections ← 0 ;
   for element ← v do
       lower, upper = accumulate_lower_upper(lower, upper, element,
           mean, STD) /* increments the bounds according to
           the algorithm description.                          */
       if (lower ∨ upper) > distance_threshold then
           detections ← detections + 1 ;
           lower, upper ← 0 ;
       end
   end
   if detections > outlier_threshold then
       Return(workload shift detected)
   else
       Return(no shift detected)
   end
```

pesudocode 4: *Pseudocode of the CUSUM-chart implementation*

The MI implementation mainly uses Scikit-learn's implementation [14], see the pseudocode implementation 5. The implementation logic can largely be described by section 2.3.3 but uses the more advanced binning method described in paper [28]. MI's hyperparameter works in reverse compared to the other methods' hyperparameters. A low value on the hyperparameter means that only workloads with great distance between them will be classified as a workload shift, and a high value the opposite. MI's hyperparameter will henceforth be called "difference". MI is a iteration-based detection method.

```
Function local_setup(type: double, name: difference_threshold):

Function Extract_reference_values(type: doubles, shape: N × M,
 name: feature_matrix):
 | Return(feature_matrix)
Function Detection(type: doubles, shape: N × 1, name: v1 ;; type:
 doubles, shape: N × 1, name: v2):
 | /* The reference value, v2, is the vectors from the
 |    reference feature matrix                                  */
 | ;
 | distance = MI(v1, v2) ;
 | if distance > difference_threshold then
 |  | Return(workload shift detected)
 | else
 |  | Return(no shift detected)
 | end
```

pesudocode 5: *Pseudocode of the mutual information implementation*

See table 3 to see a summary of which types the detection methods belong to.

Table 3: *Shows which detection methods are iteration- or point-based.*

detection method	point-based	iteration-based
Euclidean Distance		X
CUSUM-Chart	X	
MI		X

3.5.2 Performance Measurement

To evaluate the detection methods some performance score is needed. This section describes how the tests were carried out and how the scores were given. The evaluation was done through testing sets. These sets are the collected data as described in section 3.2. The testing sets were then given to the simulation program, which would present the data to the detection methods in the same way as would be expected at runtime.

The hardware utilization simulator would present and load the previously collected data files. The data files, which are the .csv files containing the hardware metrics. The .csv files would be presented on an iteration basis, exactly as in the runtime environment. Though, as can be seen from section 3.5.1 some data needs to be preloaded, to act as the reference values. The detection program would then read these presented .csv files and use one of the detection algorithms. The result of the detection algorithm would then be reported back to the simulation program, which would tally if this was a reported workload shift or not and which hardware metrics were responsible for the detection. The simulation program could then tally the detection percentage for a given hardware metric, enabling feature pruning/feature detection.

The hardware simulator would run negative and positive detector tests

separately. A negative test is when the detector is given a data file containing the same workload as the current one loaded, i.e this data file should not be classified as a workload shift. A positive test on the other hand is the inverse and should be classified as a workload shift. For example for the negative test, if workload TPC-C iteration 1 would be loaded first, then iteration 2 would be presented, then iteration 1 again loaded and then 3 presented and so on. After all the other collected iterations have been presented to iteration 1, i.e 2-last presented, then TPC-C iteration 2 will be loaded, and iteration 3 presented and so on. Then the same would happen for all workloads, and for each repetition the whole apparatus started over again. Equation 13 shows the total amount of negative test iterations this would generate. The positive tests would be generated similarly. For example, TPC-C iteration 1 would be loaded then Wikipedia iteration 1 presented then TPC-C iteration 1 loaded and Wikipedia iteration 2 presented and so on for Wikipedia 1-last and then CH-benCHmark 1-last and then iteration 2 TPC-C loaded and the cycle repeated and for all workloads, and started over for each repetition. Equation 14 shows the total amount positive test this would generate, note that this equation is only valid for $number_workloads = 3$. A detected shift on the negative test is called a "false positive", and a detected shift on the positive test is called a "true positive".

$$number_repetitions \times number_workloads \times \left(\sum_{i=1}^{number_iterations-1} (i) \right) \tag{13}$$

$$number_repetitions \times number_workloads \times number_iterations^2 \tag{14}$$

For the selection of hyperparameters on hardware detection, a search script was used. The search script would select hyperparameters to try from Gaussian distributions. The Gaussian distributions themselves were created from the NumPy function "normal" [37]. This function needs the centre and the STD to be defined for the wanted Gaussian distribution. The Gaussian distributions would then be sampled at the beginning of a test, and given to the search script to score the hyperparameters. The test itself would consist of running all the negative and positive tests. The number of false positives and true positives would then be returned to the script after the tests had finished. These numbers were then multiplied with weights, which are defined in the results section 4.1.4. After the multiplication, the scores would be summed and the used hyperparameters stored together with their score. Then several of these "iterations" would be run, each time with a new set of hyperparameters, and at the end, the best score and hyperparameters were reported.

Once again the metrics would be classified based on their performance. Each metric would get a rating depending on how it performed, this rating is derived from the percentage of detections. For the negative tests, the categories range from bad to good, and are specified as: *bad* ¿= 0.2, -*high* ¿= 0.1, -*mid* ¿= 0.05, -*low* ¿ 0, *good* = 0. For the postive tests the ratings were specified as: *good* = 1,

+high ¿= 0.95, *+mid* ¿= 0.9, *+low* ¿= 0.8, *-low* ¿= 0.5, *-mid* ¿= 0.3, *-high* ¿= 0.1, *bad* ¿= 0.

3.5.3 Implementation for queries

9 different query statements were identified as occurring within OLTP-Bench. Out of these, the following 5 were found to be very rare:

- SET
- BEGIN
- ROLLBACK
- SHOW
- COMMIT

The remaining four consisted of the vast majority of queries (¿99%), and were therefore chosen for the feature vector. These are:

- INSERT
- DELETE
- UPDATE
- SELECT

Whilst the rare query statements could have been used for the feature vector, they seemed to unreliable as features. if 1 in 10 000 000 queries are consist of the statement SHOW, and the following iteration doubles this, it's not particularly interesting. However, with certain algorithms a doubling of a feature could result in workload shifts being detected. Whilst the chance of this occurring for the clustering algorithm chosen in this project was small, it was still considering a risk and therefore the rare query statements were discarded from the feature vector.

The actual workload detection was done via the DBSCAN clustering algorithm. This algorithm will always utilize all the data it has access too, and then divide it into a number of clusters that is deemed reasonable. When the model has been fitted to the data, the current and the previous iteration is predicted. If these two sequential iterations belong to different clusters, that means that the algorithm has detected a workload shift.

When the detector finds that a workload shift has been encountered, the detector forwards this information to the optimizer function, which then can reset optimization parameters and rerun the DoE phase.

During testing, it was found that this method did not adequately separate our workloads. Since the implementation of TPC-C and CH-Benchmark are similar, they have almost identical statistics for query percentages, since CH-Bench is essentially a TPC-C workload, combined with a few more complicated

transactions. Since these more complex transactions result in a a slowdown in terms of TPS, a natural divider between these was to use the total number of transactions in an iteration as a feature. This causes some issues however, as this feature varies from hundreds of thousands to millions of queries. When this is placed alongside the other features, which are in the form of statistics between 0 and 1, the DBSCAN algorithm could not adequately differentiate the workloads. There was therefore a need to normalize the number of queries into a smaller range. Since each iterations data needs to be normalized as it is collected, it is not possible to simply look at the maximum and minimum number of queries for all iterations. A new normalization method was then introduced, which normalizes the data over time by the following function, where n_i is the number of queries at iteration i, resulting in the normalized query measurement q_i.

$$q_i = \frac{n_i}{\frac{1}{i}\sum_j n_j}$$

The value of q_i then is a measurement of how much the query count for the current iteration differ from the average query count of all previous iterations. To avoid problems of consistency in the detection algorithm, all query data from previous iterations is retroactively updated so q_i is up to date with the most recent average query count value.

4 Results

4.1 Experimental Settings

The same set of software was used for all experiments, seen below in table 4

Table 4: *Software used for data collection. *Hypermapper was modified to allow for a workload detector. To see specifics, see section 3.1.2*

name	version
psutil	5.9.0
OLTP-Bench	Final
PostgreSQL	14.2
rsync	3.2.3
HyperMapper	2.2.9*

4.1.1 Optimal Configurations

The data regarding the optimal configurations for each of three workloads was executed over 2 repetitions, with each repetitions containing 60 iterations. Combined over all 3 workloads, this means 360 data points were collected. Since the basis of these tests was to examine the optimal configuration of each workload, there was no workload-detection data collected for the workloads.

The data was collected on an AWS instance with the following specifications:

Table 5: *AWS instance used for testing*

Region	Sweden, Stockholm (eu-north-1)
Instance Type	m5d.2xlarge, 64 bit
CPU	Intel Xeon, Platinum 8175M, 2.5GHz
RAM	DDR4, 32GB
Hard-drive	NVMe SSD 300 GB
OS	Ubuntu 20.04

4.1.2 Multi-Workload Optimization

The AWS instance that was used is the same as for the optimal configurations, which is detailed in table 5.

Each workload was run with the following three steps sequentially

- Design of Experiment - 10 iteration
- Optimization - 20 iterations
- Post-optimization - 5 iterations

This results in 35 iterations per workload, for 105 iterations total.

4.1.3 Queries

Data was collected separately for the queries to investigate the viability of these as a detection mechanism. To avoid unforeseen consequences with gathering both hardware-utilization and query data, hardware-utilization was disabled for collecting this data.

All data presented in section 4.3.1 was collected using an AWS server with the following specifications:

Table 6: *AWS instance used for testing*

Region	Sweden, Stockholm (eu-north-1)
Instance Type	m5d.xlarge, 64 bit
CPU	Intel Xeon, Platinum 8175M, 2.5GHz
RAM	DDR4, 16GB
Hard-drive	NVMe SSD 150 GB
OS	Ubuntu 20.04

These specifications is very similar to the one described in table 5, with the difference being that this one had half the RAM and storage space. Due to a lack of budget, this machine was used as it was less expensive to use.

For each workload, one full repetition of 50 iterations were executed, resulting in 150 data points total.

4.1.4 Hardware Statistics

This section defines all the parameters used for hardware utilization collection and hardware utilization detection. The parameters for the hardware collection part: the measurement cycle, the number of non-optimized and the number of optimized iterations. For the detection part: Number of positive and negative tests and which weights were used.

- The measurement cycle was set to five seconds, i.e the collections program would call and collect metrics from psutil every fifth second.
- The experiment for non-optimized data used 40 iterations for each workload. These iterations comes from 4 repetitions each containing 10 iterations.
- For optimized data, each workload was executed for 100 iterations, divided into 2 repetitions each containing 50 iterations.

For more information where this data comes from see section 3.2.2.

The experimental setup on workload detection used the discussed scheme in section 3.5.2. i.e, positive and negative tests and weights to discern hyperparameter scores. Go back to the referenced section to see exactly how the positive and negative test numbers were generated.

- For the non-optimized data in total 540 negative tests, and 1200 positive tests were executed.
- The optimized data used the same system for testing as the non-optimized data, but one should remember that a repetition of optimized data contains 50 iterations. In total this would generate 4900 negative tests and 15000 positive tests.
- For the detection of hyperparameters, the random search scheme was used. The random search would run for 100 iterations and then the best hyperparameters were extracted according to score. For the weights, a false positive gave -10 and a true positive +1.

The same AWS instance as in section Optimal Configurations 4.1.1 was used
.

4.2 Analysis

4.2.1 Optimal configurations

The experiment detailed in section 4.1.1 was executed, resulting in two complete repetitions for each of the three workloads. This resulted in the optimal configurations found in table 7, 9 and 11 for Wikipedia, TPC-C and CH-benCHmark respectively.

From these repetitions, the difference in performance between the initial DBtune configuration and the best configuration is compared. These can be found in tables 8, 10 and 12, again, for Wikipedia, TPC-C and CH-benCHmark respectively.

Table 7: *Optimal configuration obtained for 2 repetitions of the Wikipedia workload*

Knob name	Run-1	Run-2
shared_buffers (MB)	4096	6656
work_mem (MB)	13	27
random_page_cost	2	1.5
effective_io_concurrency	200	200
max_wal_size(GB)	24	24
max_parallel_workers_per_gather	16	4
max_parallel_workers	8	8
max_worker_processes	4	4
checkpoint_timeout (min)	5	5

Table 8: *Throughput per second for Wikipedia with the initial vs the best configuration for both repetitions*

	Run-1	Run-2
Initial config (TPS)	279	302
Best config (TPS)	361	378

When looking at table 7, it can be seen how quite a few knobs vary between the two repetitions, especially work_mem and shared_buffers. Most likely, this is due to either the knobs being unimportant for Wikipedia throughput, or that the repetitions didn't contain enough iterations, and therefore the optimal configuration wasn't acquired. Even so, there is more similarity than differences in the configurations which implies some level of consistency between the repetitions.

Wikipedia had a 29% and 25% boost in performance thanks to the optimization, which shows just how valuable it can be to optimize the server configuration, since this is essentially equivalent to 25% reduction in server costs.

Table 9: *Optimal configuration obtained for 2 repetitions of the TPC-C workload*

Knob name	Run-1	Run-2
shared_buffers (MB)	7680	8192
work_mem (MB)	40	5
random_page_cost	2.5	4
effective_io_concurrency	100	200
max_wal_size(GB)	64	64
max_parallel_workers_per_gather	8	1
max_parallel_workers	4	4
max_worker_processes	4	4
checkpoint_timeout (min)	10	10

Table 10: *Throughput per second for TPC-C with the initial vs the best configuration for both repetitions*

	Run-1	Run-2
Initial config (TPS)	1434	1481
Best config (TPS)	2045	2296

The story for TPC-C in table 9 is quite similar to the one discussed previously, table 7. Just like Wikipedia, max_wal_size is staying consistent between repetitions, even though the knobs has 7 different values that can be chosen between. Here, shared_buffers is staying much closer to each other. The knob work_mem behaves very similarly in both Wikipedia and TPC-C, and is wildly shifting between repetitions.

Table 10 shows that TPC-C experienced massive improvement, with a 40% and 55% boost in performance. Yet again, this shows the potential optimization methods like BO has for the future of the server space.

Table 11: *Optimal configuration obtained for 2 repetitions of the CH workload*

Knob name	Run-1	Run-2
shared_buffers (MB)	7168	7168
work_mem (MB)	6	50
random_page_cost	0.1	0.1
effective_io_concurrency	300	1
max_wal_size(GB)	32	16
max_parallel_workers_per_gather	1	4
max_parallel_workers	8	4
max_worker_processes	4	4
checkpoint_timeout (min)	10	10

Table 12: *Throughput per second for CH-benCHmark with the initial vs the best configuration for both repetitions*

	Run-1	Run-2
Initial config (TPS)	587	475
Best config (TPS)	634	583

Yet again, the work_mem knob varies massively between repetitions, as is seen in 11. Interestingly enough, shared_buffers and random_page_cost stay the same, which seem so imply that perhaps shared_buffers is more important for write-heavy workload like TPC-C, and less important for read-heavy loads such as Wikipedia.

Table 12 shows that there was some performance gain for CH, but not nearly to the same degree as for TPC-C. Here, the gain was between 8 - 22 %. The large difference in performance between these two seems to suggest that the number of iterations used was not adequate for getting close to the optimal configuration.

Table 13: *Table that shows the number of knobs that varied between the optimal configurations found.*

	wiki-1	wiki-2	tpcc-1	tpcc-2	ch-1	ch-2
wiki-1	0	5	8	7	8	9
wiki-2		0	8	7	8	8
tpcc-1			0	5	8	7
tpcc-2				0	7	7
ch-1					0	5
ch-2						0

When running DBtune for the three different workloads separately, it resulted in significantly different optimal server configurations, as can be seen in tables 7, 9, 11. Whilst differing optimal configuration were obtained between repetitions

of the same workload, there is a greater difference between different workloads, which is shown in table 13. Which knobs changed between repetitions also seemed to depend on what workload was being executed, which suggests that different knobs vary in importance depending on the workload.

The most stable knob that had different optimal values for different workloads was max_wal_size, which only shifted for CH-benCHmark.

Some knobs, such as max_worker_processes didn't change at all for any workload, which seems to imply that either the knob always has the same optimal value regardless of workload, or that the search-space was poorly designed.

4.2.2 Multi-Workload Optimization

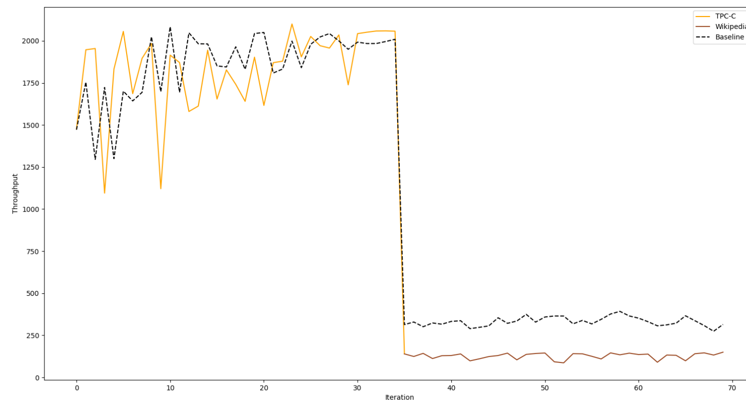

Figure 6: *Plot that shows the Baseline, which was fully optimized for TPC-C, versus the multi-optimizer system that optimized for TPCC, then Wikipedia separately when a shift was detected. The post-optimization phase can be seen to start at iteration 30. The workload shift to Wikipedia happens at iteration 35.*

The results for the comparison between the multi-workload optimization to a naive baseline can be seen in figure 6. Unfortunately, the performance between the two systems can not be directly compared, due to the fact that the baseline had more space available on the storage drive, which results in a performance gain compared to the multi-workload system. The reasons for this difference in storage space is detailed in section 4.4.3. Due to this performance gain, the exact value of the throughput as well as some other metrics cannot be used as a point of comparison. Despite this, some conclusions regarding the overall behaviour of the systems can be drawn from this plot.

Initially, the general behaviour for both systems is very similar for TPC-C, with the multi-workload system achieving a slightly higher performance. When the workload then shifts to Wikipedia, both systems experience a large drop in

throughput.

Neither seems to be able to optimize Wikipedia very efficiently, but the behaviour between the two systems can be seen to diverge. The naive system seems to experience a drop in performance towards the end of Wikipedia, an issue that is not experienced by the multi-workload system. When the post-optimization phase starts in the Wikipedia section, the performance for the multi-workload system gradually increases to reflect the change to the optimal configuration.

Why the naive-system seems to experience a drop towards the end of Wikipedia, even though the system isn't optimizing at that point is unknown.

4.2.3 Hardware Statistics

For brevity, most plots related to hardware utilization are not shown at all, and are instead presented in table format. Some of the metrics are also always zero, or otherwise uninteresting, and not included in the regular tables in the appendix. These metrics have their own table in the appendix, section 6.5, table 37.

The section is structured as follows: (1) First comparisons between the workloads, how the categorization data is related to the plots, and then the metrics are categorized into good and bad. (2) Results are shown from analysing 3D plots to see codependencies. (3) Statistics are shown that are independent for each workload.

Starting with the distance comparison between the workloads. The categorizations try to quantify the information that can be visually observed within the plots. Such metrics between the workloads are: (1) The "distance", which is the point-by-point mean distance between the two workloads. (2) The "inner STD", which is the point-by-point distance STD. (3) "Outer STD", which is the STD between repetitions when the data points have been aggregated into one data point from Euclidean Workload. (4) "intermingling", which is how much the workloads intersect. To see a more detailed description of these four categories see section 3.3.2.

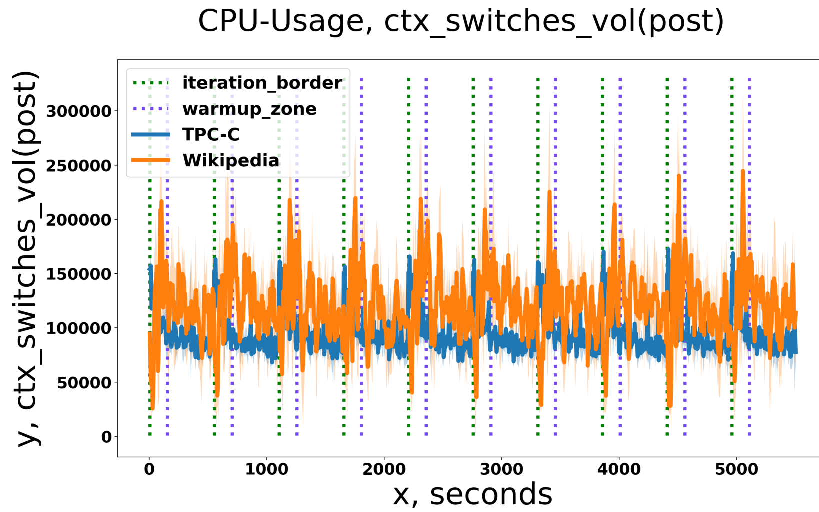

Figure 7: *Shows TPC-C and Wikipedia for the PostgreSQL isolated metric "voluntary" on a point basis. The green line shows where an iteration starts and from the green to the purple is the warm up zone of OLTP-Bench. The x-axis is showing the time-stamp when the point was collected. Note that the x-axis is concatenated time data as described in section 3.3.2 under category Timeline.*

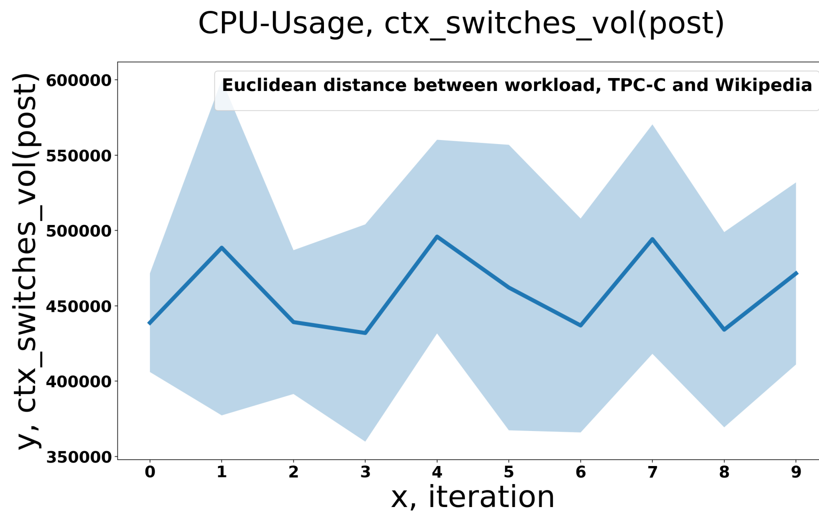

Figure 8: *Shows The Euclidean distance between TPC-C and Wikipedia for the PostgreSQL isolated metric "voluntary" on an iteration basis. Note that this data is generated from the category Euclidean Workload, see section 3.3.2.*

Table 14: *Showing 6 metrics which shows the most promising separation between TPC-C and Wikipedia. The categorizations have previously been described in section 3.3.2*

metric	distance	inner STD	outer STD	intermingling
- CPU -				
voluntary (P)	low	very high	low	very high
involuntary (P)	high	medium	low	non
- Virtual Memory -				
slab	high	low	low	non
vms (P)	medium	low	low	non
- Disk -				
read_count (P)	high	medium	low	non
- Net -				
bytes_recv	medium	high	medium	high

In figure 7 one can see a timeline plot generated from the category Timeline, see section 3.3.2. The plot shows TPC-C versus Wikipedia for the CPU metric "voluntary". As the documentation page of psutil describes it, voluntary is "The number voluntary [...] context switches performed by this process (cumulative).", for more such information see the psutil documentation [45] and to see all metrics used see table 30 in section 6.3. The plot shows that there is a frequent intermingling between both workloads, a small distance and a high variation in the point-to-point distance between the workloads, i.e a *high* "inner variation". The same information can be derived from table 14, which shows the previously stated distance categories on the metrics, "intermingling", "inner STD", ... Figure 8 once again shows TPC-C versus Wikipedia but plotted on an iteration basis, with the plot coming from the Euclidean Workload category, see section 3.3.2. Here one can see that the euclidean distance on average move from $[\approx 450000, \approx 500000]$, the confidence on separation should now be high. For the STD, it seems to make up around 20% of this distance between the workloads, which is categorized as "outer STD" in the tables.

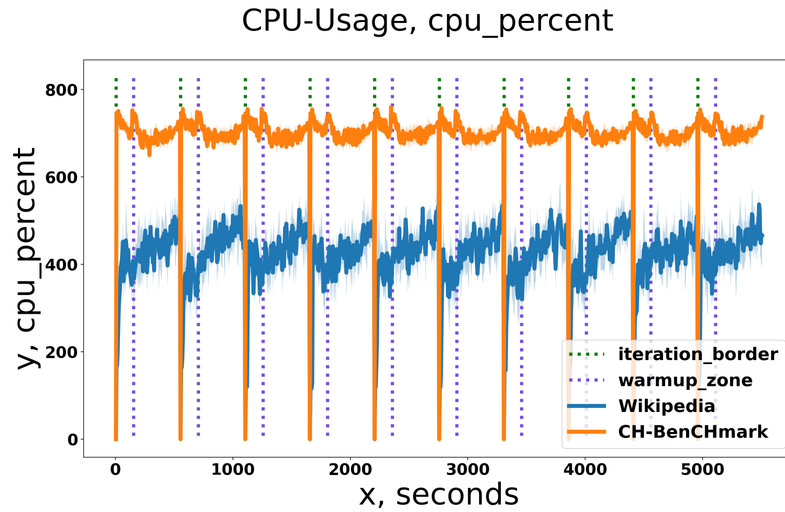

Figure 9: *"CPU-percentage" system wide Wikipedia versus CH-benCHmark . The green line shows where an iteration starts and from the green to the purple is the warm up zone of OLTP-Bench. The x-axis is showing the time-stamp when the point was collected. Note that the x-axis is aggregated time data as described in section 3.2.2.*

From the previous paragraph, one can understand that there is a difference, between taking the data on an iteration basis compared to taking the data on a point basis. Metrics that are classified in the tables with: "distance" = *high*, "inner STD" ¡ *high* and "intermingling" = *low* are usually very separated. For example, see figure 9 which plots Wikipedia versus CH-benCHmark and is categorized as: "distance" = *medium*, "inner STD" = *medium* and "intermingling" = *non*. From the figure, one can see that it has a constant and a large divide between both workloads. However, "intermingling" and "inner STD" are not the only categories which separated metrics well. Another set of good metrics are metrics categorized as: "distance" ¿= *medium* and "outer STD" = *low*. Such metrics can usually be separated well if one just aggregates the data to become one point per iteration. This can be seen by comparing the iteration based figure 8 to the point based figure 7. In this figure metric "voluntary" is categorized as: "distance" = *low* and "outer STD" = *low*.

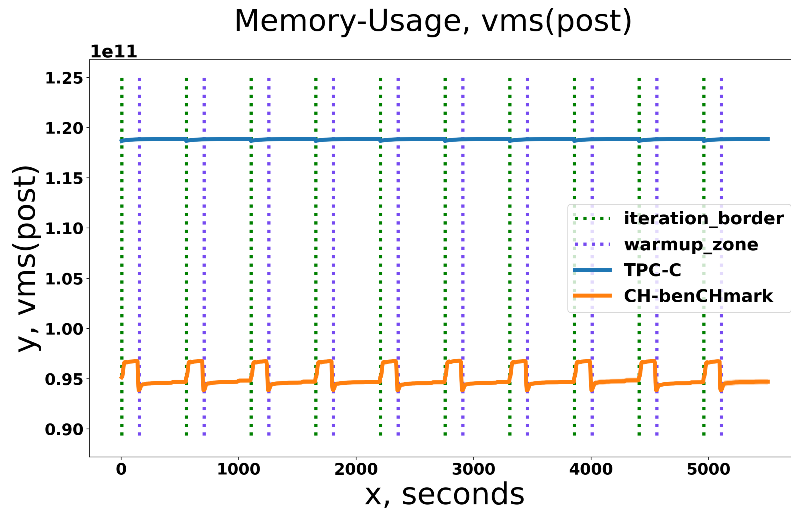

Figure 10: *Shows the PostgreSQL metric "vms (P)" for TPC-C versus CH-benCHmark . The green line shows where an iteration starts and from the green to the purple is the warm up zone of OLTP-Bench. The x-axis is showing the time-stamp when the point was collected. Note that the x-axis is aggregated time data as described in section 3.2.2.*

One also needs to keep in mind that even if the mean distance between two workloads is low they can still be separated as long as the STD is also low. Such metrics with: "distance" ¡ *high*, "inner STD" = *low* and "intermingling" = *non* are also great metrics for separation. For example, see figure 10. This figure plots the total memory consumption of the aggregated PostgreSQL processes for TPC-C versus CH-benCHmark. As one can see from the figure, the distance is quite close ≈ 0.2, which is around 20% of the lower lines mean value, but with zero STD from what can be observed, meaning a low "inner STD".

Table 15: *Showing 6 metrics which shows the most promising separation between TPC-C and CH-benCHmark. The categorizations have previously been described in section 3.3.2*

metric	distance	inner STD	outer STD	intermingling
- CPU -				
cpu_percent	high	low	low	non
user (P)	high	low	low	non
- Virtual Memory -				
slab	low	medium	medium	non
vms (P)	low	low	low	non
data (P)	high	low	low	non
- Disk -				
read_chars (P)	high	medium	low	non

Table 16: *Showing 6 metrics which shows the most promising separation between Wikipedia and CH-benCHmark. The categorizations have previously been described in section 3.3.2*

metric	distance	inner STD	outer STD	intermingling
- CPU -				
cpu_percent	medium	medium	low	non
user (P)	medium	medium	low	non
- Virtual Memory -				
cached	high	low	low	non
slab	high	low	medium	non
vms (P)	low	low	low	non
data (P)	high	low	low	non

To move on to the rest of the tables, there are two more tables like table 14, showing the other workload comparisons. All these tables show six important metrics for the separation of the two presented workloads in a table. The other two tables are TPC-C versus CH-benCHmark 15 and Wikipedia versus CH-benCHmark 16, which show the categorizations of the metrics as previously discussed. From the TPC-C versus Wikipedia table, one can see that slab seems to be a good metric, with *high* "distance" and *low* "inner" and "outer STD" and no "intermingling". To find all metrics, regardless of quality, see appendix 6.5 and the tables 31, 32 and 33 for TPC-C versus Wikipedia, TPC-C versus CH-benCHmark and Wikipedia versus CH-benCHmark.

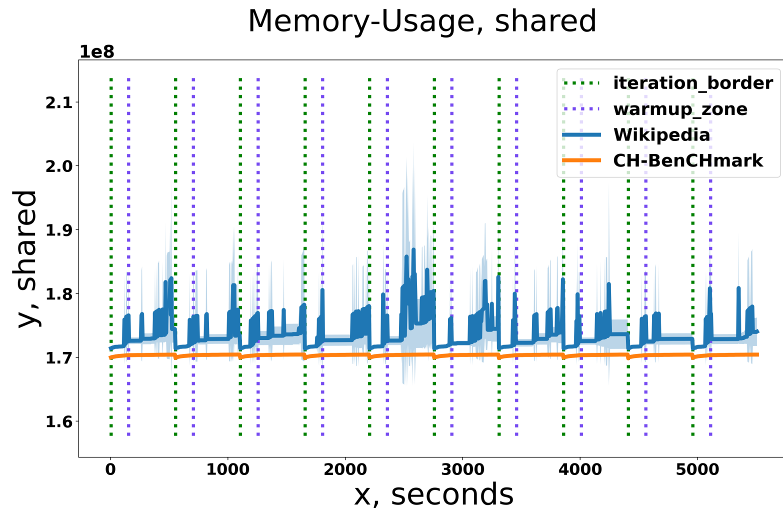

Figure 11: *Shows Wikipedia and CH-benCHmark for the system-wide metric "shared" on a point basis. The green line shows where an iteration starts and from the green to the purple is the warm up zone of OLTP-Bench. The x-axis is showing the time-stamp when the point was collected. Note that the x-axis is aggregated time data as described in section 3.2.2.*

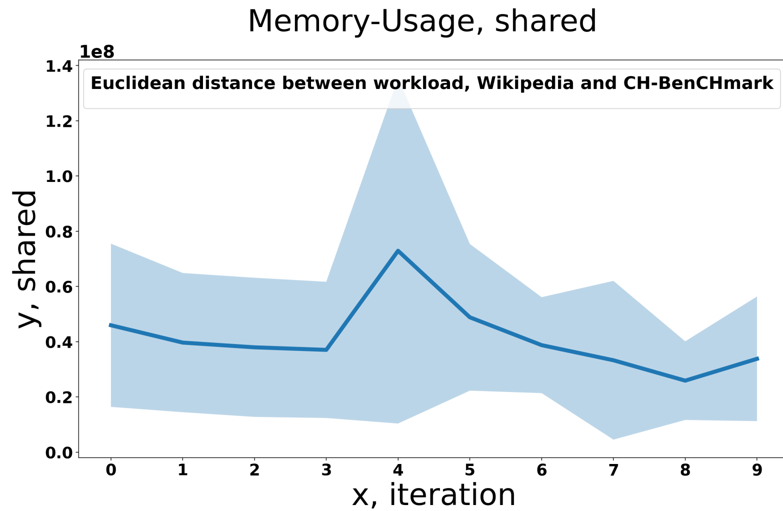

Figure 12: *Shows The Euclidean distance between Wikipedia and CH-benCHmark for the system-wide metric "shared" on an iteration basis. Note that this data is generated from the category Euclidean Workload, see section 3.3.2.*

For now, only metrics with good separation have been shown, but how does a bad metric look, with *high* STD in ”inner” and ”outer”, *low* ”distance” and *very high* ”intermingling”? Figure 11 shows Wikipedia versus CH-benCHmark for the metric ”shared”. One can see that the ”intermingling” is frequent and the ”inner STD” high. The ”outer STD” is also high and one can see this from figure 12, where the STD has a span of around 0.6 distance units, which makes up most of the mean distance.

To collect all the table data into one short simple table, summarizing the data, a table was crafted showing how many good and bad metrics there are, and to which category they belong:

1. The number of metrics which separate the workloads well when it comes to both point and iteration-based data. Point-based data is data that is not aggregated into a single iteration, i.e each iteration contains many data points. Iteration-based data is the opposite where each iteration contains only one data point, by euclidean distance.
2. The division of system-wide and PostgreSQL based metrics. A label shows which category is tallied, either post receptively system for PostgreSQL and the system-wide metrics.
3. The last row shows how many metrics are agreed upon to be well separated for all workload comparisons.

The inclusion criteria for a metric to be defined to have a ”well” separation can be found at table 17 and the summarizing table itself at 18. From the latter table, one can see that even though the PostgreSQL isolated metrics roughly makes up only 36% of the metrics they represent half of the good point-based metrics, this is especially prevalent in the all row where PostgreSQL isolated metrics make up 75% of the good metrics. One can also see from this table that on average, there exist ≈ 15 metrics which separate the workloads well for the point-based metrics. For the iteration-based, there exist on average 29 metrics that separate the workloads well. There are nearly double as many ”good” iteration-based metrics as point-based metrics. The PostgreSQL isolated metrics again make up a large portion of the iteration-based metrics, 50%, and again especially for the all row. One can also see from the table that when Wikipedia is included the amount of good metrics is drastically reduced.

Table 17: *Shows the inclusion criteria for point- and iteration-based data. The X means that the column is irrelevant to the selection criteria.*

distance	inner STD	outer STD	intermingling
point-based inclusion			
(low, medium, high)	(low, medium)	X	<= *low*
iteration-based inclusion			
(low, medium, high)	X	low	X
high	X	medium	X

Table 18: *Tallies how many metrics separate the workloads well for each separate workload comparison, and also shows how many metrics separate all workloads well. The table tallies the point-based and iteration-based data. The data is divided by the PostgreSQL isolated metrics (post) and the system-wide (system) metrics. The selection criterion to include point- respectively iteration-based data can be found in table 17.*

workloads	all point	post point	system point	all iteration	post iteration	system iteration
In total 53 metrics, 19 PostgreSQL						
TPC-C, Wiki	14	9	5	26	12	14
TPC-C, CH	19	8	11	32	17	15
Wiki, CH	13	5	8	29	12	17
all	4	3	1	18	11	7

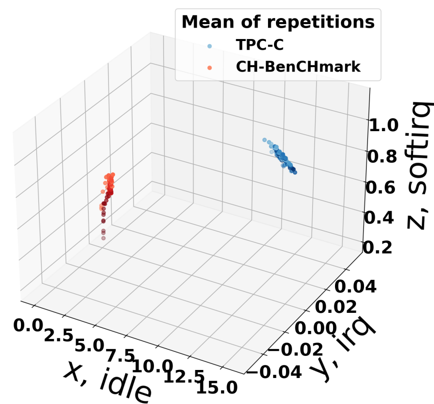

Figure 13: *3D mean plot of idle, irq and softirq. Red is CH-benCHmark and blue TPC-C workload.*

his paragraph signals the end of the distance comparisons and shows the start of the codependency analysis. An important metric is codependencies, if one wants to improve the collection system as much as possible, it is unnecessary to collect perfectly correlated metrics. One note though, the codependencies were purely discovered visually, by analysing the 3D plots. What type of 3D plots were generated can be seen in section 3.3.2. Once again the plotting information is quantified by categorisation, but this time generated visually. The range is from -high to +high where ±high means near-perfect negative or positive correlation. ±Medium and ±Low means medium respectively low correlation. In

table 19 such codependencies are shown. The correlating metrics are displayed in the metrics column, and then in the rest of the columns, it is shown how much each workload correlates with the corresponding metrics. For example, in the table, one can see that "idle" and "softirq" has a high negative correlation for the TPC-C workload, which also can be viewed in figure 13. In the figure, one can see that as "idle" increases "softirq" nearly linearly decrease for TPC-C, meanwhile, for CH-benCHmark which is reported to not correlate, most of the time "softirq" moves in a straight line upwards whilst "idle" remains stagnant.

Table 19: *The table shows codependencies, between the metrics for all workloads. The first column shows which metrics correlate, and the rest shows how much the metrics correlate in each workload. CH = CH-benCHmark.*

metrics	TPC-C	Wikipedia	CH
- CPU -			
5min, 15min	+high		
idle, softirq	-high	-low	non
soft_interrupts, voluntary	non	+high	non
soft_interrupts, involuntary	+high	non	non
- Virtual Memory -			
buffers, shared	+medium	+low	non
buffers, cached	+medium	non	non
cached, shared	+medium	non	non
slab, shared	+low	+medium	non
rss, vms	+low	+low	+medium
- Disk -			
bytes_recv, packets_recv	+high	+high	+high

Table 20: *Shows some selected individual statistics from all the workloads, which represents some key strokes of the rest of the metrics within the same workload. The table shows the categorization of: (1) The mean distance between non-optimized data and optimized data. (2) The mean-STD of non-optimized data. (3) If the non-optimization is biased towards the starting iteration of the workload. (4) If the distance between the non-optimized and the optimized data trend in any direction. (5) The variance of the trend. What each category, like high and low represents, and how the categories were generated is described in section 3.3.2.*

metric	distance	STD	bias	growth	growth variance
- TPC-C -					
1min	high	medium	high	+very high	low
slab	low	low	low	+high	low
vms (P)	high	low	low	+high	low
- Wikipedia -					
slab	low	low	low	-low	low
vms (P)	high	low	low	-low	low
- CH -					
shared	high	low	low	+medium	medium
vms (P)	high	low	medium	+medium	medium

Finally, the individual performance results for each workload. For example, an important ingredient in finding a good metric is how it compares to optimized and non-optimized data. For these type of measurements, there are one table showing a few metrics for each workload, table 20, the full description of this table can be found in appendix 6.5 in three tables, 34 for TPC-C, 35 for Wikipedia and 36 for CH-benCHmark. These tables shows the categorization of: (1) "distance", the mean distance between optimized and non-optimized data, generated exactly the same as for the non-optimized data but on different data. (2) "STD", the mean-STD of non-optimized data. (3) "bias", if the non-optimized data is biased towards the starting iteration of the workload. (4) "growth", if the distance between the non-optimized and the optimized data trend in any direction. (5) "growth variance", the variance of the growth. What the categorization means, and how the categorizations were generated are described in section 3.3.2.

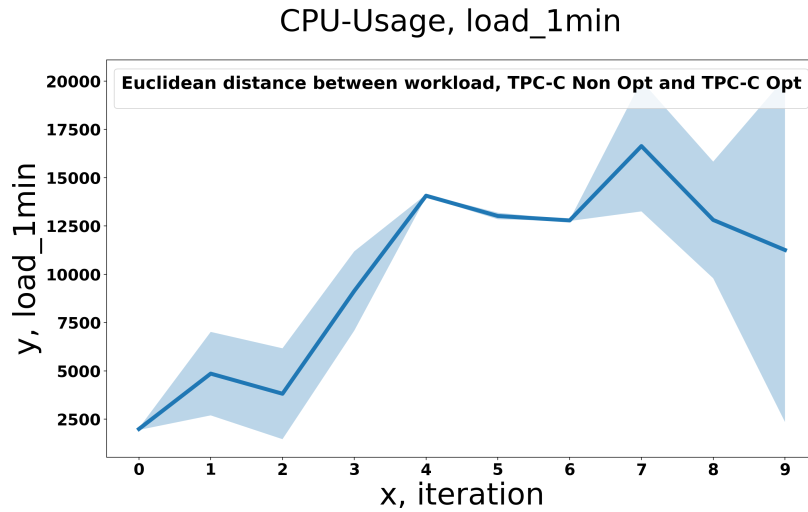

Figure 14: *The Euclidean distance between TPC-C optimized and non-optimized data for the metric "1min" on an iteration basis. Note that this data is generated from the category Euclidean Workload, see section 3.3.2.*

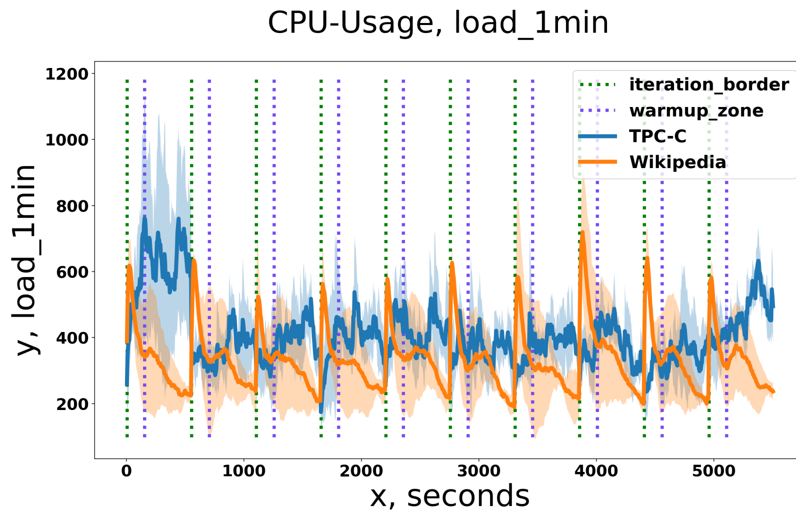

Figure 15: *TPC-C versus Wikipedia for the metric "1min" on a point basis. The green line shows where an iteration starts and from the green to the purple is the warm up zone of OLTP-Bench. The x-axis is showing the time-stamp when the point was collected. Note that the x-axis is concatenated time data as described in section 3.3.2 under category Timeline.*

From table 20 one can see that TPC-C has a *high* "growth" which is symp-

tomatic for the whole workload. The "bias" is *low* for TPC-C and Wikipedia for all metrics except "1-", "5-" and "15min". CH-benCHmark is different and has a *medium* "bias" in metrics "vms (P)", "buffers", "num_fds (P)" and *high* in "data (P)". For the "growth" parameter, TPC-C *has* a high to *medium* leaning on most metrics, CH-benCHmark a *medium* to *low* and Wikipedia *low* only. Figure 14 shows TPC-C plotted for metric "1min" when taking the Euclidean distance between itself on optimized and non-optimized data. The metric "1min" has a *very high* positive "growth" with *low* "growth variance". One can see from the plot that "1min" starts low with low STD and then quickly climbs without much STD between the repetitions generating a *very high* "growth" with *low* "growth variance". There is no specific plot generated to visualize the "bias" but it can for example be seen by a timeline plot, see figure 15. Here one can see that the first iterations are much higher than the rest for the "1min" metric of TPC-C.

4.3 Workload Detection

4.3.1 Query-based Statistics

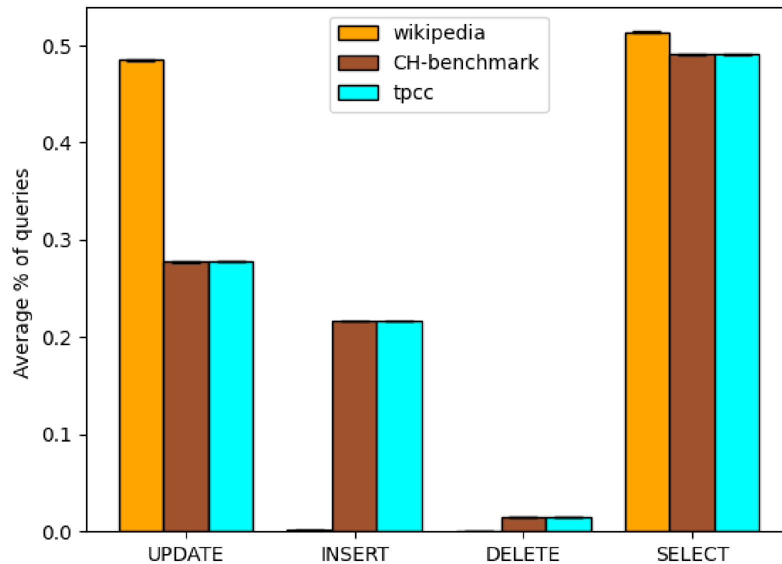

Figure 16: *Plot of percentage of queries for each of the three workloads. Note the small lines in the middle which signify the confidence interval.*

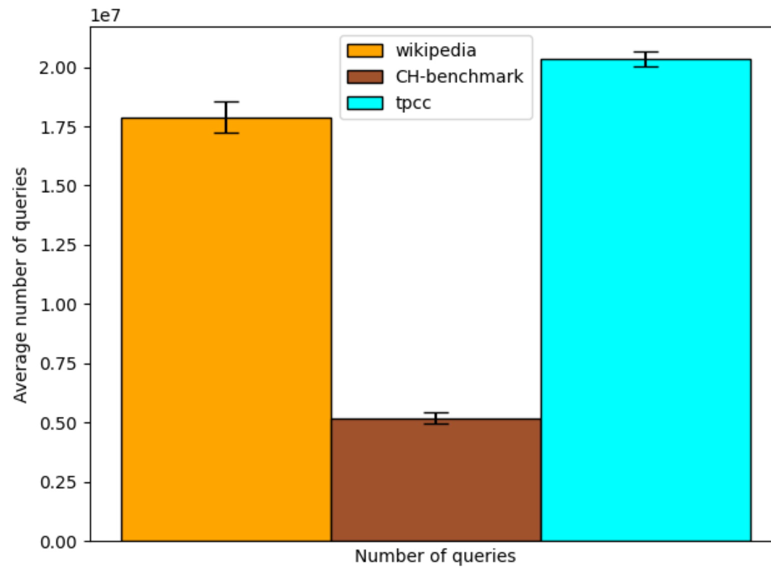

Figure 17: *Plot of total number of queries for each of the three workloads. Confidence interval included.*

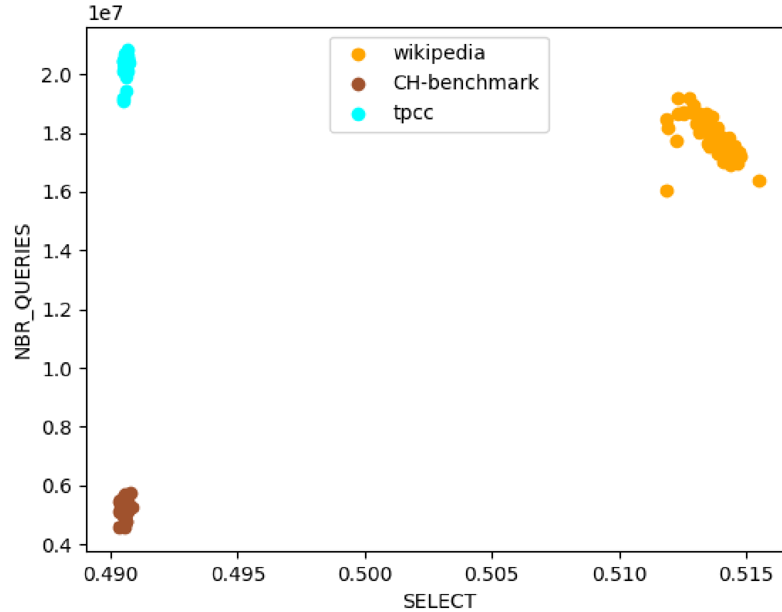

Figure 18: *Scatter plot of all data points used, plotted for number of queries and percentage of SELECTS. Note that the number of queries has not been normalized according to section 3.5.3*

From figure 16, it can be seen that Wikipedia can be quite clearly separated from TPC-C and CH-benCHmark. Wikipedia has almost twice the percentage of UPDATE, and practically no INSERT queries when compared to the other two workloads. The standard deviation is also so low that Wikipedia can be reliably distinguished.

What the figure also shows, is just how similar TPC-C and CH-benCHmark are compared to each other. In every query-based statistics that was collected, they are practically identical.

As figure 17 suggests, the number of queries seems to separate CH-benCHmark from Wikipedia and TPC-C. By plotting the data points collected for percentage of SELECT and number of queries, the three workloads are well separated, which can be seen in figure 18

4.3.2 Hardware Statistics

In this section the found results from using the detection algorithms in 3.5.1 will be presented. The section starts of with showing the results on non-optimized data and then on optimized data. The scores are generated from the search script see section 3.5.2 and the weights and amount of tests is defined in section 4.1.4.

The results on non-optimized data is shown in two parts, one part without pruning and one with. The first shown result is generated by just taking the found hyperparameters from the search script, without any feature pruning. Then after, some metrics will be pruned by analysis of the results, and then the score tallied again.

The scores for the different detection methods were 1200, 780 and 0 for Euclidean, CUSUM-chart and MI respectively. The following hyperparameters were used: (1) 5.018 for the Euclidean distance hyperparameter. (2) 6.83 and 0.95 for distance and the percentage of allowed outliers for CUSUM-chart. (3) The difference hyperparameter was set as 0 for MI.

Table 21: *Shows the metrics categorization for non-general metrics for CUSUM-chart workload detection on the negative test for non-optimized data, see a more detailed description of the categories in section 3.5.2. The general metrics table can be found in appendix 6.6 and tabel 38. In the test, the hyperparameters, i.e the distance and the allowed outliers were set to 6.83 and 0.95. The columns in the table show how the metrics performed for each workload when doing self-detection.*

metric	tpcc	wikipedia	CH-benCHmark
- CPU -			
num_fds (P)	good	-low	-low
- Virtual Memory -			
shared	-low	-high	good
data (P)	good	good	-mid
vms (P)	good	good	-low

Every metric for all detection methods had zero false positives, see the general table of the detection results in appendix 6.6 table 38. The exception to this is CUSUM-chart, which had 47 false positives. Mostly these false positives are related to Wikipedia and the metric "shared", "shared" which is categorized as a bad metric in the hardware analysis for all workload comparisons involving Wikipedia. To see the categorizations of the other metrics on CUSUM-chart see table 21.

Table 22: *Shows some selected good metrics for the categorization of Euclidean and CUSUM-chart workload detection on the positive test for non-optimized data, see a more detailed description of the categories in section 3.5.2. For the hyperparameters, the Euclidean distance hyperparameter was set to 5.018. For CUSUM-chart's hyperparameters, they were set to 6.83 and 0.95 for the distance respectively the percentage of allowed outliers. The columns in the table show how the metrics performed for each workload versus another workload during the detection.*

metric	tpcc vs wikipedia	tpcc vs CH-benCHmark	wikipedia vs CH-benCHmark
*** Euclidean ***			
- CPU -			
idle	+high	+high	bad
- Virtual Memory -			
data (P)	bad	+high	+high
cached	+high	bad	+high
- Disk -			
read_chars (P)	bad	+high	bad
*** CUSUM-chart ***			
- Virtual Memory -			
data (P)	-low	+high	+high
cached	+high	bad	+high
rss (P)	+high	+high	-high
shared (P)	+high	+high	bad
text (P)	+high	+high	bad
vms (P)	+high	+high	+high

The results are much more varied for the positive test. The Euclidean and CUSUM-chart did not produce a single false negative. For MI the opposite was true, every single iteration was misclassified and in total 1 200 false negatives were produced. A general table can be seen in appendix 6.6, table 39, which shows all metrics agreed upon as being bad by the detection methods. For the individual Euclidean and CUSUM-chart tables see table 22, which shows a selection of good metrics for both detection methods, the MI table is omitted since every single metric is just bad. For the full tables for Euclidean and CUSUM-chart see appendix 6.6 and tables 40 respectively 41. From all these tables one can see that many metrics have a *+high* detection rate when detecting different workloads, and one should remember that these metrics also have zero false positives.

For the pruning part, the Euclidean can remain untouched, CUSUM-chart only needed to remove metrics defined in table 21 to achieve a full score, 1 200, but for MI things were a bit different. MI seems to be sensitive to bad metrics, bad metrics will be classified over and over again as a workload shift for both the positive and negative tests. Therefore 0 likely was given as the best

hyperparameter, making all metrics bad for the positive test but generating zero false positives for the negative test. By running MI for any other hyperparameter value, one can easily discover which metrics are bad. To try this the difference hyperparameter was set to 0.5. The resulting bad metrics from the negative test are shown in appendix 6.6 in table, 42. By this simple pruning of the bad metrics, the score is improved to 105, which is a low score, the hyperparameter was set to 0.6283. The problem seems to be that the bad metrics for the negative test overlap with the good metrics for the positive test, and therefore MI can not increase its score further.

Table 23: *Shows the pruned metrics for the Euclidean and CUSUM-chart during testing on optimized data.*

purged metrics
*** Euclidean ***
- CPU -
"idle", "num_fds (P)", "1min", "5min", "steal"
- Virtual Memory -
"shared", "vms (P)", "buffers", "rss (P)", "shared (P)"
- Disk -
"read_chars (P)", "read_count (P)", "write_count (P)", "write_time"
*** CUSUM-chart ***
- CPU -
"num_fds (P)", "1min", "5min", "15min",
- Virtual Memory -
"shared", "data (P)", "vms (P)", "cached", "slab", "buffers", "rss (P)", "shared (P)"
- Disk -
"read_chars (P)", "read_count (P)", "write_count (P)", "read_bytes (P)", "write_chars (P)"

Table 24: *Shows a selection of the non-pruned metrics categorization for Euclidean and CUSUM-chart workload detection on the positive test for the optimized data, see a more detailed description of the categories in section 3.5.2. In the test, the hyperparameters were: 5.018 for the Euclidean distance and for CUSUM-chart the distance and the allowed outliers were set to 6.83 and 0.95. The columns in the table show how the metrics performed for each workload versus another workload during the detection.*

metric	tpcc vs wikipedia	tpcc vs CH-benCHmark	wikipedia vs CH-benCHmark
*** Euclidean ***			
- CPU -			
involuntary (P)	-mid	bad	+high
system (P)	+low	bad	+high
- Virtual Memory -			
data (P)	bad	+high	+high
- Disk -			
read_bytes (P)	+high	bad	+mid
*** CUSUM-Chart ***			
- CPU -			
busy_time	+high	bad	-mid
user (P)	bad	bad	-high
- Virtual Memory -			
text (P)	+high	+high	bad
- Disk -			
read_time	-low	bad	+high

On optimized data both Euclidean and CUSUM-chart achieved good scores after considerable feature pruning, but MI did not compete. The enacting of the feature pruning viewable in table 23, on CUSUM-chart, considerably increased the score. Before pruning, the score was around 11 000, after, CUSUM-chart achieved a score of 15 000 on the optimized data, which is the maximum. i.e no false positives or false negatives. The score decreased a bit though, on the non-optimized data after the pruning. The score became 1 094, on non-optimized data. The hyperparameters were not reinvestigated after the pruning though, due to lack of time. Some categorizations can be viewed in table 24 and all in appendix 6.6 at table 43. The Euclidean had a more drastic improvement after the pruning. Before the pruning, the score was negative, -1 000 on the optimized data. After the pruning, the score reached 14 985, nearly perfect. On the non-optimized data, the Euclidean kept its full score of 1 200. Also for the Euclidean, the hyperparameters were not reinvestigated. The pruned metrics can be viewed in table 23 and some categorizations at 24 and all in appendix 6.6 at table 44. For MI the same results as previous were achieved, a score of 0.

From the previous paragraph one can see that if you would use CUSUM-chart during optimization and Euclidean once the optimization is done, one would

have a perfect detection algorithm for the tried data.

4.4 Discussion

4.4.1 Regarding credibility of results

For the reader to be able to trust the results, the tests must be realistic. Unfortunately, it cannot be concluded that the tests used in this project are so. Whilst the authors have done their best to select industry-standard workloads to test multi-workload optimization, the fact of the matter is that it still does not quite align with reality. The test setup consists of three databases chosen from OLTP-bench, which are then compared against each other. Due to limitations, there has not been any validation to see if this setup is a good simulation of a real-world scenario. One can imagine that one database suddenly swapped for another is unrealistic though. In the database it means that all tables and queries are switched out in a fraction. One could argue that the 10-minute iteration period could represent any time, and when the system measures then a great deviation between the workloads has already happened. The problem with this is that a complete change in everything in the database likely does not represent a slow or fast workload change, if it would, then the workload shift should have been detected a lot earlier. For these purposes, much of the project should be considered a proof-of-concept, rather than a realistic analysis of workloads in real-world scenarios.

4.4.2 Optimal Configurations

As table 13 shows, CH has a different optimal configuration than TPC-C, but when looking at figure 16, the two workloads seem to be doing the exact same thing. The implication of this is that these two are truly two different workloads, and that query-percentages does not fully characterize the workloads. This proves that expansion of the feature vector is necessary, and motivates the addition of the number of queries as a feature.

One thing that should be touched upon is how vital it is that table 13 demonstrates that each workload has a different optimal configuration. If the optimal configurations had been the same it would mean that the current workload is essentially irrelevant for the optimizer, since it would still converge to the same optimal point regardless of the workload. For this reason, it was crucial that the workloads converged to different points, as if they hadn't there would have been no purpose of multi-workload optimization.

4.4.3 Multi-workload Optimization

As mentioned previously, the two plots cannot be compared quite apples-to-apples for multiple reasons.

One reason is how the load was placed upon the system. While collecting the data for the multi-workload system, after the workload switched over from Wikipedia to CH-benCHmark, it started behaving strangely, with many iterations

failing and performance becoming irregular. For the sake of transparency, a plot that includes this data can be found in the appendix under figure 19. A bug was also found that seemed to cause a sort of storage-leak that loaded more and more data into storage over time, until the drive was entirely full. This caused the CH-benCHmark to eventually crash, which is why the aforementioned figure has less iterations under CH-benCHmark. Unfortunately, due to a lack of time and monetary budget, the multi-workload system could not be started again with the bug fixed.

Due to the instability CH-benCHmark seemed to cause, when data was to be collected for the baseline, it was decided that CH-benCHmark would be ignored and instead only TPC-C and Wikipedia would be used in the naive test. Unfortunately, it turns out that SSD performance, which is the type of storage used for the AWS instances chosen, is highly dependant on the amount of data loaded into it. The more data loaded, the slower it will be. Therefore, the decision to only load Wikipedia and TPC-C into storage results in Wikipedia achieving a much higher throughput on the naive system as compared to the multi-workload system, and makes overall analysis unreliable. This is why there cannot be a head-to-head throughput analysis. As a consequence of this, it means that the possible performance gain of using multi-workload optimization cannot be truly calculated. From looking at table 8, some conclusions could be drawn however. Here, an average throughput of 370 was reached. By looking at figure 6, it can be calculated that the naive baseline achieved an average throughput of 332. Whilst this isn't a perfect comparison, it still shows that by properly optimizing for Wikipedia can result in at least an 11% increase in performance, over having an optimal configuration from a different workload.

4.4.4 Query-based Statistics

When looking at figure 16, the similarity between TPC-C and CH-benCHmark may seem somewhat chocking, considering how different workloads they really are. The reason for this is almost certainly that CH-benCHmark is a combination of a TPC-C workload, as well as a trickier OLAP-workload. Since OLAP consists of far fewer, but more difficult transactions, it means that in terms of query percentages, the OLAP part barely makes a dent in the statistics, which means that CH-benCHmark cannot be distinguished from TPC-C using only these statistics. This was the motivation for why the number of queries needed to be considered, since it makes sense that more complex queries would result in less throughput overall, which is also what can be seen in figure 17.

One thing that must be discussed is the increased variability and magnitude of the number of queries compared to the other feature vectors. Since this feature has a very different variance and base value, it can cause problems for many different clustering algorithms. The clustering algorithm used for query-based detection in this project, DBSCAN, is as previously described reliant on the hyperparameter *eps*. This hyperparameter controls how far apart points can be, while still being considered to be in the neighbourhood of one another. Since the number of queries varies much more than the percentage of queries, that

means it has a more dominant role in classifying workload shifts. The result of this is that the hyperparameter needs to be set very carefully, or the system will be only reliant on the query count. By looking at 18, it can then be seen that TPC-C and Wikipedia would not be able to be adequately separated from each other if you were to only rely on this feature. For this project, *eps* set to 0.15 was found to be sensitive enough to divide workloads, without classifying false workload shifts.

Initially the idea was to run the query-detection system on the data collected for the multi-workload system, which is discussed in section 4.2.2. Unfortunately, the bug that was described in section 4.4.3 also corrupted the query data that was collected. It is for this reason there is no analysis regarding the complete multi-workload system queries.

4.4.5 Hardware Statistics

The separation of workloads on non-optimized data were very promising. Many metrics could separate the workloads well, and a few were agreed upon to separate all workloads well. Both point-based and iteration-based data performed well. Especially the iteration-based data. Promising detection methods which have no problem with such aggregations easy detection. Methods like CUSUM-chart would have a hard time with such aggregations, though. Even if one re-implemented CUSUM-chart with aggregations in mind, it would take many iterations before being useful for detection. This is because CUSUM-chart relies on the deviation from the mean.

Another promising result from hardware utilization is that many good metrics come from the PostgreSQL isolated metrics. This is a good result, looking back into earlier sections, one can see that OLTP-Bench occupy a lot of resources for some workloads. For example, when running Wikipedia a large chunk of the system's RAM is occupied. If this was not the case, if PostgreSQL isolated metrics would not separate the workloads well. Then it would be hard to know if the collected data showed an actual difference between the workloads or in how OLTP-Bench runs them. Now both PostgreSQL isolated metrics and system-wide metrics agree that the workloads can be separated, even better an unproportionate amount of the metrics agreeing that the workloads can be separated are from the PostgreSQL isolated metrics.

From the multi-workload Optimization section 4.2.2, and the discussion 4.4.3, one can see that the SSD is severely hampered when being close to being full. This, unfortunately, invalidates the category "distance", when comparing the optimized and non-optimized data. This invalidation occurs, since the optimized data would only have one of the three workloads loaded at a time, meanwhile the non-optimized data would have all loaded while running the workloads. Making it unknown if the distance is an effect of the actual distance between the optimized and non-optimized data, or just an artefact of how the benchmarking was done. The authors would suggest that it at least seems to depend on the workload, if one looks at the Wikipedia data, there is no growth at all between the iterations when comparing between optimized and non-optimized data. It

is just a constant distance, which is very odd, since if the configuration files would greatly affect the hardware utilization metrics, then some growth trend or high growth variance should be generated, but this is not the case. From the multi-workload optimization, it seems that this workload is hard to optimize. TPC-C seem to be the workload most affected by the optimization, one can also see the largest growth for TPC-C. Still, even in TPC-C many metrics show a low "growth" and "growth variation", indicating that some metrics might work well in optimized and non-optimized scenarios of TPC-C. Regardless, one would not need to run workload detection during optimization, since the DBMS controls this phase, the user knows when it starts and ends. Therefore workload detection could be started after optimization, though creating the risk of missing a workload shift during the optimization.

For codependencies, nothing special was discovered. TPC-C seems to have more codependencies than any other workload, maybe making it easier to optimize than the others. Also as previously discussed and which one should note, the codependencies are only investigated between a few selected metrics, see section 3.3.2 for more information.

For the detection algorithms, Euclidean and CUSUM-chart performed very well on non-optimized data reaching a maximum score of 1 200. Many metrics showed a detection rate of 95% for several workload comparisons and with zero false positives for the same metrics. One interesting fact is that the supposed naive function, Euclidean, needed no feature pruning at all, which was needed for CUSUM-chart. Likely since the Euclidean can fully exploit the iteration-based metrics, which CUSUM-chart can not. For MI things did not look as bright. Even after feature pruning, MI could only reach a score of 105. The problem is that MI thinks that workloads of the same type are completely unrelated for many metrics. From the above discussion, one could easily suggest that just keeping with the Euclidean for all non-optimized workload detections is the best. Since this method is the simplest and seems to perform the best.

For the optimized data, the Euclidean and CUSUM-chart produced promising results. Both methods achieved a near-perfect score on their own, and if combined a perfect score would be achieved on both optimized and non-optimized data. It was surprising that CUSUM-chart succeeded. From previous discussions, it has been said that the SSD being full or not, severely affects database performance, and as discussed, earlier in this section, it also seems to greatly affect hardware metrics as well. Therefore a method like CUSUM-chart, which tries to detect deviations from the mean should fail if a constant artificial divide is set up between the workloads. But this error in the tests seems not to be great enough to destroy the natural separation of the workloads.

5 Conclusions & Future Work

5.1 Conclusions

When optimizing for workloads individually, it was found that TPC-C benefited greatly from optimization, with a boost in performance around 50%, followed by Wikipedia which achieved a 25% gain, and lastly CH-benCHmark, which experienced a gain of 8-22% .

The program that was created for multi-workload optimization is functional, and is capable of utilizing detection algorithms to restart optimization. The increase in performance for workloads in this system didn't match the gain experienced during single-workload optimization, most likely due to bugs and unforeseen consequences of taxing SSD storage.

In terms of workload detection, multiple systems were devised that all seemed to solve the issue. It was found that for query-based statistics remained very stable throughout optimization. The statistics differed significantly between workload, either via direct percentages of query types, or via the normalized number of queries in an iteration. The queries could therefore be used for workload detection, regardless of if the program is in the DoE, optimization, or post-optimization phase.

For the hardware metrics, the result was similar. When used in non-optimizing scenarios, it was trivial to separate the different workloads for most methods, all succeeded except MI. Dozens of features were identified that were statistically different between each of the three workloads. For optimizing scenarios, also a perfect detection was achieved, even with some testing setbacks. Another finding by analyzing the optimized data was that one could see, for example, that the workload Wikipedia seems unaffected by the optimization, in terms of hardware metrics.

5.2 Future Work

As this projects has dealt with a new area of research, there are many different fields that could be looked into. One key idea that would vastly simplify future work would be the creation of benchmarking software specifically designed for continuous database optimization and workload shifting. If this could be designed in such a way that complicated queries and gradual workload transitions were implemented, then this would give much more confidence in continuous optimization methods and their applicability in real-world scenarios, as well as result in more stable software that would accelerate testing in the field.

Another area of research is workload characterization and detection. Whilst hardware and query data certainly are capable of distinguishing workloads for this project, there may be more robust ways that are more capable of dealing with gradually shifting workloads. To know the most reliable ways to distinguish between any types of workloads clearly is a massive boon for the field of DBMS optimization. Further research within workload characterization could lead to even more exciting discoveries, such as transfer-learning, where data obtained

from previous optimization of workloads could somehow help inform what the most likely new optimal configuration is.

Further research can also be done regarding the applicability of the mutual information between optimization sessions on different servers. It could be possible that two servers with differing hardware may still have the same optimal configuration. If this could be identified, then it could make optimization sessions much faster, since eventually all different workloads and their estimated optimal configuration would be known, making optimization of an entirely new server trivial.

One aspect which is not touched upon in this thesis is optimization as a whole. For example, what would performance-wise be the most optimal detection cycle, detection every fifth minute, tenth and so, which resources to use, query, database statistics, hardware utilization? Should one utilize a less resource-intensive setting, checking key metrics, and once these go out of a bound, activate the whole measurement system? For such decisions, the detection performance must be compared with the impact on database performance from utilizing these detection programs. Another aspect, not researched is hardware measurement time cycles and excluded metrics. How often should one collect data points? Should excluded resource-intensive metrics be included, and the measurement cycle increased? As one can imagine the possibilities are endless. Other aspects could also be taken into account, for example in some cases even though a new workload has been found, does one want to start over the optimization process? When is the optimization necessary. For example, if a new workload is detected, but system utilization is very low, maybe an optimization is not needed. Maybe in such cases, some other system could decide if de-allocation could be done. All such research would provide insight-fullness into how to do an optimal optimization.

Even though queries were utilized, PostgreSQL provides many more such database metrics, and third-party tools provide even more. For example, many metrics generated by the database itself are missing, like shared_blks_written. In future research such metrics should be included, with their performance impact, this would help determine which metrics are the most important.

References

[1] D.V. (1) Aken et al. “Automatic database management system tuning through large-scale machine learning.” In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Vol. Part F127746. SIGMOD 2017 - Proceedings of the 2017 ACM International Conference on Management of Data. (1)Carnegie Mellon University, 2017, pp. 1009–1024.

[2] Gsmith et. al. *Monitoring*. `https://wiki.postgresql.org/wiki/Monitoring`. Accessed: 2022-05-24. postgreSQL wiki.

[3] S. Alabed and E. Yoneki. “High-Dimensional Bayesian Optimization with Multi-Task Learning for RocksDB.” In: University of Cambridge, Cambridge, UK, 2152, 2021.

[4] Pavlo Andy and et al. *OLTPBench*. `https://github.com/oltpbenchmark/oltpbench`. Accessed: 2022-02-8.

[5] ANSI. *The SQL Standard – ISO/IEC 9075:2016 (ANSI X3.135)*. `https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/`. Accessed: 2022-02-15.

[6] Cuadra-Sánchez Antonio and Aracil Javier. *Traffic Anomaly Detection*. Ed. by Abdelhamid Mellouk. 2015.

[7] Souza Artur L. F. et al. “Bayesian Optimization with a Prior for the Optimum”. In: *CoRR* abs/2006.14608 (2020). arXiv: `2006.14608`. URL: `https://arxiv.org/abs/2006.14608`.

[8] Bishop Christopher M. *Pattern Recognition and Machine Learning*. 6 Springer Science+Business Media, LLC, 2006.

[9] Richard Cole et al. “The Mixed Workload CH-BenCHmark”. In: *Proceedings of the Fourth International Workshop on Testing Database Systems*. DBTest ’11. Athens, Greece: Association for Computing Machinery, 2011. ISBN: 9781450306553. DOI: `10.1145/1988842.1988850`. URL: `https://doi.org/10.1145/1988842.1988850`.

[10] Ubuntu Community. *rsync*. `https://help.ubuntu.com/community/rsync`. Accessed: 2022-05-5.

[11] Oracle Corporation. *What Is a Database?* `https://www.oracle.com/database/what-is-database/`. Accessed: 2022-03-29.

[12] *DB-Engines Ranking*. `https://db-engines.com/en/ranking`. Accessed: 2022-02-07.

[13] DBtune. *DBtune is an AI-powered database tuning service*. `https://www.dbtune.ai/`. Accessed: 2022-06-23. 2022.

[14] scikit-learn developers. *sklearn.feature_selection.mutual_info_regression*. https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.mutual_info_regression.html?highlight=sklearn%20feature_selection%20mutual_info_regression. Accessed: 2022-05-18. scikit-learn.

[15] Djellel Eddine Difallah et al. "OLTP-Bench : an extensible testbed for benchmarking relational databases." In: *Proceedings of the VLDB Endowment* 7.4 (2013), pp. 277–288. ISSN: 2150-8097.

[16] Osama Eldawebi. "Efficient optimization of databases using parameter importance methods". In: 2022.

[17] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231.

[18] Yifan Feng et al. "An improved X-means and isolation forest based methodology for network traffic anomaly detection." In: *PLoS ONE* 17.1 (2022), pp. 1–18. ISSN: 19326203. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=a9h&AN=154969898&site=eds-live&scope=site.

[19] Evelyn Fix and J. L. Hodges. "Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties". In: *International Statistical Review / Revue Internationale de Statistique* 57.3 (1989), pp. 238–247. ISSN: 03067734, 17515823. URL: http://www.jstor.org/stable/1403797 (visited on 06/07/2022).

[20] Konrad Fraczek and Malgorzata Plechawska-Wojcik. "Comparative Analysis of Relational and Non-relational Databases in the Context of Performance in Web Applications". In: Apr. 2017, pp. 153–164. ISBN: 978-3-319-58273-3. DOI: 10.1007/978-3-319-58274-0_13.

[21] Peter I. Frazier. *A Tutorial on Bayesian Optimization*. 2018. DOI: 10.48550/ARXIV.1807.02811. URL: https://arxiv.org/abs/1807.02811.

[22] The PostgreSQL Global Development Group. *What is PostgreSQL*. https://www.postgresql.org/about. Accessed: 2022-03-28.

[23] *Heavy statistics collector overhead*. https://www.oreilly.com/library/view/postgresql-10-high/9781788474481/c6ab4e4b-b44e-4cf4-90ab-04854f6b3e3f.xhtml.

[24] *IONICE(1)*. Linux man page: man ionice, Accessed: 2022-06-03. June 2011.

[25] A. Jhingran, S. Padmanabhan, and A. Shatdal. "Join query optimization in parallel database systems." In: *Proceedings 1993 IEEE Workshop on Advances in Parallel and Distributed Systems, Advances in Parallel and Distributed Systems, 1993., Proceedings of the IEEE Workshop on* (1993), pp. 114–119. ISSN: 0-8186-5250-0. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=

true&AuthType=ip,uid&db=edseee&AN=edseee.588851&site=eds-live&scope=site.

[26] D.R Jones, M. Schonlau, and W.J Welch. "Efficient Global Optimization of Expensive Black-Box Functions". In: *Journal of Global Optimization* 13 (1998), pp. 455–492. DOI: https://doi.org/10.1023/A:1008306431147.

[27] Moppel Kaarel and et. al. *pgwatch2*. https://github.com/cybertec-postgresql/pgwatch2. Accessed: 2022-05-25. Cybertech.

[28] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. "Estimating mutual information". In: *Phys. Rev. E* 69 (6 June 2004), p. 066138. DOI: 10.1103/PhysRevE.69.066138. URL: https://link.aps.org/doi/10.1103/PhysRevE.69.066138.

[29] Alexey Lesovsky. *pgSCV - PostgreSQL ecosystem metrics collector*. https://github.com/lesovsky/pgscv. Accessed: 2022-05-25.

[30] Guoliang Li et al. "QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning". In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2118–2130. ISSN: 2150-8097. DOI: 10.14778/3352063.3352129. URL: https://doi.org/10.14778/3352063.3352129.

[31] D.R. Llanos and B. Palop. "TPCC-UVa: an open-source TPC-C implementation for parallel and distributed systems." In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, Parallel and Distributed Processing* (2006). ISSN: 1-4244-0054-6.

[32] mishima340. *PostgreSQL fails to execute query 15 under CHBENCHMARK workload #299*. https://github.com/oltpbenchmark/oltpbench/issues/299. Accessed: 2022-05-04. 2019.

[33] L. Nardi et al. "HyperMapper: a Practical Design Space Exploration Framework." In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2019 IEEE 27th International Symposium on, MASCOTS* (2019), pp. 425–426. ISSN: 978-1-7281-4950-9.

[34] Luigi Nardi and et al. *Hypermapper*. https://github.com/luinardi/hypermapper. Accessed: 2022-04-8.

[35] Luigi Nardi and et al. *Optimization Methods*. https://github.com/luinardi/hypermapper/wiki/Optimization-Methods. Accessed: 2022-04-8.

[36] *numpy.linalg.lstsq*. https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html.

[37] *numpy.random.normal*. https://numpy.org/doc/stable/reference/random/generated/numpy.random.normal.html.

[38] Cédric St-Onge et al. "Detection of time series patterns and periodicity of cloud computing workloads." In: *Future Generation Computer Systems* 109 (2020), pp. 249–261. ISSN: 0167-739X. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=edselp&AN=S0167739X19312440&site=eds-live&scope=site.

[39] pganalyze. *FAQ*. https://pganalyze.com/faq. Accessed: 2022-06-23.

[40] *pgmetrics*. https://pgmetrics.io/. Accessed: 2022-05-25. pgmetrics.

[41] PostgreSQL. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. https://www.postgresql.org/. Accessed: 2022-02-15.

[42] The GNOME Project. *What is swap memory?* https://help.gnome.org/users/gnome-system-monitor/stable/mem-swap.html.en. Accessed: 2022-03-23.

[43] Francois Raab and et al. *TPC-C Overview*. http://www.tpc.org/tpcc/detail5.asp. Accessed: 2022-02-15.

[44] *RENICE(1)*. Linux man page: man renice, Accessed: 2022-06-03. June 2014.

[45] Giampaolo Rodola Revision. *psutil documentation*. https://psutil.readthedocs.io/en/latest/. Accessed: 2022-03-22.

[46] Bobak Shahriari et al. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.

[47] TECMINT. *How to Clear Cache in Linux?* https://www.tecmint.com/clear-ram-memory-cache-buffer-and-swap-space-on-linux/. Accessed: 2022-04-5. 2021.

[48] *The Statistics Collector*. https://www.postgresql.org/docs/9.6/monitoring-stats.html.

[49] COVER THOMAS M and THOMAS JOY A. *ELEMENTS OF INFORMATION THEORY*. 2nd ed. WILEY-INTERSCIENCE, 2006.

[50] *threading — Thread-based parallelism*. https://docs.python.org/3/library/threading.html.

[51] *TPC BENCHMARK™C*. Standard Specification, Revision 5.11. Presidio of San Francisco, Building 572B Ruger St. (surface), 2010.

[52] *TPC BENCHMARK™H*. (Decision Support), Standard Specification, Revision 3.0.0. Presidio of San Francisco, Building 572B Ruger St. (surface), 2021.

[53] Olsson Viktor. *ConfigPlotter*. git@github.com:viktor-51/ConfigPlotter.git. June 2022.

[54] LUO YUAN et al. "Deep Learning-based Anomaly Detection in Cyber-physical Systems: Progress and Opportunities." In: *ACM Computing Surveys* 54.5 (2021), pp. 1–36. ISSN: 03600300. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=bth&AN=151371533&site=eds-live&scope=site.

[55] Ji Zhang et al. "An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 415–432. ISBN: 9781450356435. DOI: 10.1145/3299869.3300085. URL: https://doi.org/10.1145/3299869.3300085.

[56] Niu Zhixian et al. "Auto-recognizing DBMS Workload Based on C5.0 Algorithm." In: *2009 Second International Workshop on Knowledge Discovery and Data Mining, Knowledge Discovery and Data Mining, 2009. WKDD 2009. Second International Workshop on* (2009), pp. 777–780. ISSN: 978-0-7695-3543-2. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=edseee&AN=edseee.4772051&site=eds-live&scope=site.

6 Appendix

6.1 Knobs search space

6.1.1 Knobs with priors

Table 25: *Table of knobs in search space with values and the prior probabilities. Where (x) refers to the unit x being used.*

shared_buffers (MB)		work_mem (kB)		random_page_cost	
Value	Prior	Value	Prior	Value	Prior
640	1.3%	4096	6.08%	0.1	7.94%
768	1.4%	5120	6.14%	0.5	8.24%
896	1.4 %	6144	6.19%	1	8.38%
1024	2.9%	6990	6.19%	1.1	8.38%
1280	3%	7168	6.14%	1.5	8.24%
1536	3.1%	8192	6.04%	2	7.95%
1792	3.2%	9216	5.89%	2.5	7.54 %
2048	6.8%	10240	5.7%	3	7.02%
2560	7.1%	11264	5.47%	3.5	6.42%
3072	7.3%	12288	5.21%	4	5.77%
3584	7.4%	13312	4.92%	4.5	5.1%
4096	7.4%	14336	4.6%	5	4.42%
4608	7.3%	15360	4.27%	5.5	3.77%
5120	7.1%	17408	3.593%	6	3.16 %
5632	6.8%	19456	3.59%	6.5	2.6 %
6144	6.4%	21504	3.25%	7	2.1 %
6656	5.8%	23552	2.92%	7.5	1.67 %
7168	5.3%	25600	2.6%	8	1.3%
7680	4.8%	27648	2.3%		
8192	4.2%	29696	2.01%		
		30720	1.75%		
		35840	1.51%		
		40960	1.29%		
		46080	1.09%		
		51200	0.92%		

Table 26: *Values and priors for the effective_io_concurrency knob*

effective_io_concurrency	
Value	Prior
1	10%
100	20%
200	30%
300	20%
400	20%

6.1.2 Knobs without priors

Table 27: *Values used in BO for knobs. Where (x) refers to the unit x being used.*

max_wal_size (GB)	max_parallel_workers_per_gather	max_parallel_workers
4	1	4
8	2	8
16	4	
20	8	
24	16	
32		
64		

Table 28: *Values used in BO for knobs. Where (x) refers to the unit x being used.*

max_worker_processes	checkpoint_timeout (min)	checkpoint_completion_target
4	5	0.9
10	10	

6.2 CH-benCHmark

Table 29: *Table that shows the difference between the original CH-configuration and the new CH-configuration which is used for this project.*

Query number	Original CH	New CH
1	5	5
2	5	5
3	5	5
4	4	5
5	5	5
6	4	5
7	5	5
8	4	4
9	5	5
10	4	5
11	5	5
12	4	4
13	5	5
14	4	4
15	5	0
16	4	5
17	5	5
18	4	5
19	5	5
20	4	4
21	5	5
22	4	4

6.3 Hardware metrics

Table 30: *The table shows all the metrics collected for hardware utilization statistics under the collections module psutil[45]. In more detail the table shows: which function was used to collect the data, what it collects, when it is restored and if system dependent (SD). If the metric is a PostgreSQL isolated metric then a (P) is shown in the function name. Headers with only a name is the collection category. – Headers – surrounded by "–" is showing that an expansion of all return parameters will be represented below. The function which is expanded is placed directly bellow the – Header –. The shorthand func. means function. The units are usually these three; milliseconds (ms) if time is involved, bytes (b) if storage or memory and lastly a count if its a numbering of something. The metric units that diverge from this will be reported individually in the table.*

function	collects	restored	SD (T/F)
CPU			
cpu_percent	CPU utilization percentage	cycle	F
num_fds (P)	opened process file descriptors	never	T
– CPU getloadavg func. contains –			
getloadavg	processes in a runnable state	(1, 5, 15) min	F
1min	processes in an runnable state	1/60 hz	F
5min	processes in an runnable state	1/300 hz	F
15min	processes in an runnable state	1/900 hz	F
– CPU freq func. contains –			
cpu_freq	CPU frequency in Mhz	cycle	T
current	CPU frequency in Mhz	cycle	T
– CPU times func. contains –			
cpu_times	CPU processing times (ms)	cycle	T
idle	time idling (ms)	cycle	F
steal	OS time, virtual environment (ms)	cycle	T
guest	time, virtual CPU for guest (ms)	cycle	T
guest_nice	same as guest but niced (ms)	cycle	T
nice	process time, niced (ms)	cycle	T
irq	hardware interrupt (ms)	cycle	T
softirq	software interrupt (ms)	cycle	T
– CPU stats func. contains –			
cpu_stats	CPU context switches	cycle	T
interrupts	interrupts since boot	cycle	F
soft_interrupts	software interrupts	cycle	T
– CPU times (P) func. contains –			
cpu_times (P)	CPU processing times (ms)	cycle	T
user (P)	time in user mode (ms)	cycle	F
system (P)	time in kernel mode (ms)	cycle	F
iowait (P)	waiting for I/O (ms)	cycle	T
– CTX Switches (P) func. contains –			
num_ctx_switches (P)	context switches (ctx)	cycle	F
voluntary (P)	number of voluntary ctx	cycle	F
continuing on next page			

function	collects	restored	SD (T/F)
involuntary (P)	number of involuntary ctx	cycle	F
virtual memory			
– virtual memory func. contains –			
virtual_memory	memory related	never	T
buffers	cache file system and metadata (b)	never	T
cached	cache for unknown things (b)	never	T
shared	mutli-access memory (b)	never	T
slab	kernel cache (b)	never	T
– virtual memory (P) func. contains –			
memory_info	memory related (P)	never	T
rss (P)	non-swapped physical memory (b)	never	F
vms (P)	virtual memory, total used (b)	never	F
shared (P)	shareable memory (b)	never	T
text (P)	memory for code (b)	never	T
data (P)	non text memory (b)	never	T
lib (P)	memory in shared libraries (b)	never	T
dirty (P)	number of dirty pages	never	T
memory_percent (P)	in rss, *total/used*	never	T
swap memory			
– swap memory func. contains –			
swap_memory	swap memory related	flushed	T
used	used swap memory (b)	iteration	F
free	free swap memory (b)	iteration	F
percent	percentage of available memory	iteration	F
sin	swapped in from disk (b)	cycle	T
sout	swapped out from disk (b)	cycle	T
disk			
– disk func contains –			
disk_io_counters	disk data	cycle	T
read_time	time reading from disk (ms)	cycle	T
write_time	time writing to disk (ms)	cycle	T
busy_time	time doing I/O:s (ms)	cycle	T
read_merged_count	number of merged reads	cycle	T
write_merged_count	number of merged writes	cycle	T
– disk (P) func contains –			
io_counters (P)	disk data	cycle	T
read_count	number of reads (b)	cycle	T
write_count	number of writes (b)	cycle	T
read_bytes	number of bytes read	cycle	T
write_bytes	number of bytes written	cycle	T
read_chars	number of bytes passed, read	cycle	T
write_chars	number of bytes passed, write	cycle	T
network			

continuing on next page

function	collects	restored	SD (T/F)
– network func. contains –			
net_io_counters	network data	cycle	F
bytes_recv	received (b)	cycle	F
packets_recv	packets received	cycle	F
errin	errors while receiving	cycle	F
errout	errors while sending	cycle	F
dropin	incoming packets dropped	cycle	F
finished			

6.4 Multi-workload Optimization

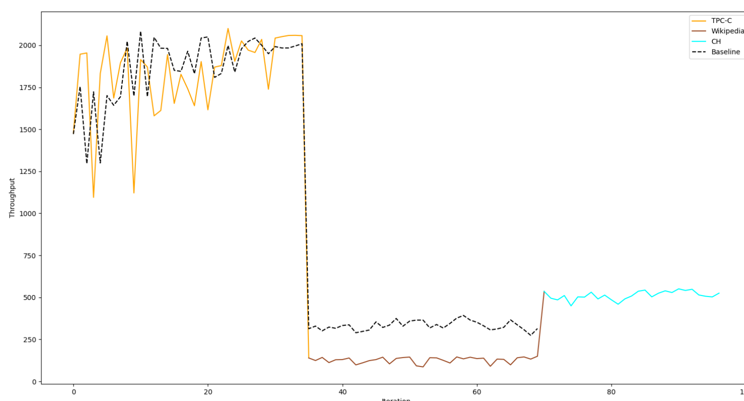

Figure 19: *Data for the multi-workload system that includes the unreliable CH-benCHmark*

6.5 Hardware analysis

Table 31: *Shows TPC-C versus Wikipedia. The table shows: (1) The mean distance between the workloads. (2) The point by point distance STD. (3) The iteration based STD. (4) How much the workloads intermingle i.e intersects. What each category, like high and low, represents and how they were generated is described in section 3.3.2.*

metric	distance	inner STD	outer STD	intermingling
- CPU -				
cpu_percent	low	high	low	very high
1min	low	very high	medium	very high
continuing on next page				

metric	distance	inner STD	outer STD	intermingling
5min	low	very high	medium	very high
15min	low	very high	medium	very high
current	low	very high	low	very high
idle	high	medium	low	non
softirq	low	very high	low	very high
steal	low	very high	low	very high
user (P)	medium	medium	low	high
system (P)	low	very high	low	very high
iowait (P)	high	medium	low	low
interrupts	high	medium	low	very high
soft_interupts	low	very high	low	very high
voluntary (P)	low	very high	low	very high
involuntary (P)	high	medium	low	non
num_fds (P)	low	low	low	non
- Virtual Memory -				
buffers	medium	medium	medium	non
cached	high	low	low	non
shared	low	very high	very high	very high
slab	high	low	low	non
rss (P)	medium	low	low	non
vms (P)	medium	low	low	non
shared (P)	medium	low	low	non
text (P)	medium	low	low	non
data (P)	low	high	medium	high
- Disk -				
read_time	high	medium	low	medium
write_time	high	high	medium	very high
read_merged_count	low	very high	high	very high
write_merged_count	high	very high	low	very high
busy_time	high	medium	low	non
read_count (P)	high	medium	low	non
write_count (P)	low	very high	medium	very high
read_bytes (P)	high	medium	low	medium
write_bytes (P)	medium	medium	high	high
read_chars (P)	high	medium	low	non
write_chars (P)	medium	medium	medium	very high
- Net -				
bytes_recv	medium	high	medium	very high
packets_recv	low	very high	medium	very high
finished				

Table 32: *Shows TPC-C versus CH-benCHmark. The table shows: (1) The mean distance between the workloads. (2) The point by point distance STD. (3) The iteration based STD. (4) How much the workloads intermingle i.e intersects. What each category, like high and low, represents and how they were generated is described in section 3.3.2.*

metric	distance	inner STD	outer STD	intermingling
- CPU -				
cpu_percent	high	low	low	non
1min	high	medium	low	low
5min	high	medium	low	non
15min	high	medium	low	non
current	low	very high	low	very high
idle	high	low	low	non
softirq	high	medium	low	low
steal	low	very high	low	very high
user (P)	high	low	low	non
system (P)	low	high	low	very high
iowait (P)	low	very high	low	very high
interrupts	low	very high	low	very high
soft_interupts	high	medium	low	non
voluntary (P)	medium	high	low	high
involuntary (P)	high	medium	low	medium
num_fds (P)	low	very high	medium	very high
- Virtual Memory -				
buffers	medium	medium	medium	medium
cached	low	high	medium	very high
shared	low	medium	low	non
slab	low	medium	medium	non
rss (P)	low	medium	low	non
vms (P)	low	low	low	non
shared (P)	medium	low	low	non
text (P)	medium	0	low	non
data (P)	high	low	low	non
- Disk -				
read_time	medium	very high	medium	very high
write_time	medium	very high	low	very high
read_merged_count	high	very high	high	very high
write_merged_count	high	very high	medium	very high
busy_time	low	very high	medium	very high
read_count (P)	high	medium	low	non
write_count (P)	low	very high	low	very high
read_bytes (P)	high	very high	low	very high
write_bytes (P)	low	very high	low	very high

continuing on next page

metric	distance	inner STD	outer STD	intermingling
read_chars (P)	high	medium	low	non
write_chars (P)	low	very high	low	very high
- Net -				
bytes_recv	high	medium	low	non
packets_recv	high	medium	low	non
finished				

Table 33: *Shows Wikipedia versus CH-benCHmark. The table shows: (1) The mean distance between the workloads. (2) The point by point distance STD. (3) The iteration based STD. (4) How much the workloads intermingle i.e intersects. What each category, like high and low, represents and how they were generated is described in section 3.3.2.*

metric	distance	inner STD	outer STD	intermingling
- CPU -				
cpu_percent	medium	medium	low	non
1min	high	medium	low	non
load_5min	high	medium	low	non
15min	high	medium	low	non
current	low	very high	medium	high
idle	high	very high	medium	high
softirq	medium	very high	medium	high
steal	low	very high	low	high
user (P)	medium	medium	low	non
system (P)	low	high	low	high
iowait (P)	high	very high	medium	high
interrupts	high	high	low	high
soft_interupts	high	high	low	high
voluntary (P)	low	very high	low	high
involuntary (P)	high	medium	low	high
num_fds (P)	low	high	medium	high
- Virtual Memory -				
buffers	high	medium	medium	non
cached	high	low	low	non
shared	low	very high	very high	high
slab	high	low	medium	non
rss (P)	low	very high	low	non
vms (P)	low	low	low	non
shared (P)	low	very high	low	high
text (P)	low	very high	very high	high
data (P)	high	low	low	non
- Disk -				
continuing on next page				

metric	distance	inner STD	outer STD	intermingling
read_time	high	very high	medium	high
write_time	medium	very high	low	high
read_merged_count	high	very high	high	high
write_merged_count	high	very high	medium	high
busy_time	high	medium	low	high
read_count (P)	high	medium	low	high
write_count (P)	low	very high	medium	high
read_bytes (P)	high	very high	low	high
write_bytes (P)	low	very high	high	high
read_chars (P)	high	medium	low	high
write_chars (P)	low	very high	medium	high
- Net -				
bytes_recv	high	medium	low	high
packets_recv	high	high	medium	high
finished				

Table 34: *Shows individual TPC-C statistics. The table shows: (1) The mean distance between TPC-C's non-optimized data and optimized data. (2) The categorized mean-STD of TPC's non-optimized data. (3) If the non-optimization is biased towards the starting iteration of the workload. (4) If the distance between the non-optimized and the optimized data trend in any direction. (5) The variance of the trend. What each category, like high and low represents, and how the categories were generated is described in section 3.3.2.*

metric	distance	STD	bias	growth	growth variance
- CPU -					
cpu_percent	medium	low	low	+medium	low
1min	high	medium	high	+very high	low
5min	high	low	high	+very high	low
15min	high	low	high	+very high	low
current	low	low	low	-low	low
idle	high	low	low	+medium	low
softirq	high	low	low	+medium	low
steal	low	very high	low	-low	low
user (P)	high	low	low	+medium	low
system (P)	low	low	low	+low	medium
iowait (P)	low	medium	low	+low	high
interrupts	low	medium	low	-low	low

continuing on next page

metric	distance	STD	bias	growth	growth variance
soft_interupts	high	low	low	+medium	low
voluntary (P)	high	low	low	+medium	low
involuntary (P)	low	low	low	+low	medium
num_fds (P)	low	low	low	+low	low
- Virtual Memory -					
buffers	high	low	low	+medium	low
cached	low	low	low	+high	low
shared	high	low	low	+high	low
slab	low	low	low	+high	low
rss (P)	high	low	low	-low	low
vms (P)	high	low	low	+high	low
shared (P)	high	low	low	-low	low
text (P)	low	low	low	-low	low
data (P)	low	low	low	+low	medium
- Disk -					
read_time	high	medium	low	-low	low
write_time	high	medium	low	+medium	low
read_merged_count	high	very high	low	+very high	medium
write_merged_count	high	high	low	-low	low
busy_time	low	low	low	+high	medium
read_count (P)	high	low	low	+low	medium
write_count (P)	medium	low	low	+low	medium
read_bytes (P)	high	low	low	+low	medium
write_bytes (P)	low	low	low	+medium	low
read_chars (P)	high	low	low	+low	medium
write_chars (P)	medium	low	low	+medium	low
- Net -					
bytes_recv	high	low	low	+medium	low
packets_recv	high	low	low	+medium	low
finished					

Table 35: *Shows individual Wikipedia statistics. The table shows: (1) The mean distance between Wikipedia's non-optimized data and optimized data. (2) The categorized mean-STD of Wikipedia's non-optimized data. (3) If the non-optimization is biased towards the starting iteration of the workload. (4) If the distance between the non-optimized and the optimized data trend in any direction. (5) The variance of the trend. What each category, like high and low represents, and how the categories were generated is described in section 3.3.2.*

metric	distance	STD	bias	growth	growth variance
- CPU -					
1min	high	high	low	+low	very high
5min	high	high	medium	+low	very high
15min	high	medium	high	+low	medium
idle	high	very high	low	-low	low
softirq	medium	medium	low	-low	low
steal	high	very high	low	+low	very high
user (P)	low	low	low	+low	very high
system (P)	high	low	low	-low	low
iowait (P)	high	very high	low	-low	low
interrupts	high	high	low	-low	low
soft_interupts	low	medium	low	+low	very high
voluntary (P)	low	medium	low	+low	very high
involuntary (P)	high	medium	low	-low	low
num_fds (P)	low	low	high	-low	low
cpu_frequency	low	low	low	+low	medium
cpu_percent	medium	low	low	-low	low
- Virtual Memory -					
buffers	high	medium	low	-low	low
cached	medium	medium	low	-low	low
shared	high	low	low	-low	low
slab	low	low	low	-low	low
rss (P)	high	low	low	-low	low
vms (P)	high	low	low	-low	low
shared (P)	high	low	low	-low	low
text (P)	low	low	low	-low	low
data (P)	low	low	low	-low	low
- Disk -					
read_time	medium	very high	low	-low	low
write_time	high	very high	low	-low	low
read_merged_count	low	very high	low	-low	low
write_merged_count	high	very high	low	-low	low
busy_time	low	high	low	-low	low
read_count (P)	high	low	low	-low	low

continuing on next page

metric	distance	STD	bias	growth	growth variance
write_count (P)	high	medium	low	-low	low
read_bytes (P)	high	very high	low	-low	low
write_bytes (P)	high	high	low	-low	low
read_chars (P)	high	low	low	-low	low
write_chars (P)	high	medium	low	-low	low
- Net -					
bytes_recv	low	medium	low	-low	low
packets_recv	low	medium	low	-low	low
finished					

Table 36: *Shows individual CH-benCHmark statistics. The table shows: (1) The mean distance between CH-benCHmark's non-optimized data and optimized data. (2) The categorized mean-STD of CH-benCHmark's non-optimized data. (3) If the non-optimization is biased towards the starting iteration of the workload. (4) If the distance between the non-optimized and the optimized data trend in any direction. (5) The variance of the trend. What each category, like high and low represents, and how the categories were generated is described in section 3.3.2.*

metric	distance	STD	bias	growth	growth variance
- CPU -					
1min	low	low	low	-low	low
5min	low	low	low	-low	low
15min	low	low	low	+medium	high
idle	low	medium	low	+medium	low
softirq	low	medium	low	+low	medium
steal	low	very high	low	+low	medium
user (P)	low	low	low	-low	low
system (P)	low	low	low	-low	low
iowait (P)	high	high	low	-low	low
interrupts	low	medium	low	+low	high
soft_interupts	low	low	low	+low	low
voluntary (P)	low	low	low	+medium	medium
involuntary (P)	low	low	low	+low	medium
num_fds (P)	low	low	medium	-low	low
cpu_frequency	low	low	low	+low	medium
cpu_percent	low	low	low	-low	low
- Virtual Memory -					
buffers	low	medium	medium	+low	medium
cached	low	low	low	+high	medium
shared	high	low	low	+medium	medium
continuing on next page					

metric	distance	STD	bias	growth	growth variance
slab	low	low	low	+medium	medium
rss (P)	high	low	low	+low	medium
vms (P)	high	low	medium	+medium	medium
shared (P)	high	low	low	+low	medium
text (P)	low	low	low	-low	low
data (P)	low	low	high	-low	low
- Disk -					
read_time	medium	medium	low	-low	low
write_time	high	medium	low	+low	medium
read_merged_count	high	very high	low	+low	very high
write_merged_count	low	very high	low	+low	very high
busy_time	low	low	low	+medium	medium
read_count (P)	low	low	low	-low	low
write_count (P)	high	low	low	+low	medium
read_bytes (P)	low	medium	low	-low	low
write_bytes (P)	medium	low	low	+low	medium
read_chars (P)	low	low	low	-low	low
write_chars (P)	high	low	low	+low	medium
- Net -					
bytes_recv	low	low	low	+low	low
packets_recv	low	low	low	+low	low
finished					

Table 37: *This table include the metrics not represented in any other tables except the collections table 30. The table displays the metric name and information why its not included in any other table.*

metric	info
- CPU -	
nice	one single spike in TPC-C and Wiki
guest	always zero
guest_nice	always zero
irq	always zero
- Virtual Memory -	
lib (P)	always zero
dirty (P)	always zero
memory_percent (P)	includes the exact same info as rss (P)
- Swap Memory -	
used	always zero
free	always zero
percent	always zero
sin	always zero
sout	always zero
- Disk -	
errin	always zero
errout	always zero
dropin	always zero

6.6 Hardware Detection

Table 38: *Shows a general table of the metrics categorization for Euclidean, CUSUM-chart and MI workload detection on the negative test for non-optimized data, see a more detailed description of the categories in section 3.5.2. In the test, the hyperparametrs were set as: (1) 5.018 for the Euclidean distance hyperparameter. (2) 6.83 and 0.95 for distance and the percentage of allowed outliers for CUSUM-chart. (3) for MI the difference hyperparameter was set as 0. The columns in the table show how the metrics performed for each workload when doing self-detection.*

metric	tpcc	wikipedia	CH-benCHmark
- CPU -			
current	good	good	good
cpu_percent	good	good	good
involuntary (P)	good	good	good
voluntary (P)	good	good	good
idle	good	good	good
interrupts	good	good	good

continuing on next page

metric	tpcc	wikipedia	CH-benCHmark
iowait (P)	good	good	good
irq	good	good	good
soft_interupts	good	good	good
softirq	good	good	good
steal	good	good	good
- Virtual Memory -			
buffers	good	good	good
busy_time	good	good	good
cached	good	good	good
rss (P)	good	good	good
shared	good	good	good
slab	good	good	good
system (P)	good	good	good
text (P)	good	good	good
user (P)	good	good	good
- Disk -			
read_bytes (P)	good	good	good
read_chars (P)	good	good	good
read_count (P)	good	good	good
read_merged_count	good	good	good
read_time	good	good	good
write_bytes (P)	good	good	good
write_chars (P)	good	good	good
write_count (P)	good	good	good
write_merged_count	good	good	good
write_time	good	good	good
- Net -			
bytes_recv	good	good	good
packets_recv	good	good	good
finished			

Table 39: *Shows a general table of the metrics categorization for Euclidean, CUSUM-chart and MI workload detection on the positive test for non-optimized data, see a more detailed description of the categories in section 3.5.2. In the test, the hyperparametrs were set as: (1) 5.018 for the Euclidean distance hyperparameter. (2) 6.83 and 0.95 for distance and the percentage of allowed outliers for CUSUM-chart. (3) for MI the difference hyperparameter was set as 0. The columns in the table show how the metrics performed for each workload when doing self-detection.*

metric	tpcc vs wikipedia	tpcc vs CH-benCHmark	wikipedia vs CH-benCHmark
- CPU -			
softirq	bad	bad	bad
soft_interupts	bad	bad	bad
steal	bad	bad	bad
current	bad	bad	bad
voluntary (P)	bad	bad	bad
- Virtual Memory -			
system (P)	bad	bad	bad
- Disk -			
write_bytes (P)	bad	bad	bad
write_chars (P)	bad	bad	bad
write_count (P)	bad	bad	bad
write_merged_count	bad	bad	bad
write_time	bad	bad	bad
- Net -			
packets_recv	bad	bad	bad

Table 40: *Shows the metrics categorization for Euclidean workload detection on the positive test for non-optimized data, see a more detailed description of the categories in section 3.5.2. In the test, the Euclidean distance hyperparameter was set to 5.018. The columns in the table show how the metrics performed for each workload versus another workload during the detection.*

metric	tpcc vs wikipedia	tpcc vs CH-benCHmark	wikipedia vs CH-benCHmark
- CPU -			
cpu_percent	bad	bad	bad
involuntary (P)	+low	bad	bad
idle	+high	+high	bad
interrupts	-high	bad	bad
iowait (P)	-mid	bad	bad
1min	bad	-high	-low
5min	bad	-mid	-low
15min	bad	bad	-high
num_fds (P)	bad	bad	bad
- Virtual Memory -			
data (P)	bad	+high	+high
buffers	bad	bad	-high
busy_time	+low	bad	-low
rss (P)	bad	bad	bad
shared	bad	bad	bad
shared (P)	bad	bad	bad
slab	-mid	bad	bad
user (P)	bad	bad	bad
text (P)	bad	bad	bad
vms (P)	bad	bad	bad
cached	+high	bad	+high
- Disk -			
read_bytes (P)	-mid	bad	bad
read_chars (P)	bad	+high	bad
read_count (P)	bad	+high	bad
read_merged_count	bad	bad	bad
read_time	bad	bad	bad
- Net -			
bytes_recv	bad	bad	-high

Table 41: *Shows the metrics categorization for CUSUM-chart workload detection on the positive test for non-optimized data, see a more detailed description of the categories in section 3.5.2. In the test, the hyperparameters, i.e the distance and the allowed outliers were set to 6.83 and 0.95. The columns in the table show how the metrics performed for each workload versus another workload during the detection.*

metric	tpcc vs wikipedia	tpcc vs CH-benCHmark	wikipedia vs CH-benCHmark
- CPU -			
cpu_percent	bad	-high	bad
involuntary (P)	bad	bad	bad
idle	bad	bad	bad
interrupts	bad	bad	bad
iowait (P)	bad	bad	bad
num_fds (P)	+high	-low	-low
- Virtual Memory -			
data (P)	-low	+high	+high
buffers	bad	bad	-high
busy_time	+high	bad	bad
cached	+high	bad	+high
rss (P)	+high	+high	-high
shared	+high	bad	-mid
shared (P)	+high	+high	bad
slab	+low	bad	-low
text (P)	+high	+high	bad
user (P)	bad	+low	bad
vms (P)	+high	+high	+high
- Disk -			
read_bytes (P)	bad	bad	-low
read_chars (P)	-low	+high	bad
read_count (P)	-low	+high	bad
read_merged_count	bad	bad	bad
read_time	bad	bad	-low
- Net -			
bytes_recv	bad	bad	bad

Table 42: *Shows the pruned bad metrics, and their categorization when using MI for workload detection on the negative test when setting the hyperparameter "difference" to 0.5. See a more detailed description of the categories in section 3.5.2. The columns in the table show how the metrics performed for each workload when doing self-detection.*

metric	tpcc	wikipedia	CH-benCHmark
- CPU -			
current	bad	bad	bad
cpu_percent	bad	bad	bad
involuntary (P)	bad	bad	bad
voluntary (P)	bad	bad	bad
soft_interupts	bad	bad	bad
softirq	bad	bad	bad
steal	bad	bad	bad
idle	bad	bad	bad
interrupts	bad	bad	bad
iowait (P)	bad	bad	bad
1min	bad	bad	bad
5min	-high	good	good
15min	-low	good	good
num_fds (P)	bad	good	good
- Virtual Memory -			
buffers	good	good	-low
busy_time	bad	bad	bad
cached	bad	good	-high
slab	bad	good	good
system (P)	bad	bad	bad
text (P)	bad	bad	bad
user (P)	bad	bad	bad
- Disk -			
read_bytes (P)	bad	bad	good
read_chars (P)	bad	bad	bad
read_count (P)	bad	bad	bad
read_merged_count	bad	bad	bad
read_time	bad	bad	-high
write_bytes (P)	bad	bad	bad
write_chars (P)	bad	bad	bad
write_count (P)	bad	bad	-high
write_merged_count	bad	bad	bad
write_time	bad	bad	bad
- Net -			
bytes_recv	bad	bad	bad
packets_recv	bad	bad	bad

Table 43: *Shows the non-pruned metrics categorization for Euclidean workload detection on the positive test for optimized data, see a more detailed description of the categories in section 3.5.2. In the test, the hyperparameters, i.e the distance was set to 5.018. The columns in the table show how the metrics performed for each workload versus another workload during the detection.*

metric	tpcc vs wikipedia	tpcc vs CH-benCHmark	wikipedia vs CH-benCHmark
- CPU -			
cpu_percent	-high	bad	-mid
current	bad	bad	bad
steal	bad	bad	bad
involuntary (P)	bad	bad	+low
voluntary (P)	bad	bad	bad
idle	bad	bad	-mid
interrupts	bad	bad	-low
iowait (P)	bad	bad	-low
busy_time	+high	bad	-mid
user (P)	bad	bad	-high
soft_interupts	-mid	-low	bad
softirq	-mid	-low	bad
system (P)	-low	bad	-mid
- Virtual Memory -			
text (P)	+high	+high	bad
- Disk -			
read_merged_count	bad	bad	bad
read_time	-low	bad	+high
write_bytes (P)	-high	bad	-high
write_merged_count	bad	-low	-low
write_time	bad	bad	bad
- Net -			
packets_recv	-low	-low	bad
bytes_recv	bad	-low	bad

Table 44: *Shows the non-pruned metrics categorization for CUSUM-chart workload detection on the positive test for optimized data, see a more detailed description of the categories in section 3.5.2. In the test, the hyperparameters, i.e the distance and the allowed outliers were set to 6.83 and 0.95. The columns in the table show how the metrics performed for each workload versus another workload during the detection.*

metric	tpcc vs wikipedia	tpcc vs CH-benCHmark	wikipedia vs CH-benCHmark
- CPU -			
cpu_percent	bad	bad	-mid
current	bad	bad	bad
15min	bad	bad	bad
involuntary (P)	-mid	bad	+high
iowait (P)	+low	-high	bad
voluntary (P)	-high	bad	bad
soft_interupts	-low	+low	bad
softirq	+low	-low	bad
system (P)	+low	bad	+high
interrupts	-mid	bad	-high
- Virtual Memory -			
data (P)	bad	+high	+high
busy_time	-mid	bad	-mid
cached	-mid	bad	-high
slab	bad	bad	bad
text (P)	bad	bad	bad
user (P)	bad	bad	bad
- Disk -			
read_bytes (P)	+high	bad	+mid
read_merged_count	bad	bad	bad
read_time	+mid	bad	+low
write_bytes (P)	+low	bad	-mid
write_chars (P)	-low	bad	-high
write_merged_count	bad	bad	bad
- Net -			
bytes_recv	bad	+low	bad
packets_recv	-low	+low	bad

6.7 Code Listings

```
import os
import copy
import time
```

```
import psutil
import numpy as NP

from FileHandling import system_file_handler as file_handler
from FileHandling.CSVHandling import CSV_handler

class Measurer:
    SAVE_SUFFIXES = [
        "cpu_data.csv", "memory_data.csv", "swap_memory.csv",
        ↪ "disk_data.csv",
        "net_data.csv"
    ]

    CPU_HEADER = [
        "load_1min", "load_5min", "load_15min", "nice", "idle",
        ↪ "irq",
        "softirq", "steal", "guest", "guest_nice", "user(post)",
        ↪ "system(post)",
        "iowait(post)", "interrupts", "soft_interupts",
        "ctx_switches_vol(post)", "ctx_switches_invol(post)",
        ↪ "num_fds(post)",
        "cpu_frequency", "cpu_percent", "time(s)"
    ]
    MEMORY_HEADER = [
        "buffers", "cached", "shared", "slab", "rss(post)",
        ↪ "vms(post)",
        "shared(post)", "text(post)", "lib(post)", "data(post)",
        ↪ "dirty(post)",
        "mem_percent(post)", "time(s)"
    ]
    SWAP_HEADER = [
        "swap_used", "swap_free", "swap_percent", "sin", "sout",
        ↪ "time(s)"
    ]
    DISK_HEADER = [
        "read_time", "write_time", "read_merged_count",
        ↪ "write_merged_count",
        "busy_time", "read_count(post)", "write_count(post)",
        "read_bytes(post)", "write_bytes(post)",
        ↪ "read_chars(post)",
        "write_chars(post)", "time(s)"
    ]
    NET_HEADER = [
        "bytes_sent", "bytes_recv", "packets_sent",
        ↪ "packets_recv", "errin",
        "errout", "dropin", "dropout", "time(s)"
```

```
]

HEADERS = [CPU_HEADER, MEMORY_HEADER, SWAP_HEADER,
↪  DISK_HEADER, NET_HEADER]

POSTSQL_RESTORE_METRICS = [
    "num_ctx_switches", "cpu_times", "io_counters"
]
POSTSQL_METRICS = [
    "num_ctx_switches", "memory_info", "num_fds",
    ↪  "cpu_percent",
    "cpu_times", "io_counters", "memory_percent"
]

def __init__(
    self, config_handler, utils, semaphore_measure,
    ↪  semaphore_storage,
    run_storage, stop_event, blocking_event,
    ↪  workload_switch_event,
    discard_event
):
    self.semaphore_measure = semaphore_measure
    self.semaphore_storage = semaphore_storage
    self.run_storage = run_storage
    self.stop_event = stop_event
    self.blocking_event = blocking_event
    self.workload_switch_event = workload_switch_event
    self.discard_event = discard_event

    self.nbr_cpus = psutil.cpu_count()
    measure_info = config_handler.measure_info()

    self.nbr_benchmarks = measure_info["nbr_benchmarks"]
    self.workload_names =
    ↪  measure_info["oltpbench_benchmarks"]
    self.save_folder = measure_info["save_folder"]
    self.sleep_time = measure_info["sleep_time"]

    self.utils = utils

    self.workload_idx = -1
    self.iterations_per_workload = [0] * self.nbr_benchmarks

    try:
        os.mkdir(self.save_folder, mode = 777)
    except FileExistsError:
```

```
            print(
                "\n Save folder already exists: {}, continueing...
                ↪  \n"
                .format(self.save_folder)
            )


    def take_measurment(self):
        iteration_data = None

        while True:
            if self.should_stop(iteration_data): break

            with self.semaphore_storage:
                start_next_iteration, next_workload_idx =
                ↪  self.run_storage
                self.run_storage[0 : 2] = [False, -1]

            if start_next_iteration:
                if iteration_data:
                    self.write_to_files(iteration_data)

                iteration_data = [[], [], [], [], []]

                self.workload_idx = next_workload_idx
                self.iterations_per_workload[self.workload_idx]
                ↪  += 1

                time_taken_per_iteration = [0] *
                ↪  self.nbr_benchmarks

                ###Order: do not want clear operations counted
                ↪  towards cpu
                self.clear_swap_n_cache()
                self.workload_switch_event.set()

                self.setup_postgreSQL_measurement()

                start_time_next_measurement = time.perf_counter()
                iteration_dependent_metrics =
                ↪  self.initial_metrics()
                measurement_dependent_metrics =
                ↪  self.collect_metric_start_points()

            sleept_time = time.perf_counter() -
            ↪  start_time_next_measurement
```

```
            sleep_left = max(self.sleep_time - sleept_time, 0)

            time.sleep(sleep_left)
            temp = time.perf_counter()
            loop_time = temp  - start_time_next_measurement

            start_time_next_measurement = temp

            next_measurement = self.collect_metrics(
                measurement_dependent_metrics,
                 ↪  iteration_dependent_metrics
            )
            measurement_dependent_metrics =
            ↪  self.collect_metric_start_points()

            if self.handle_stop(iteration_data): break
            if self.not_blocked():
                time_taken_per_iteration[self.workload_idx] +=
                 ↪  loop_time
                measurment_duration =
                 ↪  time_taken_per_iteration[self.workload_idx]

                timestamped_data = self.append_timestamp(
                    next_measurement, measurment_duration
                )

                for idx, data_item in enumerate(iteration_data):
                    data_item.append(timestamped_data[idx])

            self.semaphore_measure.release()
            ### Give possibility for other thread to grab
            ↪  semaphore
            time.sleep(0.5)

        self.semaphore_measure.release()

    def clear_swap_n_cache(self):
        self.utils.clear_cache()
        self.utils.clear_swap()

    def initial_metrics(self):
        swap_memory_all = list(psutil.swap_memory())
        swap_iteration_dependent = swap_memory_all[1 : -2]

        return {"swap_iteration" : swap_iteration_dependent}
```

```
def setup_postgreSQL_measurement(self):
    self.postgre_processes = []

    setup_points = 3
    found_post_processes = {}

    processing_time = 0
    for idx in range(0, setup_points):
        sleep_left = max(0, self.sleep_time -
        ↪  processing_time)
        time.sleep(sleep_left)
        start_time = time.perf_counter()

        if idx >= setup_points - 1:
            for process in psutil.process_iter():
                try:
                    if process.name() == "postgres":
                        hash_id = hash(process)

                        if hash_id in found_post_processes:
                            found_post_processes[hash_id][0]
                            ↪  += 1
                        else:
                            found_post_processes[hash_id] =
                            ↪  [1, process]

                except psutil.Error as e:
                    print(f"An unexpected error occured:
                    ↪  {e}")
                    try:
                        del found_post_processes[hash_id]
                    except KeyError as ke:
                        print(
                            "tried to remove key which was
                            ↪  dead"
                            + ", but not in dict"
                            , ke
                        )

        processing_time = time.perf_counter() - start_time

    for hash_id, value_list in found_post_processes.items():
        if value_list[0] == 1:
            process = value_list[1]

            try:
```

```
                    children = process.children(recursive=True)

                    is_not_parent = True
                    for child in children:
                        try:
                            if hash(child) in
                            ↪  found_post_processes:
                                is_not_parent = False
                                break;

                        except psutil.Error as e:
                            print(f"An unexpected error occured:
                            ↪  {e}")


                except psutil.Error as e:
                    print(f"An unexpected error occured: {e}")
                    is_not_parent = False


                if is_not_parent:
                    self.postgre_processes.append(process)


    def collect_metric_start_points_postgresql(self):
        self.previous_metrics_post = []

        current_idx = 0
        for process in copy.copy(self.postgre_processes):
            try:
                self.previous_metrics_post.append(
                    process
                    .as_dict(attrs =
                    ↪  self.POSTSQL_RESTORE_METRICS)
                )
                current_idx += 1
            except (psutil.NoSuchProcess, psutil.ZombieProcess)
            ↪  as e:
                self.postgre_processes.pop(current_idx)
                print(
                    f"The psutil process with pid {e.pid} was not
                    ↪  reachable"
                    + " removing it"
                )
            except psutil.Error as e:
                print(f"An unexpected error occured: {e}")
```

```
def collect_metric_start_points(self):
    self.collect_metric_start_points_postgresql()

    cpu_times = list(psutil.cpu_times()[1 : ])
    cpu_times.pop(1)
    cpu_times.pop(2)
    cpu_stats = list(psutil.cpu_stats()[1 : 3])

    swap_memory_all = psutil.swap_memory()
    swap_reset = [swap_memory_all.sin, swap_memory_all.sout]

    disk_data = psutil.disk_io_counters()[4 :]
    net_data = psutil.net_io_counters()

    ###Trash value, used for initing psutil cpu measurment
    psutil.cpu_percent()

    return {
        "cpu_times" : cpu_times, "cpu_stats" : cpu_stats,
        "swap_reset" : swap_reset, "disk_data" : disk_data,
        "net_data" : net_data
    }

def dict_subtracter(self, source, target):
    #print("subbing")
    return self.dict_handler(source, target, NP.subtract)

def dict_adder(self, source, target):
    #print("adding")
    return self.dict_handler(source, target, NP.add)

def dict_handler(self, source, target, func):
    new_dict = {}

    for key in source:
        if key in target:
            new_dict[key] = func(source[key], target[key])
        else:
            new_dict[key] = source[key]

    return new_dict

def collect_metrics_postgres(self):
    first = True
```

```
        for idx, process in enumerate(self.postgre_processes):
            try:
                metrics = process.as_dict(attrs =
                ↪  self.POSTSQL_METRICS)
                prev_metrics = self.previous_metrics_post[idx]

                subtracted_dict = self.dict_subtracter(
                    metrics, prev_metrics
                )

                if first:
                    first = False
                    collected_metrics = subtracted_dict
                else:
                    collected_metrics = (
                        self.dict_adder(
                            collected_metrics, subtracted_dict
                        )
                    )

            except (psutil.NoSuchProcess, psutil.ZombieProcess)
            ↪  as e:
                print(
                    f"The psutil process with pid {e.pid} was not
                    ↪  reachable"
                )
            except psutil.Error as e:
                print(f"An unexpected error occured: {e}")

        return collected_metrics

    def collect_metrics(self, last_measurement,
    ↪  initial_measurement):
        postgres_metrics = self.collect_metrics_postgres()

        cpu_load_avg_np = (NP.array(psutil.getloadavg()) /
        ↪  self.nbr_cpus) * 100
        cpu_times_current = list(psutil.cpu_times()[1 : ])
        cpu_times_current.pop(1)
        cpu_times_current.pop(2)

        cpu_times = NP.subtract(
            cpu_times_current, last_measurement["cpu_times"]
        ).tolist()

        post_cpu_times = postgres_metrics["cpu_times"]
```

```
cpu_times.extend(post_cpu_times)

cpu_stats = NP.subtract(
    psutil.cpu_stats()[1 : 3],
    ↪  last_measurement["cpu_stats"]
).tolist()

cpu_stats.extend(postgres_metrics["num_ctx_switches"])
cpu_stats.append(postgres_metrics["num_fds"])

cpu_freq = psutil.cpu_freq().current
cpu_percentage = postgres_metrics["cpu_percent"]

cpu_data = list(cpu_load_avg_np)
cpu_data.extend(cpu_times)
cpu_data.extend(cpu_stats)
cpu_data.append(cpu_freq)
cpu_data.append(cpu_percentage)

memory_data = list(psutil.virtual_memory()[7 : ])
memory_data.extend(postgres_metrics["memory_info"])
memory_data.append(postgres_metrics["memory_percent"])

swap_memory_all = list(psutil.swap_memory())

swap_iteration_dependent = NP.subtract(
    swap_memory_all[1 : -2],
    ↪  initial_measurement["swap_iteration"]
)

swap_reset = NP.subtract(swap_memory_all[-2 : ],
↪  last_measurement["swap_reset"])

swap_memory = list(swap_iteration_dependent)
swap_memory.extend(swap_reset)

disk_data = NP.subtract(
    psutil.disk_io_counters()[4 :],
    ↪  last_measurement["disk_data"]
).tolist()
disk_data.extend(postgres_metrics["io_counters"])

net_data = NP.subtract(
    psutil.net_io_counters(),
    ↪  last_measurement["net_data"]
).tolist()
```

```
        computer_data = [cpu_data, memory_data, swap_memory,
        ↪  disk_data, net_data]

        return computer_data

    def not_blocked(self):
        return not self.blocking_event.is_set()

    def should_stop(self, iteration_data):
        if not self.semaphore_measure.acquire(blocking = False):
            while(not self.semaphore_measure.acquire(timeout =
            ↪  self.sleep_time)):
                if self.handle_stop(iteration_data):
                    return True

        if self.handle_stop(iteration_data):
            return True
        else:
            return False

    def handle_stop(self, iteration_data):
        if self.stop_event.is_set():
            self.write_to_files(iteration_data)
            return True
        else:
            return False

    def generate_save_paths(self, workload_idx, iteration_idx):
        workload_string = self.workload_names[workload_idx]
        iteration_string = "iteration_" + str(iteration_idx)

        path_objects_dir = [self.save_folder, workload_string,
        ↪  iteration_string]
        dir_path = "/".join(path_objects_dir)

        save_paths = []
        for save_suffix in self.SAVE_SUFFIXES:
            next_save_path = dir_path + "/" + save_suffix

            save_paths.append(next_save_path)

        return dir_path, save_paths

    def genereate_CSV_files(self, save_paths):
        for idx, save_path in enumerate(save_paths):
```

```
            next_header = self.HEADERS[idx]
            CSV_handler.create_file(save_path, next_header)

    def append_timestamp(self, measure_data,
    ↪   measurment_duration):
            data_with_time = []

            for data_item in measure_data:
                data_item = list(data_item)
                data_item.append(measurment_duration)

                data_with_time.append(data_item)

            return data_with_time

    def write_to_files(self, measure_data):
        if self.discard_event.is_set():
            self.discard_event.clear()
            self.iterations_per_workload[self.workload_idx] -= 1

            return

        save_dir, save_paths = self.generate_save_paths(
            self.workload_idx,
            ↪   self.iterations_per_workload[self.workload_idx]
        )

        file_handler.make_dir(save_dir)
        self.genereate_CSV_files(save_paths)

        for idx, data_item in enumerate(measure_data):
            next_save_path = save_paths[idx]
            CSV_handler.append(next_save_path, data_item)
```

Listing 1: *The hardware utilization statistics collections module*

6.8 Author Contribution

Table 45: *The table shows author contributions. X means that the author was participatory, H means that a supporting role was taken, O means that no help was done and C means implementation of code but no text writing.*

section	Jonas	Viktor
Thesis Formatting	H	X

continuing on next page

section	Jonas	Viktor
Correction reading	X	H
Appendix	X	X
Abstract	X	H
Preface	X	X
Introduction		
start	X	O
Research Questions	X	X
Background & Related Work		
Background	X	O
Terms	X	X
Regarding Objectives	X	O
Bayesian Optimization	O	X
Expected Improvement	O	X
Random Forest	X	O
Workload Detection	X	O
Clustering	X	O
Classification	X	O
Anomaly Detection	O	X
Tools	X	X
PostgreSQL	X	X
OLTP-Bench	X	X
HyperMapper	X	X
DBtune	X	X
rsync	X	O
Data Collection Tools	X	X
Amazon Web Services	H	X
Related Work	X	O
Methodology		
Hardware	X	X
DBtune Implementation	X	X
Benchmarks	X	X
Query Collection	X	O
Hardware Collection	O	X
Query Statistics Processing	X	O
Hardware Statistics Processing	O	X
Multi-Workload Testing	X	H
Workload Detection	X	O
Detection Methods	O	X
Performance Measurement	O	X
Implementation for queries	X	O
Results		
Experimental Settings	X	X
Optimal Configurations	X	O
Multi-Workload Optimization	X	C

continuing on next page

section	Jonas	Viktor
Hardware Statistics	O	X
Query-Based Statistics	X	O
Discussion		
Regarding credibility of results	H	X
Optimal Configurations	X	O
Multi-workload Optimization	X	O
Query-based Statistics	X	O
Hardware Statistics	O	X
Conclusions & Future Work		
Conclusions	X	O
Future Work	X	X
finished		

Authors: Jonas Boström & Viktor Olsson
Supervisor: Dr. Luigi Nardi
Examiner: Christoph Reichenbach

Orignal Title: Workload detection and Continuous Automatic Bayesian Optimization in Database Management Systems

Making servers faster

To reduce costs and energy-demands for database-servers, it's important to ensure each individual server performs as well as possible. This thesis examines automatic server optimization in regards to the database's current workload.

With a world becoming more and more digital, we also become more dependant on databases, their hosting systems known as Database Management Systems(DBMS), and the physical hardware of the servers themselves. These servers require plenty of energy for execution and cooling, large warehouses for storage of these and all the metals required to build them. For these reasons, it is in the publics interest that servers become as fast as possible so that all these demands are lessened. By analyzing the DBMS current workload it is possible to adjust the server configuration in such a way that the performance greatly increases. But what if the current workload was to shift into another workload? In that case, it is possible that the current server configuration isn't perfect. Therefore, if it is possible to find out when the workload shifts in this manner, this optimization can be restarted to ensure the fastest possible server. In our thesis, it is this optimization and associated workload-shift classification that has been the focus. We design a system capable of identifying and optimizing for multiple workload. As for workload classification, two different methods are investigated. The first is through data associated with hardware-metrics, such as processor-usage. For the second method, data is gathered via a connection directly to the DBMS and investigating what tasks, known as queries, the server is executing. The results found that the multi-workload system designed was a success, and could successfully restart optimization when new workloads are detected. It was found that queries could perfectly disintguish between the workloads chosen. The same was found to be true for hardware-metrics.