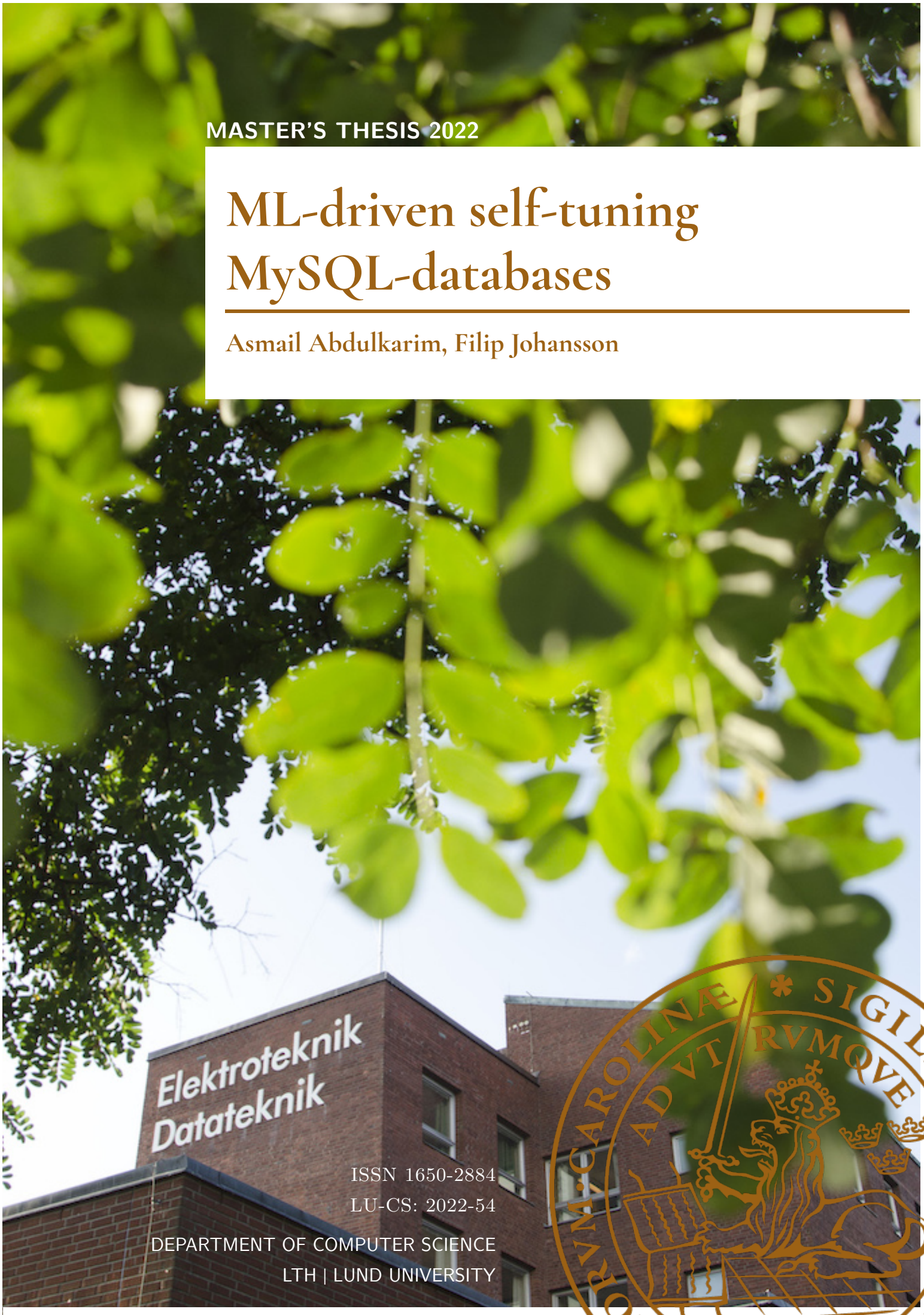

MASTER'S THESIS 2022

ML-driven self-tuning MySQL-databases

Asmail Abdulkarim, Filip Johansson

ISSN 1650-2884
LU-CS: 2022-54

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS: 2022-54

ML-driven self-tuning MySQL-databases

ML-drivna självjusterande
MySQL-databaser

Asmail Abdulkarim, Filip Johansson

ML-driven self-tuning MySQL-databases

Asmail Abdulkarim
asmail790@gmail.com

Filip Johansson
safilipjohansson@gmail.com

June, 2022

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

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

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

Abstract

As databases has become a more critical part of our digital infrastructure, the cost of maintaining a database has increased. Database management systems such as MySQL allow users to tune parameters, or "knobs" as they are called, to improve performance. This is a difficult optimization problem, both for humans to do manually and for computers. This is because there are hundreds of knobs with complicated dependencies between many of them and because the optimal configuration depends on the application. The optimal configuration then also changes over time as the workload changes. For these reasons automating this process is of interest to many companies and database administrators. In this project, we use an existing database optimization tool and investigate the possibility of defining a search space for MySQL capable of producing good configurations in reasonable time for any workload.

For this task, we used database benchmarking, which are arbitrary databases exposed to some arbitrary workloads. Performance was measured using throughput, or units of work per time unit. We selected three different representative benchmarks sufficiently different from each other. We then defined a large search space of 52 knobs that had some chance of being impactful, investigated how long warm-up and measurement time we had to use when benchmarking, collected 1040 data points for the three benchmarks and then used four different feature importance methods to identify the most important knobs. The 9 most impactful ones were then collected into a final search space and tested on the three benchmarks. This search space was able to find configurations with between 2x and 4.43x higher throughput than the default configuration in just a few hours. Additionally 12 knobs were mentioned as "honorable mentions" and we conclude that further performance gain is possible through incorporating some of these into the final search space.

Keywords: MySQL, Database management system, knobs, tuning, feature importance, optimization, machine learning

Acknowledgements

We would like to thank our supervisor Luigi Nardi, and to his team at DBtune for their help. In particular Muhammed Umair. Furthermore, we are also grateful to Erik Hellsten for his input about HyperMapper. This project would not have been successfully completed without their help and their knowledge in the field of database configuration tuning.

Asmail Abdulkarim & Filip Johansson

Contents

List of Figures

List of Tables

Chapter 1
Introduction

A database is an organized collection of data and needs to be managed by a database management system (DBMS). The database management system makes it possible to read, update, delete and insert data in a database. Database systems are an integral part of our modern society. To manage the large quantities of data and the high number of users, a need has grown for high-performance database systems. Performance is mainly measured in throughput (units of work per time unit) and it can be improved in various ways such as by optimizing the queries, the database structure and the applications interacting with the database. It can also be increased by tuning the internal parameters of the database management system. The focus of this master's thesis is the DBMS MySQL which is used in wide range of areas: in Media & Entertainment such as by BBC, in Retail such as by Axfood AB, in Government such as by NASA, in Telecom such as by Verizon Wireless, in Healthcare & Pharma such as by Eli Lilly, in social platforms such as by Twitter [21] and so on. At the time of writing this it is the second most popular DBMS according to DB-Engines, just after Oracle in the number one spot [22].

In this master's thesis the task was to optimize the internal parameters, or knobs as they are often called, for MySQL databases. This is difficult for both humans and computers, mainly due to the large number of knobs and due to the time-consuming nature of measuring the performance, which we describe further in the background section. We used three different benchmarks from a benchmarking tool called BenchBase, which are arbitrary databases exposed to arbitrary workloads. For the optimization tool we used HyperMapper, which is based on Bayesian Optimization, all described in the theory and tools sections in chapter 2. We took the approach of defining a large search space of knobs that we identified as having some chance of being important for throughput. We then investigated how long "warm-up" and "measurement time" we needed to use when benchmarking and then sampled the search space for ten days using the three benchmarks. On the resulting data sets we then used four different feature importance methods to identify the most important knobs. Nine knobs were identified as seemingly more important than the others and an additional twelve were identified as "honorable mentions". The top nine knobs were collected into a final search

space that within a few hours was able to find configurations with between 2x and 4.43x higher throughput than the baseline default configuration, and between 3% and 10% higher than the baseline option innodb_dedicated_server that when turned on automatically sets four of the most important knobs.

Since we found in the sampling of the original large search space configurations with 65% higher throughput than with innodb_dedicated_server turned on for one of the benchmarks we concluded that through further refinement of the final search space, by including some of the "honorable mentions", it is possible to find configurations with even higher throughputs. More testing is therefore needed before one can settle on a search space suitable for any situation and any workload, and we write this down as future work.

Chapter 2
Background and Related Work

2.1 Background

2.1.1 Database configuration tuning

Databases are a vital part of many applications. They are often large and commonly serve hundreds of users trying to access them concurrently with some combination of read- and write-type queries. In order to improve the database's performance, database management systems (DBMS) allow administrators to tune parameters or "knobs" as they are called. Optimizing these knobs for throughput, meaning number of transactions or "units of work" per time unit, impacts both wait times and costs. This is perhaps especially of interest when one uses a cloud service such as AWS and compute time directly translates to money spent.

Optimizing database knobs for throughput is an impossibly difficult problem though, and for several reasons. Not only do many DBMS's such as MySQL and PostgreSQL include hundreds of knobs with varying value ranges, creating an enormous search space, but there are also dependencies between many of them such that changing one knob impacts others. Furthermore every application is unique since the optimal configuration depends on things such as the hardware setup and on the database itself (its structure, size and so on). The optimal configuration also changes over time as the workload changes. There is also a limit on how many configurations that can be evaluated in reasonable time, since one needs to observe the throughput over some time period. This will typically limit the number of evaluations to a maximum of a few hundred, making gradient-based optimization techniques impractical.

Traditionally this tuning is done manually by a database administrator (DBA) using a set of guidelines and hard-earned experience. The DBA might go through a time consuming and difficult process of trying to Figure out what the bottlenecks are and which knobs to tune, followed by some manual trial and error. The bottlenecks for performance can be a number of different things relating to disk input and output, efficiency of utilizing the faster RAM memory for speed ups, threads operating concurrently, sizes of things and etc. DBAs

might spend upwards of 25% of their time tuning and in total personnel is roughly 50% of the total cost of ownership for a large scale database system[23]. For all these reasons automatic database configuration is a growing research domain, of interest to many organisations and DBAs.

2.1.2 Problem statement

The objective is to find an optimal configuration

$$\mathbf{x}^* \in \arg\max_{\mathbf{x}\in\mathbf{X}} f(\mathbf{x}) \tag{2.1}$$

where $\mathbf{X}$ is the set of all possible MySQL configurations and $f(\mathbf{x})$ is throughput based on a specific configuration $\mathbf{x}$. Instead of directly trying to solve this, we take the approach of reducing the dimensionality of the search space $\mathbf{X}$, to then optimize over this smaller search space, keeping every knob not included at its default value. With a limited time budget the hope is that this approach should be able to find better configurations, compared to solving the larger search space.

2.1.3 DBtune

This master's thesis is done in collaboration with the startup company DBtune. The company has developed a service for automatically tuning database knobs. DBtune use the open source Bayesian optimization tool HyperMapper [16] to optimize database knobs in real time on existing databases. We got to use the company's existing framework and code for optimization, as well as ask questions to members of the team. DBtune has developed a working product for the DBMS PostgreSQL, as well as some other DBMSs. Testing is done using the database benchmarking tool BenchBase, a newer version of OLTPBench [8].

2.2 Theory

2.2.1 Bayesian optimization

Bayesian optimization is a black-box derivative-free global optimization technique. It builds a surrogate model for the objective function, treating every point as unknown with some prior probability distribution, making it a so called random field. Commonly one assumes a prior normal distribution using some mean function, often just a constant μ, and some covariance function to account for the prior variance and for how correlated different points should be, turning the surrogate model into specifically a Gaussian process. The variance is decided with a multiplicative constant a_0 and the covariance function also includes scaling factors $\boldsymbol{a_{1:d}}$ for every dimension. It is constructed to be stationary, so that only the (weighted) distance between two points matters. One can use for instance the exponential covariance function

$$C(\boldsymbol{x}, \boldsymbol{x'}) = C(\|\boldsymbol{x} - \boldsymbol{x'}\|) = a_o \exp(-\|\boldsymbol{x} - \boldsymbol{x'}\|^2) \tag{2.2}$$

where $\|\boldsymbol{x}-\boldsymbol{x'}\|$ is the weighted distance between $\boldsymbol{x}$ and $\boldsymbol{x'}$ according to $\|\boldsymbol{x}-\boldsymbol{x'}\|^2 = \sum_{i=1}^{d} a_i(\boldsymbol{x_i}-\boldsymbol{x'_i})^2$. This is indeed stationary.

Other covariance functions often include some shape parameter usually denoted ν. Collecting the parameters as $\boldsymbol{\theta} = (\boldsymbol{a_{0:d}}, \nu, \mu)$ one wants to first estimate the most likely $\boldsymbol{\theta}$ given available training data $\boldsymbol{y}$. This means finding the maximum of $p(\boldsymbol{\theta}|\boldsymbol{y}) \propto p(\boldsymbol{y}|\boldsymbol{\theta})p(\boldsymbol{\theta})$ where one can either put some prior on the parameters or leave it flat. By assumption $p(\boldsymbol{y}|\boldsymbol{\theta})$ is multivariate normal $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ where the expression for the covariance matrix (or kernel) $\boldsymbol{\Sigma}$ is calculated using the covariance function. One can then solve this (d+2)-dimensional optimization problem using some traditional optimization technique.

Once the model parameters are set, one can evaluate the posterior probability distribution of the output of the objective function at any point $\mathbf{x}$ given the known values of the training data, a process called Gaussian process regression or Kriging. The initial data is usually sampled at random points across the search space, using for example the technique Latin hypercube sampling. We simplify the posterior notation by defining:

$$f(\mathbf{x})\big|f(\mathbf{x}_{1:n}) \equiv f(\mathbf{x})\big|[f(\mathbf{x}_1), ..., f(\mathbf{x}_n)]. \tag{2.3}$$

Bayesian optimization is a technique of choosing the next point to evaluate given this posterior distributions over the field, using some so called acquisition function. Most commonly the expected improvement acquisition function is used, which for every point defines the expected value of the increase of the maximum value evaluated so far if that point is evaluated. It is defined as

$$EI_n(\boldsymbol{x}) := E_n[[f(\mathbf{x})|f(\mathbf{x}_{1:n}) - f_n^*]^+] \tag{2.4}$$

where f_n^* is the best point so far and the plus sign indicates only using positive values (setting negative values to zero). Points that are far from other previously evaluated points have high variance, and therefore possibly a lot of energy above the current maximum value. Evaluating these types of points is called exploration. Points closer to previously evaluated points with high values have a higher expected mean, which might give them a lot of energy above the current maximum value, in spite of their lower variance. Evaluating these types of points is called exploitation.

The expected improvement has a closed form expression, which can be obtained by using integration by parts [10]. The maximum can be found using some traditional first- or second order derivative-based optimization technique. One then evaluates the objective function at that point, use this additional training point to update the model parameters, calculate the next maximum expected improvement point and so on.

The Gaussian process surrogate model was recently thought to be the only surrogate model used for Bayesian optimization according to a tutorial from 2018 [10]. However one can imagine doing this with any regression model plus a model for the uncertainty or variance. It is natural to do with a random forest model for regression, where a number of so-called decision trees each independently models the value at a certain point, using different subsets of the parameters. The mean and variance of the individual trees predictions at point $\boldsymbol{x}$ can then be used to model the distribution of $f(\mathbf{x})\big|f(\mathbf{x}_{1:n})$, and the expected improvement acquisition function can be used in the same way. Random forest models have the advantage of naturally being able to handle categorical parameters.

2.2.2 Benchmarking

Benchmarking is a common tool used by DBAs to gain insight into how hardware, knob configuration and application impacts the performance of a database. A benchmark loads data into a new database to then expose it to a workload. A workload is a set of queries performed by a number of arbitrary users. The workload can be read- or write-heavy and the queries can be more or less complex. Metrics such as throughput and latency are observed every second. When benchmarking one needs to decide which warm-up time (the initial time period where the benchmark is running but no measuring is performed) and measurement time to use. We here describe the three different benchmarks used for this project, all implemented into the tool BenchBase.

TPC-C The TPC-C benchmark is the current industry standard for evaluating OLTP (Online transaction processing) database systems. It models wholesale suppliers and contains common transactions observed in the wholesale supplier industry such as customer orders, customer payments, ship orders, and warehouse inventory queries [7]. The TPC-C benchmarks are complex and write-heavy [8].

Twitter The Twitter benchmark implemented in BenchBase mimics common patterns found in the popular social media platform Twitter such as read-heavy and skewed many-to-many relationships. The database is based on an anonymized social graph network snapshot from Twitter in 2009 [8].

YCSB The Yahoo Cloud Serving Benchmark, YCSB is intended to be used primarily for benchmarking cloud databases and in particular serving systems. Serving systems provide low-latency acesss both for reading/writing data. Typically a web page server is a serving system since user expects a short web-page loading time. This is contrasted to relational OLAP systems which usually have high latency (due to complex queries). The key differences between cloud database systems and traditional database systems are the requirements for elastic scaling (on-demand scaling), availability, and simplified application deployment and development. Due to the relatively high hardware failure rate presented in horizontal scaling, fault tolerance is considered to be important property. As a consequence, many cloud systems sacrifice complex queries and strong ACID (atomicity, consistency, isolation, durability) transaction properties to achieve availability [6].

Many cloud databases are key-value stores which is why YCSB benchmark consists of simple operations. Although MySQL is not a key-value store, it is still a useful benchmark to learn how availability, consistency, and caching impact the performance [8].

2.2.3 MySQL

MySQL is a popular database management system owned by Oracle. It comes in different editions: “MySQL Standard Edition”, “MySQL Enterprise Edition”, “MySQL Cluster Carrier Grade Edition” and “MySQL Community Edition”[1]. The community edition is open source and licensed under GPL while the others are commercial [2]. MySQL Enterprise Edition pro-

[1] `https://www.mysql.com/products/`
[2] `https://www.mysql.com/products/community/`

vides the customer with additional tools and plugins for managing scalability, security reliability, availability, and application development, which are unavailable with the community edition or improvement upon. One of the more prominent additional features in the Enterprice Edition for performance is the "MySQL Thread Pool" server plugin. With the enterprise thread-handling model, it is possible to connect many more clients without being burdened with thread overhead. MySQL enterprise edition achieved 60 times better scalability for the Sysbench OLTP read/write benchmark than the community edition [20]. We will only focus on the community edition.

The heart of MySQL is the storage engine. There are several options, but we only focus on the default InnoDB storage engine. The engine has in-memory structures as well as on-disk structures, see Figure 2.1.

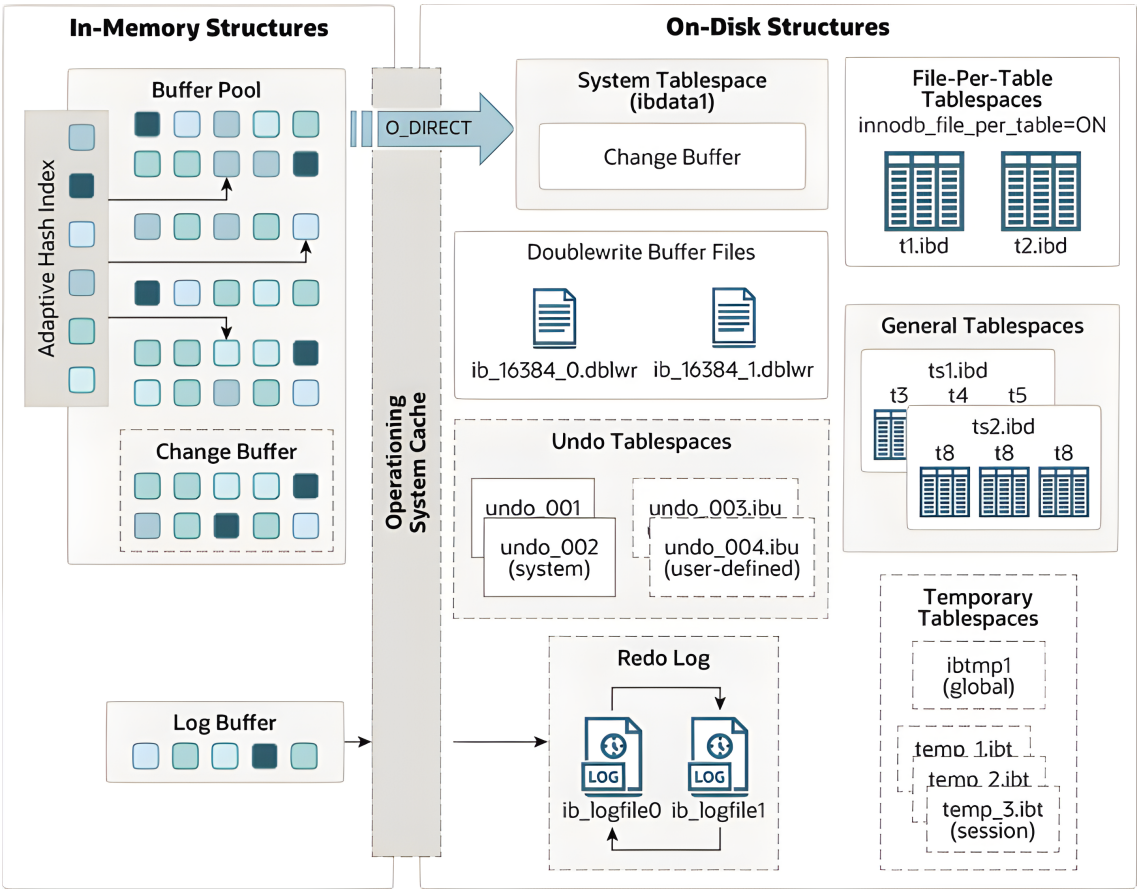

Figure 2.1: Diagram of the structure in the InnoDB storage engine. From the MySQL documentation [17]

The in-memory structures are the buffer pool, the change buffer, the adaptive hash index and the log buffer [18]. The buffer pool stores parts of the database in memory for faster access. The complete database is however stored on disk, and so the buffer pool needs to continuously flush modified data to the disk. There are so called "page cleaner" threads that do this, and several knobs have to do with optimizing this process, deciding things like what data to flush and when. Ideally data that is read or written too often should be stored in memory, which enables read-, as well as write-, queries to complete quickly and the flushing to disk to be done later by the page cleaner threads. The data structure of the buffer pool is

that of a least recently used (LRU) list, where data is moved to the top when accessed again, keeping it in the buffer pool longer. When first inserted into the buffer pool from disk, the insertion point is not at the top but rather 5/8 of the way down, decided by the knob innodb_old_blocks_pct. Data is also only moved to the top if at least one second, decided by the knob innodb_old_blocks_time, has passed since it was first inserted. This is to make data recently accessed more than once on separate occasions stay in longer, since this would increase the probability that it is going to be accessed again. Furthermore an algorithm called read ahead is implemented that tries to predict what data is soon to be accessed, preemptively putting it into the buffer pool.

The change buffer caches changes with INSERT, UPDATE and DELETE queries to be merged later. This does not necessarily need to happen until data is read from disk into the buffer pool when it is accessed by the user with a read operation. Waiting and doing it in blocks reduces random access disk input/output. This can be modified to only be done for some operations and not for others with one knob innodb_change_buffering and the maximum size of the change buffer can be decided by another knob called innodb_change_buffer_max_size.

The adaptive hash index keeps pointers to data in the buffer pool for fast look up. The log buffer holds log data to be written to disk.

The on-disk structures are all the different possible types of tablespaces where data is stored, the doublewrite buffer, the redo log and the undo log [19]. Tables, which contain the actual data, are by default created in so called file-per-table tablespaces. There are then one file for each table. Tables can also instead be created in system tablespaces or manually by the user in general tablesspaces, both containing more than one table per file. The user can also create session temporary tables in temporary tablespaces.

Undo logs, which contain information about how to undo the latest changes, are stored in undo tablespaces. If things are modified and also read simultaneously with a consistent read operation by a different user, the read operation needs to check the undo logs to retrieve the unmodified data.

When data is flushed from the buffer pool to disk, it is first stored in the doublewrite buffer to then be written into the proper positions in tablespaces. This is for crash recovery reasons, and does not add a lot of overhead since data can be written to the doublewrite buffer in one go sequentially. Distributing data into the proper tables is then more time consuming and I/O demanding.

Redo logs store information about ongoing changes and inserts in case of sudden crash. The changes and inserts can then be performed when the database comes online again.

There are knobs deciding how many threads are allowed to operate on InnoDB simultaneously, and how to decide which threads are allowed inside using so called “concurrency tickets” that every thread gets a number to use before having to move back to the end of the queue. Here is a trade off between letting large transactions complete and smaller ones being able to sneak in without having to wait forever.

The few knobs mentioned here are just examples. Overall the performance knobs in MySQL and InnoDB have to do with things like disk input/output, efficiency of utilising RAM memory, threads operating concurrently and the maximum sizes of different things.

2.2.4 Feature importance

We are interested in trying to figure out which knobs have the greatest impact on throughput. This is different from unsupervised dimensionality reduction techniques such as principal component analysis, since here we are interested in the impact on an objective.

One approach is to fit a linear regression model to training data while putting a penalty on the model weights, then while increasing the penalty, observe in which order the weights go to zero. This approach is called Lasso. One could also use the R-squared measure of what proportion of the variance is predictable by different knobs such as in a paper by Kannellis et all [12] on reducing the number of knobs to tune for the DBMSs PostgreSQL and Cassandra.

One problem with simply using linear regression methods is that it does not care about where the default point is. If the default value on one knob is good and all other values are much worse, then it is going to look as if that knob is very important to include in the search space. What we really want is to find out which knobs can be improved significantly from their default value. We use four feature importance methods, one that comes out naturally from HyperMappers random forest model, and three implemented in the tool CAVE (which described in section 2.3.3). Note that since CAVE uses an existing data set it doesn't know the exact values of all the points used by the feature importance methods, and so instead it will fit a random forest model to the data and compute the feature importance values on the model.

Random forest

HyperMapper uses Gini importance to estimate the importance of a parameter. Gini importance is an importance method for measuring the total reduction in impurity by splits using feature X in a random forest. HyperMapper uses random forest regression (the target variable is continuous) and variance reduction as the splitting criteria (the criteria used for dividing the data into smaller, more homogeneous groups). The importance of a feature $\boldsymbol{X}$ of a random forest regression with reduction variance as the splitting criteria can be expressed as the weighted sum of each node's contribution to a decrease in variance, weighted by the proportion of samples reaching that node and averaged over all trees of the ensemble [14]. Mathematically, the importance of a future $\boldsymbol{X}$ for a particular node $\boldsymbol{A}$ in a regression tree (with reduction variance as the splitting criteria) can be calculated as follow:

$$\text{imp}_{\text{A}}(X) = \frac{N_A}{N_{total}} \cdot \left(V_A - \frac{N_{A_R}}{N_A} \cdot V_{A_R} - \frac{N_{A_L}}{N_A} \cdot V_{A_L} \right) \tag{2.5}$$

where N_A is the number of samples who reached node $\boldsymbol{A}$, N_{A_R} the number of samples who reached the right child node of $\boldsymbol{A}$, N_{total} the total number of samples in the tree, V_A the variance of node $\boldsymbol{A}$, V_{A_R} the variance of the right child node of $\boldsymbol{A}$. The two variables N_{A_L} and V_{A_L} is defined similar to N_{A_R} and V_{A_R} respectively but denote the left child node. The expression $\frac{N_A}{N_{total}}$ is the weight for node $\boldsymbol{A}$ and the expression $(V_A - \frac{N_{A_R}}{N_A} \cdot V_{A_R} - \frac{N_{A_L}}{N_A} \cdot V_{A_L})$ is $\boldsymbol{A}$'s contribution to a decrease in variance. By summing over all nodes in the regression tree using Equation 2.5 we get the importance of $\boldsymbol{X}$ for the regression tree. By dividing the sum with the number of trees in the ensemble we get the importance of $\boldsymbol{X}$ for in the random forest regression [14]. In the later sections we will use the abbreviation RF to refer to Gini importance.

Ablation analysis

The feature importance method ablation analysis (sometimes abbreviated to AA) implemented in CAVE [3] is based on observing the changes of the objective function in stepping from the default configuration to the best known configuration, the so called incumbent. Starting at the default configuration, it observes the change in the objective function (modeled using a random forest model) in changing one single parameter to its incumbent value. It does this with every parameter. Out of all these configurations, with only one change compared to the default, the highest increase is observed. This increase is a percentage of the total increase in moving from the default to the incumbent configuration, and that is the importance value given to that knob. The algorithm then moves to this configuration, with every parameter at its default value except one, and from there repeats the process with every other knob, changing their value to the incumbent value and computing the biggest increase and the importance value. It is theoretically possible to get negative importance values, and when this happens we change it to zero and normalize the other important values to sum to one.

Ablation analysis is of course more reliable the closer the incumbent is to the true optimum. Furthermore when done on a model as in CAVE as opposed to sampling the specific configurations needed for the analysis, reliability improves with the accuracy of the model. The model specifically needs to be accurate around the hypercube between default and the incumbent. More samples in this area make the method more reliable.

This is of course only one of the possible ways of changing values from the default configuration to arrive at the incumbent configuration, and this illustrates the difficulty in defining a universal feature importance method. Ablation analysis should be fairly reliable if there is a linear relationship between parameters and the objective in the area between default and the incumbent, but wherever there are dependencies between parameters it is going to over- and undervalue some of them. The concept of value or importance as a whole also becomes harder to define when considering parameters one by one. With these factors in mind ablation analysis should give an indication of which parameters are most important to change from their default to their incumbent value.

Local parameter importance

Another feature importance method implemented in CAVE is called local parameter importance (LPI). It also uses the incumbent, and keeps everything constant except one parameter. If the parameter has n discrete values, the algorithm computes the value of the objective function for those n values and computes the variance of those values. In case of a continuous parameter, the algorithm uses 500 evenly spaced values. Parameters are given importance weights according to the variance for that parameter divided by the sum of all variances. This is a local method, since it looks at one parameter at a time, keeping every other parameter at the incumbent value. This useful since we care about which parameters are important around the incumbent, an area that the optimizer hopefully is going to explore more than all the corners of the search space.

fANOVA

The third method implemented in CAVE is called fANOVA. This is a global method that investigates first order effects and pairwise second order effects. For the first order effect, similar to local parameter importance, it investigates the variance when changing one knob. The difference is that the algorithm computes the average across all other configurations of the entire search space while keeping that single knob fixed, for each value of that knob. In case of a continuous variable, it is not split into 500 evenly spaces values as in LPI, but instead the normal definition of the variance of a continuous variable is used. The method also looks at pairwise second order effects, investigating the variance of the averages when changing two parameters. Just like LPI the importance is proportional to the variance and normalized so that all importance values sum to one.

Since this is a global method it is reliable only if given a sufficiently large data set and should be given less trust for smaller datasets. It can however give some indication of the importance of parameters in areas of the search space that LPI does not consider, which is useful since we are unlikely to be close to the incumbent until later in the optimization process.

2.3 Tools

Here we describe the tools used in our project.

2.3.1 BenchBase

BenchBase is a database benchmarking tool and the successor of OLTPBench which now is archived and no longer maintained. Benchbase target primarily relational database systems as they are harder to predict than key-value storage systems. BenchBase is written in Java and is open-source. BenchBase was created to fix common issues found in benchmarking tools, such as inadequate benchmark suites, no portability, insufficient set of metrics and lack of a straight forward mechanism for extending new benchmarks [8].

In BenchBase, the size of the database, the number of active terminals (SQL clients), the warmup time and the measurement time are all configurable parameters. The size of the database is determined by scalefactor. Scalefactor is defined differently depending on the benchmark. In TPC-C scalefactor is the number of warehouses [3]. In Twitter scalefactor is the number of users divided by 500 [4]. In YCSB scalefactor is the number of rows in the table USERTABLE expressed in 1000s[5].

2.3.2 HyperMapper

HyperMapper is an open-source multi-objective black-box optimization tool with support of three different optimization methods: Bayesian optimization, evolutionary optimization, and Local Search. However, its main strength is Bayesian optimization, making it practical

[3] https://github.com/cmu-db/benchbase/blob/main/config/mysql/sample_tpcc_config.xml
[4] https://github.com/cmu-db/benchbase/blob/main/config/mysql/sample_twitter_config.xml
[5] https://github.com/cmu-db/benchbase/blob/main/config/mysql/sample_ycsb_config.xml

for optimizing expensive black-box functions. HyperMapper utilizes a surrogate model of the black-box function to train efficiently and predict accurately. Currently, the surrogate model can either be a random forest regressor or a Gaussian process. HyperMapper is also capable of handling functions with unknown feasibility constraints [16]. It has been successfully applied on multiple optimization problems with huge search spaces: 3D vision, hardware design space exploration and database management systems parameter tuning [16] [15] [9].

In HyperMapper the optimizer starts with a design of experiment (DOE) phase before the Bayesian optimization phase. Points are sampled across the entire search space and used to initialize the surrogate model. The number of points generated and the sampling method used during the DOE phase are configurable parameters. Currently available sampling methods are random sampling, standard Latin hypercube, k-Latin hypercube, and grid search. With defaults settings HyperMapper uses expected improvement as acquisition function. HyperMapper can make use of prior knowledge about search space by injecting a list of probabilities or distribution modeling probability of a good value for each knob separately. HyperMapper also provides settings which allow users to observe the importance of each input parameter for each optimization iteration. This option is only available if the surrogate model is random forest [11].

2.3.3 CAVE

CAVE which stands for "Configuration Assessment, Visualization and Evaluation" is an algorithm configurator tool with the added benefits of a comprehensive automated report utility and without restriction to the type of algorithm. An algorithm configurator is a tool that optimizes hyperparameters of a algorithm. Many algorithms are highly parameterized and can be tuned to achieve good performance for a set of problem instances. According to the authors of CAVE, similar tools are either restricted to specific algorithms (such as the tool PAVE) or lack a utility for a thorough algorithm analysis (such as the tool HAL). The automated report includes feature importance methods such as ablation analysis, local parameter importance, and fANOVA. CAVE comes bundled with its own algorithm configurator called SMAC3. SMAC3 can utilize observed data from previous runs to predict the performance of a configuration-instance pair [3].

We note that tuning MySQL parameters for a specific benchmark can be viewed as algorithm tuning where the algorithm is MySQL, the instance problem a benchmark and the algorithm configuration being the MySQL parameters. CAVE is not actively maintained and uses deprecated dependencies which causes problem for the automated report generator. We used separate code for producing plots.

CAVE takes a data set and does feature importance according to the previously described ablation analysis, local parameter importance and fANOVA methods. Since it uses an existing data set it can't sample the points necessary, but instead it fits a random forest model to the data, and does the feature importance methods on the model. Its accuracy then depends on the quality of the underlying model, which should be improved by more data points around the relevant areas.

2.3.4 Hardware

The final benchmarks were carried out in AWS EC2 cloud service. However, before final benchmarks, multiple checks and preparation were conducted on private and school computers from LTH. The computers were specifically used for checking that the search space is reasonable, that its knobs values do not cause MySQL to crash, and different enough to capture its impact.

Amazon Elastic Compute Cloud (Amazon EC2) is a service where users can rent virtual machines. As mentioned in the name the service provides user with elasticity, i.e. to dynamically scale up and down depending on the workload, if one wants to. EC2 utilizes virtualization to partition physical computers into multiple virtual server machines [1]. The EC2 provides the user with a predefined set of instance catalogs tailored for different purposes, one of which is the M series, denoting a general-purpose computer. EC2 also allows users to choose the region where the computer will be located.

In this project, only m5d.2xLarge instance type was used. The m5d.2xLarge is a general-purpose computer with a decent amount of RAM (32 GB) and CPU power (8 vCPUs). The instance comes with a 300GB NVMe SSD storage physically attached to the host computer[2].

We believe that choosing a less powerful instance than m5d.2xLarge would not be realistic since most modern private computers have at least 8 cores. Choosing a more powerful instance would on the other hand be too expensive.

2.4 Related work

Kanellis et al. found that by tuning just the top five knobs for the database management systems Cassandra and PostgreSQL it was possible to get 99% of the improvement compared to tuning 29 (PostgreSQL) or 30 (Cassandra) knobs. They handpicked the search space by going through the documentation and prior work, and then gathered 25 000 samples over five days using 30 AWS instances, for two separate workloads. Using this massive training set they were able to do linear regression and even look at second-order effects of dependencies between knobs to pick out the five most important knobs. By then sampling 4000 points using the top five knobs and keeping every other knob constant, the best point had more than 99% of the improvement compared to default of the full search space, for both workloads. Their results indicate that identifying the most important knobs is of high value [13].

Similar work has been done for storage systems such as for one of the more popular Linux's file-formats Ext4 where the search space was sampled using Latin hypercube sampling. The authors used reduction in variance to determine feature importance. Their search space had 29 544 possible configurations, and they ran every configuration three times over a few months, creating a large training set [4].

2.4.1 OtterTune

OtterTune is an automatic database configuration tool based on machine learning techniques. Conceptually it can be divided into 3 parts: workload characterization, knob identification and automatic tuning. By learning distinctive features of the target workload it can utilize previously stored configuration to accelerate the tuning process. Since the purpose of

this thesis is to find the most impactful knobs the workload characterization part will not be described.

For finding knobs with high impact OtterTune use Lasso regression and in particular Lasso path algorithm which perform automatic feature selection and ordering of features by importance. By including polynomial features in the regression dependencies between knobs can be captured.

In the first step in the tuning process, OtterTune searches for a similar workload by comparing the Euclidean distance between different workloads for each filtered metric separately. It then proceeds with choosing the most similar defined by the lowest average of the chosen metrics. In the second step, Gaussian process regression is used to recommend new configurations that it believes will improve the target metric. OtterTune does not use all knobs in a tuning session since it would be too large search space. Instead, it starts with the most important ones and over time increases the number of knobs used [23].

In one of their experiments, for the benchmark TPC-C, OtterTune concluded the 10 most impactful knobs for MySQL are:

- innodb_buffer_pool_size
- innodb_thread_sleep_delay
- innodb_flush_method
- innodb_log_file_size
- innodb_thread_concurrency
- innodb_max_dirty_pages_pct_lwm
- innodb_read_ahead_threshold
- innodb_adaptive_max_sleep_delay
- innodb_buffer_pool_instances
- thread_cache_size

They used the AWS EC2 service with Ubuntu OS and MySQL version 5.6. The benchmarking tool OLTpbench were installed in a separate server: OLTpbench in a m3.xlarge instance and MySQL in a m4.large instance. The OtterTune's tuning manager was deployed on a local server with 128 GB RAM and 20 cores. They used an 18GB database size [23].

Chapter 3
Methodology

3.1 Converting HyperMapper results to CAVE format

The process of converting HyperMapper CSV files to SMAC3 so-called runhistory files is quite simple. SMAC3 accepts a multitude of different runhistory file formats. One of which happens to be CSV file format with a quite similar structure to HyperMapper's CSV file.

A runhistory file contains information about algorithm runs. It contains the input, the output (expressed as cost), along with some extra information for each run: the status of the run (TIMEOUT, CRASHED, ABORT, MEMOUT, CAPPED, etc.), the total CPU time for the run and the input seed (for the random generator inside the algorithm) [5]. CPU time is time spent on processing the algorithm, not including sleep time and time for IO operations [1]. Since MySQL does not use seed value, the seed value was set to one. The CPU time was also set to a constant value since this metric is not tracked. For all runs, the status field was set to SUCCESS. Unsuccessful runs were encoded as having a cost value positive 1. Since a successful run cost is between $[-\infty, 0]$, $+1$ is always the largest cost. An unsuccessful run can alternatively be coded with the value CRASHED.

Analysis methods like LPI and AA require one additional file called trajectory which keeps track of the current best configuration throughout the runhistory. In trajectory file only incumbent runs are included, so if a run performs worse than the current best it is not included in the trajectory file [5]. For example, if the cost is -1 for run x_1,-2 for run x_2,-5 for run x_3,-3 for run x_4,-7 for run x_5,-1 for run x_6 and -10 for run x_7 then the trajectory file would contain x_1,x_2,x_3,x_5 and x_7 in the order as presented. The trajectory file also requires total elapsed time (the total time from start to finish for a run). This is also set to a constant value. The trajectory file is created from the runhistory file with the help of a script provided

[1] https://www.gnu.org/software/libc/manual/html_node/Processor-And-CPU-Time.html#Processor-And-CPU-Time

by our colleague. The script can be found in appendix D.1.

3.2 Design space

3.2.1 Knobs and value ranges

The knobs to use and values to include for the training set were decided after a process of reading MySQL documentation and other sources. We first studied the structure of MySQL and the storage engine InnoDB, while summarizing the function of many (or most) knobs. Then we categorized every knob as green (probably matters for performance), yellow (might matter for performance), orange (probably does not matter but possibly or is niche), and red (does not matter).

We then refined our categories, especially by looking through the orange knobs again, and decided to use all 52 green and yellow knobs for our training set. We then went through those again, deciding on value ranges that would be wide enough to cover the possible optimal values, but not unnecessarily wide. During this phase we tested different values for some knobs to check if the ranges seemed reasonable. Once the 52 knob search space was finished we tested it on 200 random points on our private computer to ensure we would not crash the system too often and none of the points concluded in a crash.

3.2.2 Benchmarks

Initially, TPC-C, Wikipedia, and CHBenchmark were selected as final benchmark candidates. However, due to unexpected bugs in BenchBase, TPC-C, YSCB and Twitter were ultimately chosen. The primary reason for choosing these is for their differences in write-read distribution. Twitter is read heavy, TPC-C write heavy and YCSB balanced [8]. CHBenchmark would have contained a combination of complex queries as well as simpler ones, but the balanced read/write ratio of YCSB also makes it an important benchmark to test along with the write- and read-heavy TPC-C and Twitter.

3.2.3 Scalefactor and number of terminals

Setting scalefactor is straightforward. The database size should be at least double the RAM size. The reason for this is to make it impossible for MySQL to cache more than half of the database into RAM since real-world databases are usually at least several times larger than the RAM. We ended up selecting the following scalefactors 650, 14000, 50000 for Twitter, TPC-C, and YSCB respectively. All three scalefactors translates to a 64GB database.

Setting the number of terminals for the benchmarks required additional work. Some random configurations were sampled and only the best and worst configuration in terms of throughput was kept. We then sampled new points using the best configuration but with different number of terminals. We thereafter sampled new points using the worst configuration and using the same settings for number of terminals as those used with best configuration. The setting that produced the largest difference in throughput between the best and worst configuration was selected. There is a reason why we proceeded in this way. If the system is not stressed hard enough then there is no point in optimizing. A bad configuration is equally

good as an optimized configuration in terms of performance. The same could be said if the system is overstressed. All configurations will perform poorly even, optimized configurations. The system needs to be stressed hard enough but not too hard so there is room for optimization and thus making the impact of knobs clearer for feature selection methods. We ended up selecting the following number of terminals: 600, 200, 400 for Twitter, TPC-C and YSCB respectively.

3.2.4 Warmup and measurement time

When benchmarking one needs to decide how long warm-up and how long measurement time to use. A warm-up period is needed because it takes a while for the throughput values to stabilize. Using a long warm-up such as 15 minutes and a long measurement time such as 30 minutes should give a very stable and precise approximation of the throughput for the given workload using that particular configuration. But 45 minutes would be a long time for every iteration, so we want to investigate how much shorter warm-up and measurement time we can use and still get good approximations of the throughput. Ideally, we want to rank different configurations correctly relative to each other, but using a shorter warm-up and measurement time. We don't want the shorter warm-up and measurement time to overvalue some configurations and undervalue others. We therefore came up with the strategy of letting ten random configurations run for 45 minutes with zero warm-up to then afterward investigate the average throughputs of different warm-ups and measurement time values. A window with starting point x and width y corresponds to warming up for x seconds and then measuring for y seconds.

We observed the average throughputs for each configuration over the window starting at 900 seconds (15 minutes) to 2700 seconds (45 minutes), to get an approximation of the true average throughput. We then investigated if we could use some earlier time window at starting point x and width y, corresponding to a warm-up period of x seconds and a time window of y seconds and still preserve the order and distances between the ten configurations.

3.3 Data collection

To be able to do feature importance $52 \cdot 10$ points were sampled in the design of experiment phase using Latin hypercube and $52 \cdot 10$ in optimization phase for each benchmark. All 52 knobs were included in the search space. The purpose of the test is to find the most important parameters. The reason for not doing only Latin hypercube sampling for search space coverage is that the optimizer will (hopefully) zoom in on the higher throughput points, creating more samples around those areas. This approach is tailored to the specific feature importance methods implemented in CAVE that investigate what is happening around the default point and the incumbent (ablation analysis and local parameter importance). We want CAVE to be able to build a more precise underlying model around those areas.

3.3.1 Necessary changes for MySQL and AWS instance

Small changes were made to MySQL setting for benchmarks to work properly. The knob *max_connections* was set to 650. The value was changed to specifically 650 since 600 terminals are created in Twitter benchmark and we need some extra to avoid deadlocks.

We decided to disable binary logging during the data loading for two reasons. The primary reason is that binlog files created before a benchmark run, do not impact the result of the benchmark. Binlog files are binary log files that contain records of all statements that modify and potentially could modify the data (eg. an UPDATE with no matched rows). Binlog file is only used during a replication and data recovery which in our case will not happen. Keep in mind that binloging was only disabled during data loading but not during the actual benchmark since binlog could potentially impact IO performance. The second reason is binlog files take quite a large amount of space and as a consequence, longer time restoring. Also, some YSCB benchmark runs ended up crashing due to being out of space.

For all the computers 16GB of swap memory was created on the SSD device. The reason is to avoid complete system failure due to running out of available RAM. We estimated that MySQL RAM usage will roughly peak at 29 GB based on the knobs buffer_pool_size and temptable_max_ram. However how the remaining knobs and how BenchBase affects the RAM is unclear, hence why swap memory was added.

3.3.2 HyperMapper configuration

For all tests, standard Latin hypercube sampling was used as the DOE sampling method, random forest as the surrogate model, and Bayesian optimization as the optimization method. All other parameters in HyperMapper, use their own default value.

3.3.3 Database restoration process

It was noted that running multiple benchmark runs in succession, without restoring the database to its original state between benchmark runs, would result in a noticeable performance downgrade as it continues. That's why a backup of the database was created before any benchmark runs. Restoration was achieved initially by a tool called `rsync` (which is automatically installed in Ubuntu) but later replaced with a simple delete, copy, and paste as it was observed to be faster.

3.3.4 Black-box function

The optimization objective of this experiment is throughput. Since HyperMapper only excepts a minimization problem $-1 \cdot \texttt{throughput}$ is returned from the black-box function. Since database crash is inevitable especially when knob search space is large, it was decided the throughput value 1 should be used for signalling a database crash. The throughput 1 represents the worst option because all successful benchmark runs will return a value between $(-\infty, 0]$. Before performing the actual benchmark the black-box function would write the

new configuration to a file that is parsed by MySQL, clear memory cache and restart MySQL. Memory cache is cleared by the command:

```
sudo sh -c "echo 3 > /proc/sys/vm/drop_caches\"
```

Linux often caches frequently accessed data on disk into memory to reduce IO which causes benchmark runs to be more nondeterministic.

3.4 Feature importance and identifying the most important knobs

In order to identify the most important knobs to use as a "final" search space for testing, we looked through plots with the four feature importance methods for the three benchmarks. We put more trust in LPI and especially AA, on all three benchmarks, because of their local nature and our relatively small data set. If one knob showed importance according to more than one method, and for more than one benchmark, we could be more sure that it in fact had an impact. Higher importance values were also (obviously) preferred. We also tried to check where the best and worst points were using parallel coordinates plots, to get a slightly better sense of what was going on and to reduce the risk of choosing knobs where the default value was good and other values were worse.

3.5 Testing the final search space

To test our final search space of 9 knobs we ran an entire Bayesian optimization run with $9 + 1$ initial Latin hypercube samples and $9 \cdot 10$ optimization iterations. We did this three times for each benchmark. We compared with the default configuration and with the default configuration with innodb_dedicated_server on, which automatically sets four of the most important knobs according to some rules (see the values along with the results in section 4.4). We compared also against the best point sampled from the 52 knob search space and also observed the speed of convergence.

Before doing this test we could have again investigated how long warm-up and measurement time that would have been appropriate, since when tuning fewer knobs throughput potentially could stabilize faster. However, instead of doing that we picked the longest warm-up and measurement time that we had decided on for one of the benchmarks for the data collection experiment ($5 + 5$ minutes), since time was now not as much of a constraint and since the warm-up and measurement time investigation itself takes around ten hours of benchmarking.

Chapter 4
Results

4.1 Experimental settings

tool	version
MySQL	8.0.29
Benchbase	commit 23498ac8a1f9fbc8325a1fc5a89ac45756f0b759, Date: May 20 2022
Python	3.10.4
HyperMapper	2.2.10
Java	Java-17-openjdk-amd64
CAVE	commit: afcbecd0b9cb97276625c16a89cb6df141e6f6f2, Date: Mar 26 2021

Table 4.1: Software versions

The benchmarks were performed on MySQL version 8.0.29, installed from the APT (Ubuntu's package management system) package repository, on BenchBase defined by the commit in Table 4.1 with HyperMapper version 2.2.10. Python version 3.10.4 was used for HyperMapper and Benchbase compiled by openjdk 17. The AA, fANOVA and LPI was performed in CAVE version defined by the commit in Table 4.1. It was also slightly modified regarding where files are outputted by our colleague.

benchmark	scalefactor	number of terminals	warmup time	measurement time
TPC-C	650	200	240	120
Twitter	14000	600	240	120
YCSB	50000	400	300	300

Table 4.2: Benchbase setting for the test mentioned in section 3.3. Time is expressed in seconds. What scalefactor correspond to for each benchmark refer to section 2.2.2.

In the test mentioned in section 3.3 (result is presented in section 4.3) and subsequent tests mentioned in section 3.5 (results are presented in section 4.4), the scalefactor, the number of terminals is according to the Table 4.2. The measurement and warm-up for TPC-C and Twitter is changed to 5 minutes (to match with YCSB) for the tests in section 3.5. How the measurement time and warm-up time were decided are described in the section 4.2. How scalefactor and number of terminals was decided are described in the section 3.2.3. The search space used in the experiments can be viewed in the Table B.2 in the appendix. How the knobs and their value ranges were selected is described in the section 3.2.1.

COMPUTER	**m5d2xlarge**
CPU	Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz
RAM	DDR4 1x32GB
Root storage device	EBS Gernal purposse SSD 2 (gp2) (size varies for instance to instance)
Storage device 2	Amazon EC2 NVMe Instance Storage (300 GB)
Region	Stockholm (eu north-1)
OS	Ubuntu 22.04 LTS

Table 4.3: Specification of the AWS machines used for performing the benchmarks.

All benchmarks were performed on m5d2xlarge machines. Additional information can be found in Table 4.3. We chose to store the database on the physically attached SSD storage (storage device 2) and BenchBase results on the root storage device. Depending on the type of benchmark extra EBS was allocated to accommodate all files produced from the benchmark. For all AWS instances, 16 GB of swap memory was created and stored on NVMe SSD. The reason why can be found in section 3.3.1 " Necessary changes for MySQL and AWS instance". All communication to EC2 instances was performed via `SSH`. All benchmarks was run as a background process with the command `Screen`.

4.2 Warm-up and measurement time

Figures 4.1 to 4.9 consist of our warm-up and time investigation for all three benchmarks in our particular use case. From looking at the plots we are able to determine how short warm-up and time we can use when benchmarking, and still rank different configurations correctly relative to each other.

4.2.1 TPC-C

Figure 4.1 shows the throughput second by second of ten different configurations, randomly selected, using the TPC-C benchmark. We see that that behaviour is more erratic for the first few hundred seconds and then stabilizes. Figure 4.2 shows the average throughput of the ten configurations over different measurement windows. For example, where "time" is 1800 and "warm-up" is 900, the values are the average throughputs between second 900 and 2700 for the ten configurations. This point we consider to be the "correct" measurement of throughputs given the configuration. We want to pick a warm-up and measurement time

shorter than that but that still ranks the ten configurations correctly relative to each other. For low warm-up and measurement times we see that configurations rank quite differently, and are very sensitive to small changes. We want to choose a point somewhere in between, and have to decide on the shortest total time with a level of noise we can tolerate.

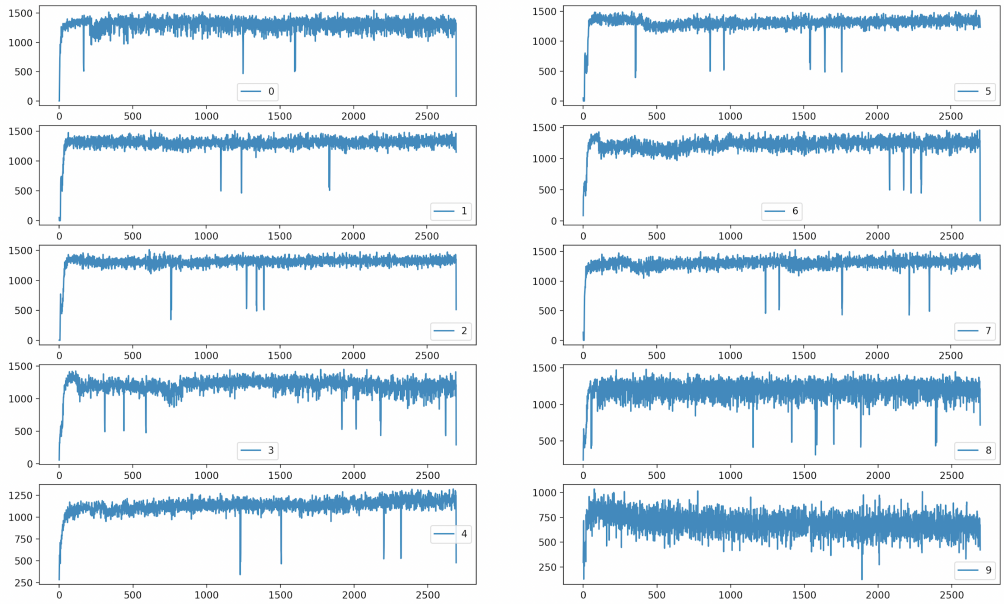

Figure 4.1: Plots of throughput during 2700 seconds or 45 minutes for the TPC-C benchmark, for 10 random points in our 52 knob search space. The points can found in Table B.3 in appendix.

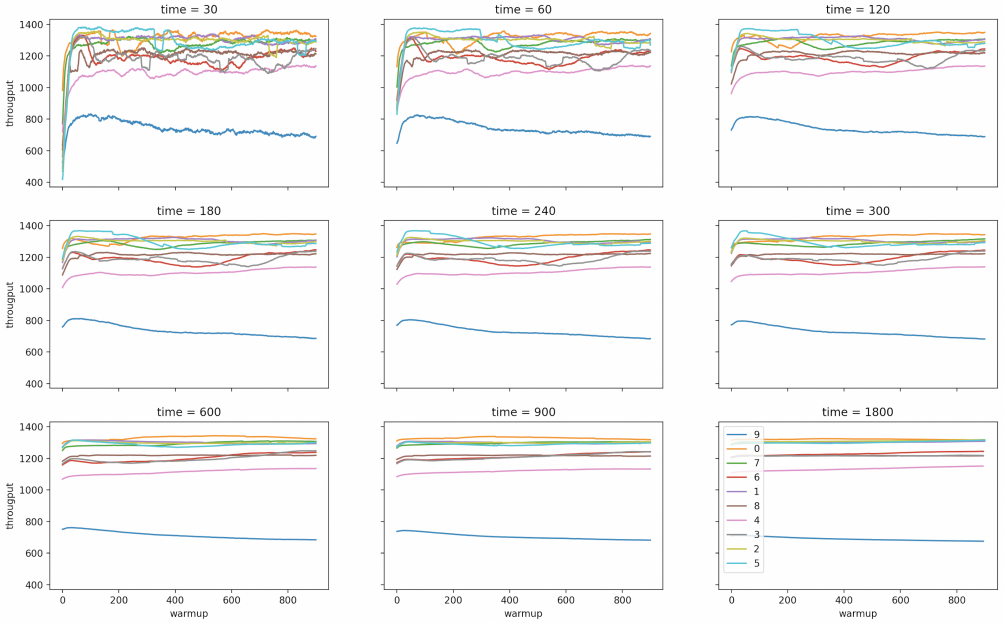

Figure 4.2: Plots of throughput for different values of warm-up and (measurement) time for the TPC-C benchmark, here with fixed times in each plot and warm-up changing along the x-axis.

In Figure 4.3 we have instead fixed the warm-ups and are changing time along the x-axis. We see that light blue is ranked too highly, and orange too low, until approximately 240 second warm-up and 240 second time. We decide this is to slow and to accept some configurations being slightly under- or overvalued. We decide to go with 180 second warm-up and 120 second time, since several of the configurations seem to need a three minute warm-up to stabilize and since from looking at the warmup = 180 plot there is not much gain from measuring for longer than 120 seconds, unless it would be several minutes longer. The most important part seems to be to remove the transient effects in the beginning, and all configurations except light blue and orange seem to have come close their final throughput values around this point. With a three minute warm-up and two minute time plus around four minutes in between every iteration we can collect around 1000 samples in a week.

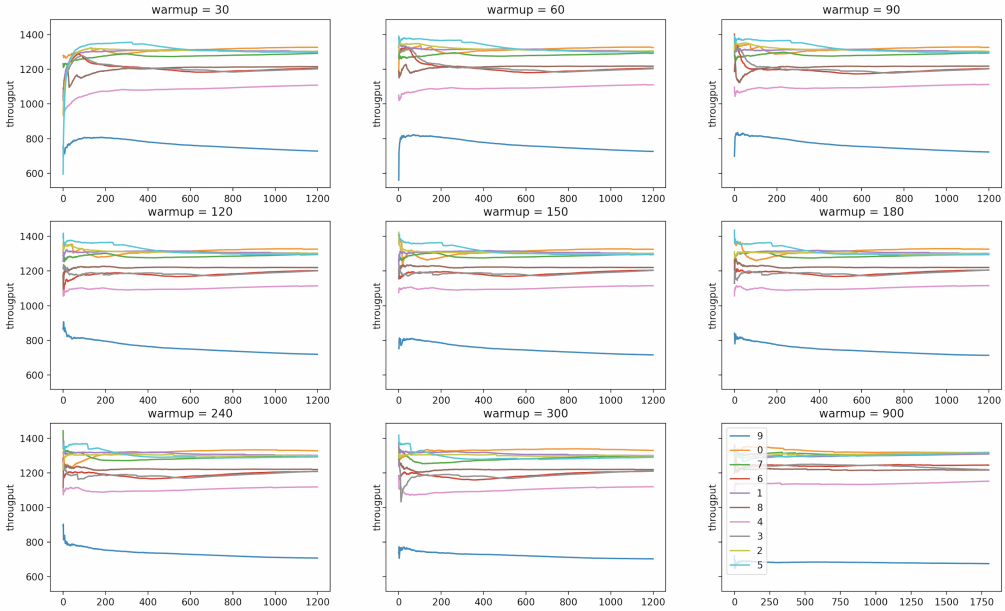

Figure 4.3: Plots of throughput for different values of warm-up and (measurement) time for the TPC-C benchmark, here with fixed warm-ups in each plot and time changing along the x-axis. The points can found in Table B.3 in appendix.

4.2.2 Twitter

In Figures 4.4 to 4.6 we see the same plots for Twitter. The behaviour of the configurations are wildly different, but they stabilize pretty well after a few minutes. We conclude that the same 180 second warm-up and 120 second time is similar enough to the 900+1800 point. Before this several of the configurations would be overvalued.

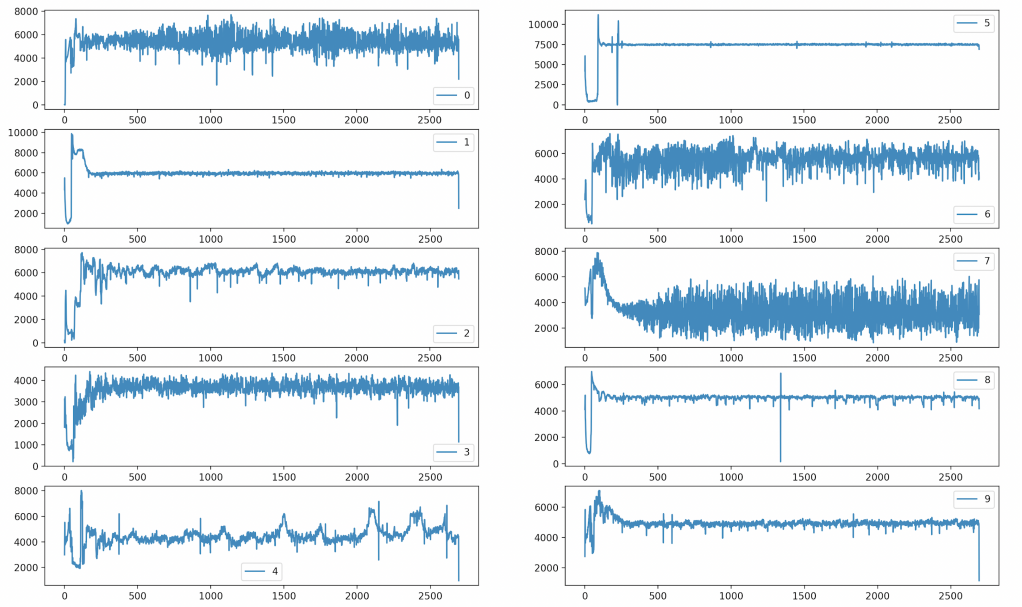

Figure 4.4: Plots of throughput during 2700 seconds or 45 minutes for the Twitter benchmark, for 10 random points in our 52 knob search space. The points can found in Table B.3 in appendix.

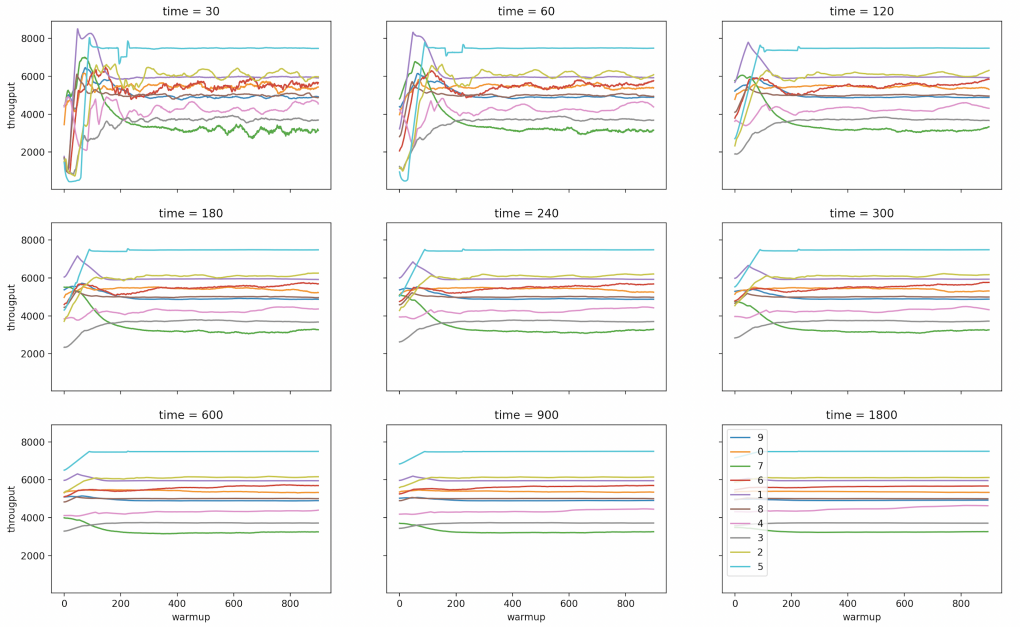

Figure 4.5: Plots of throughput for different values of warm-up and (measurement) time for the Twitter benchmark, here with fixed times in each plot and warm-up changing along the x-axis.

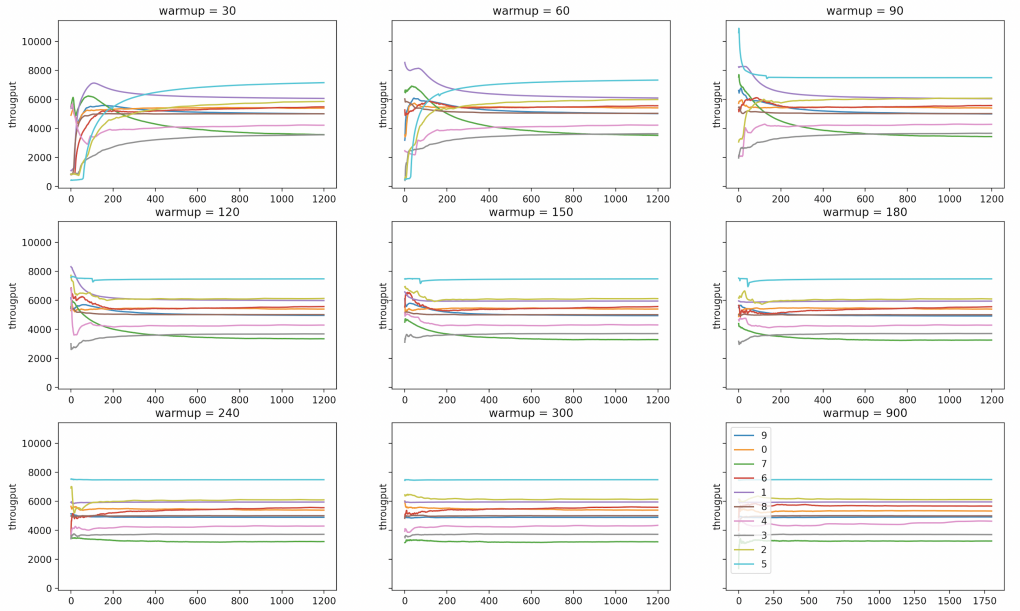

Figure 4.6: Plots of throughput for different values of warm-up and (measurement) time for the Twitter benchmark, here with fixed warm-ups in each plot and time changing along the x-axis.

4.2.3 YCSB

From looking at Figure 4.7 to Figure 4.9 we see that YCSB seem to require longer warm-up and time than the previous two benchmarks. The initial transient effects seem to be going on longer. At the 180 second warm-up and 120 second time point configurations assume the wrong order, with for example gray coming in at number seven instead of ten, blue not clearly beating purple, and pink being number ten instead of eight. We decided that we need a longer warm-up and measurement time, and that beyond a 300 second warm-up and 300 second measurement time there is not much to gain unless we measure for much longer. So we decided to set both warm-up and measurement to 300 seconds for this benchmark. The trade off was that we had to wait longer for the results to come in.

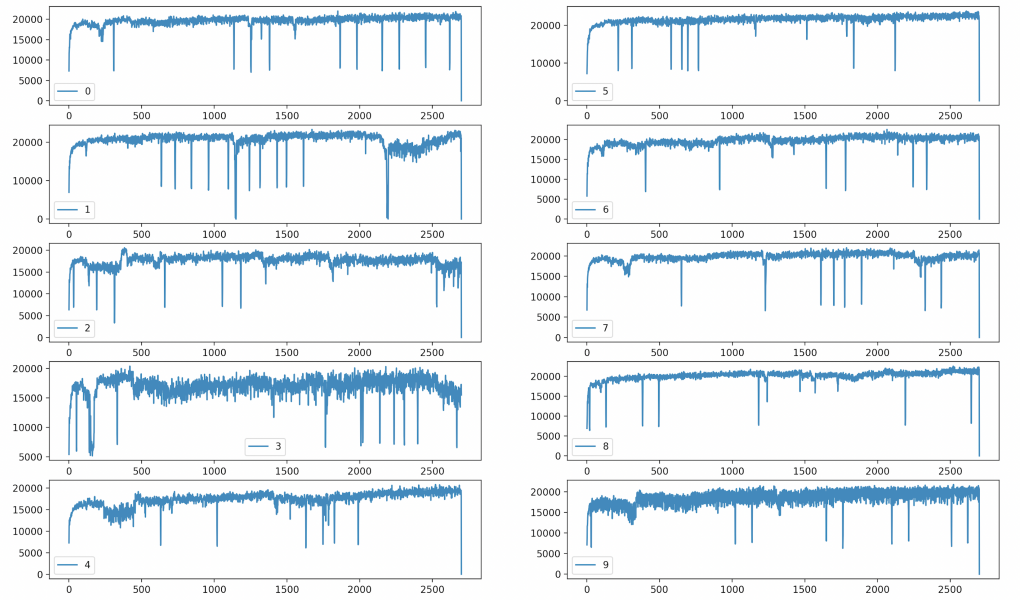

Figure 4.7: Plots of throughput during 2700 seconds or 45 minutes for the YCSB benchmark, for 10 random points in our 52 knob search space.

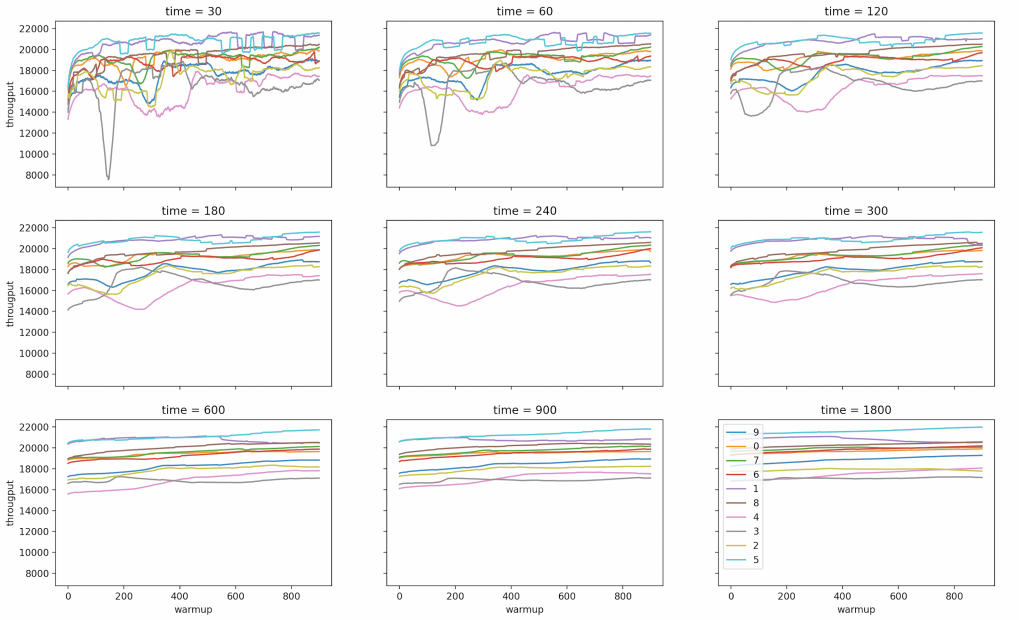

Figure 4.8: Plots of throughput for different values of warm-up and (measurement) time for the YCSB benchmark, here with fixed times in each plot and warm-up changing along the x-axis.

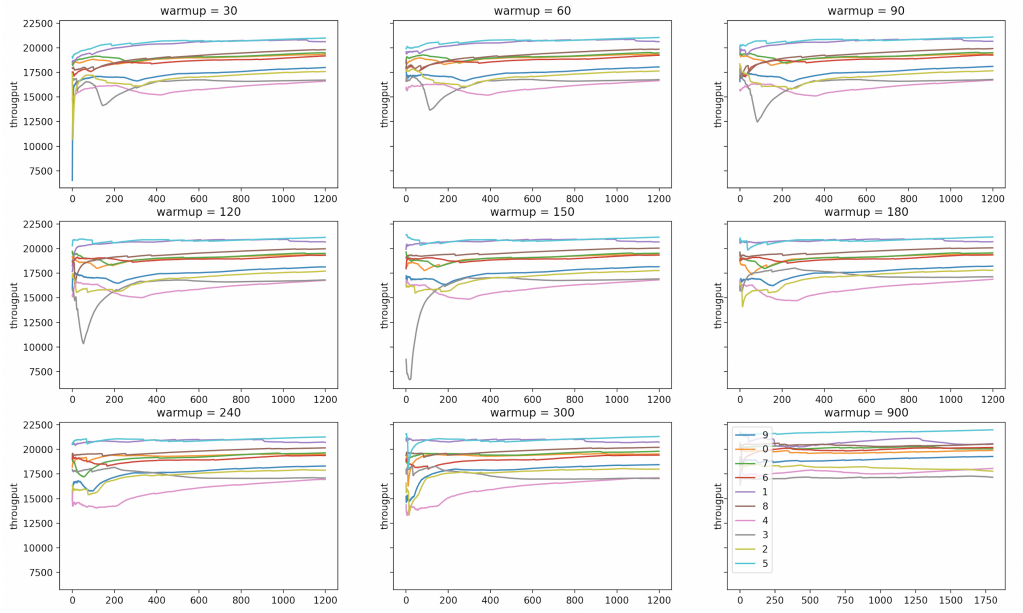

Figure 4.9: Plots of throughput for different values of warmup and (measurement) time for the YCSB benchmark, here with fixed warm-ups in each plot and time changing along the x-axis.

4.3 Feature importance

Here we present the figures relating to our feature importance investigation, and outline the reasoning behind choosing the knobs we did. Our analysis of individual knobs is in the next subsection. In Figure 4.10 we see the importance weights according to the four methods for the TPC-C benchmark. We have zoomed in to better see the other weights beyond innodb_buffer_pool_size, which has a sum of around 1.5 as seen in Figure A.2 in the appendix. In Figure 4.11 we show only the fANOVA weights, with the top weights of combinations of two knobs included.

In Figure 4.12 we see a parallel coordinates plot with all of the knobs we decided to investigate further. This meant leaving innodb_buffer_pool_size and innodb_flush_method out since they where so clearly impactful for all three benchmarks. Every line in the plot is one knob value, and to the far right is the throughput, split up into different categories for visualisation purposes. Darker green mean higher throughput, the default is highlighted and red means worse than default. This plot gives some indication of which knob values that produced the best and the worst throughputs. Keep in mind that only half of the points are randomly spread out across the search space, on a Latin hypercube, and that the other half of the points is the optimizer choosing points based on expected improvement. So a large part of the fact that some knob values contributed to high throughputs more often than others is mainly because of the optimizer decided to investigate those areas of the search space more.

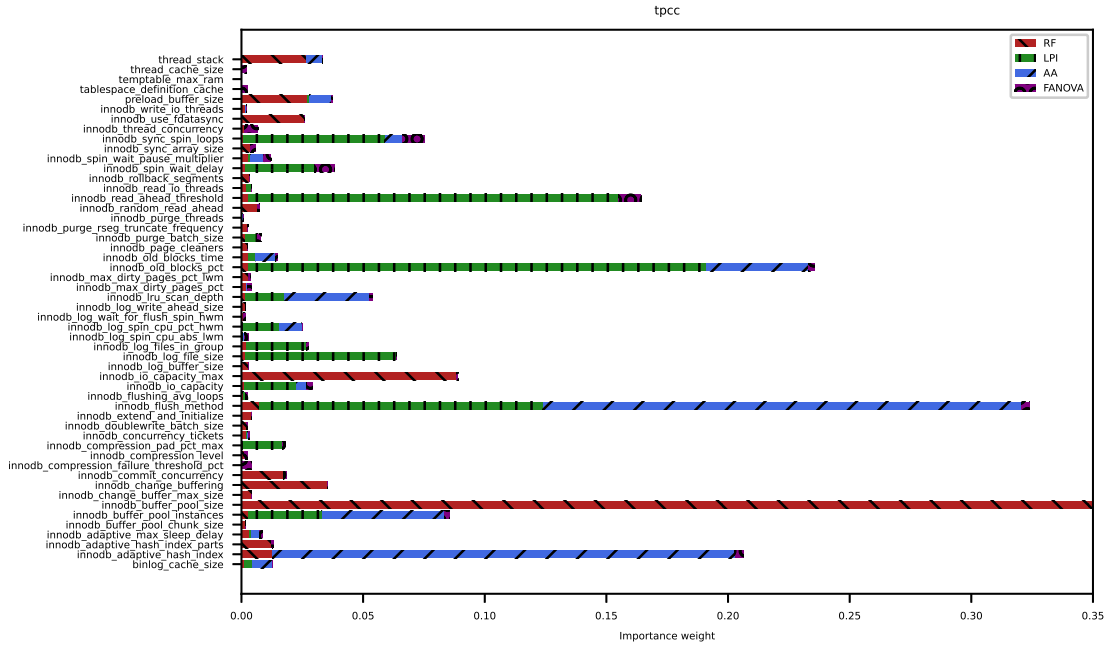

Figure 4.10: Knob importance weight for TPC-C according to RF, LPI, AA, fANOVA. This figure is identical to figure A.2 but zoomed in.

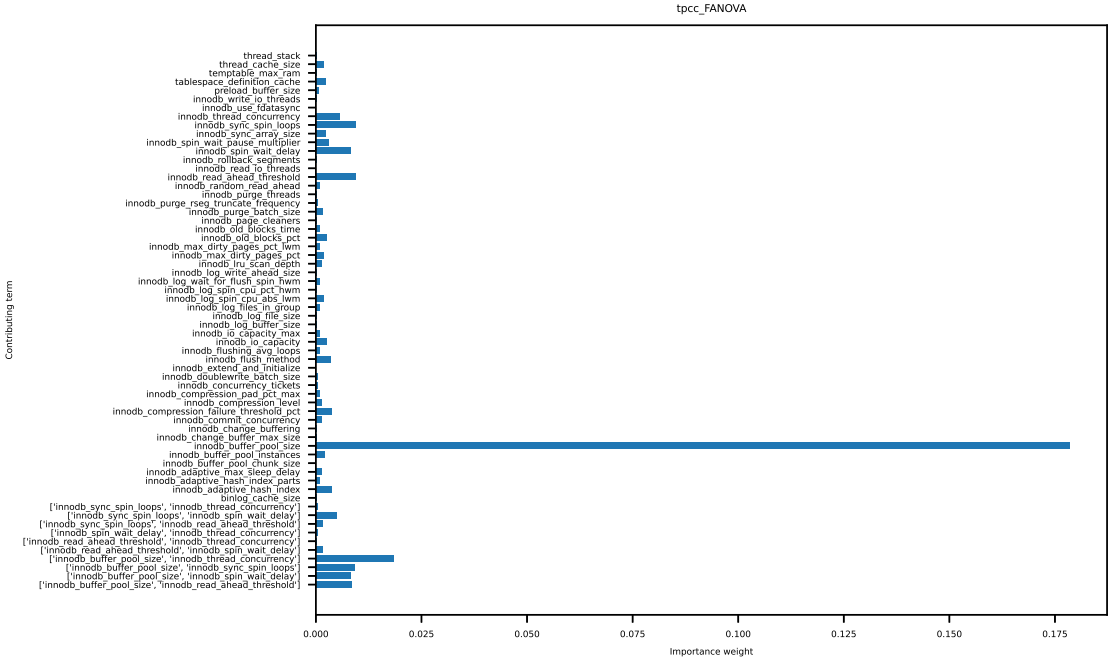

Figure 4.11: fANOVA with first and seconder interactions between knobs for TPC-C benchmark.

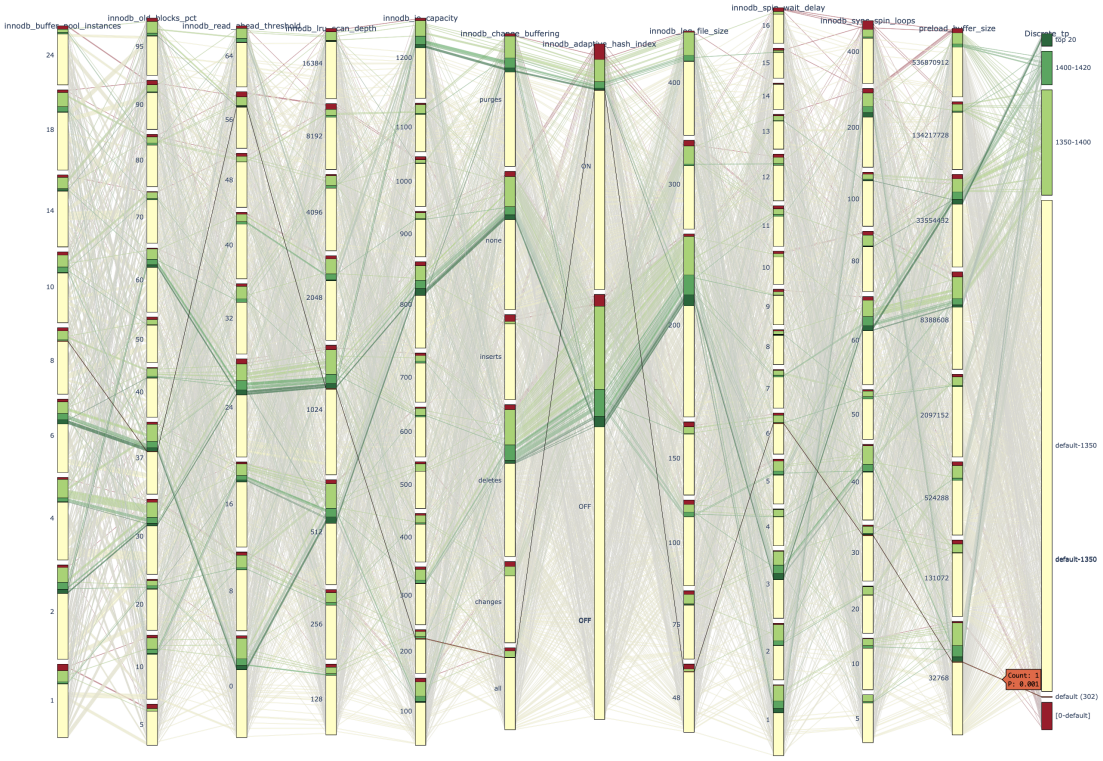

Figure 4.12: Parallell coordinates plots of knobs we decided to investigate further for the TPC-C benchmark, giving some indication of which combinations of values resulted in which throughput.

Figures 4.13 to 4.15 show the same set of plots as for TPC-C above, but now for Twitter. The parallel coordinates plot includes different knobs. They are the knobs that we decided to investigate further with respect to their impact on performance on Twitter. We analyse the promising knobs one by one in the next section.

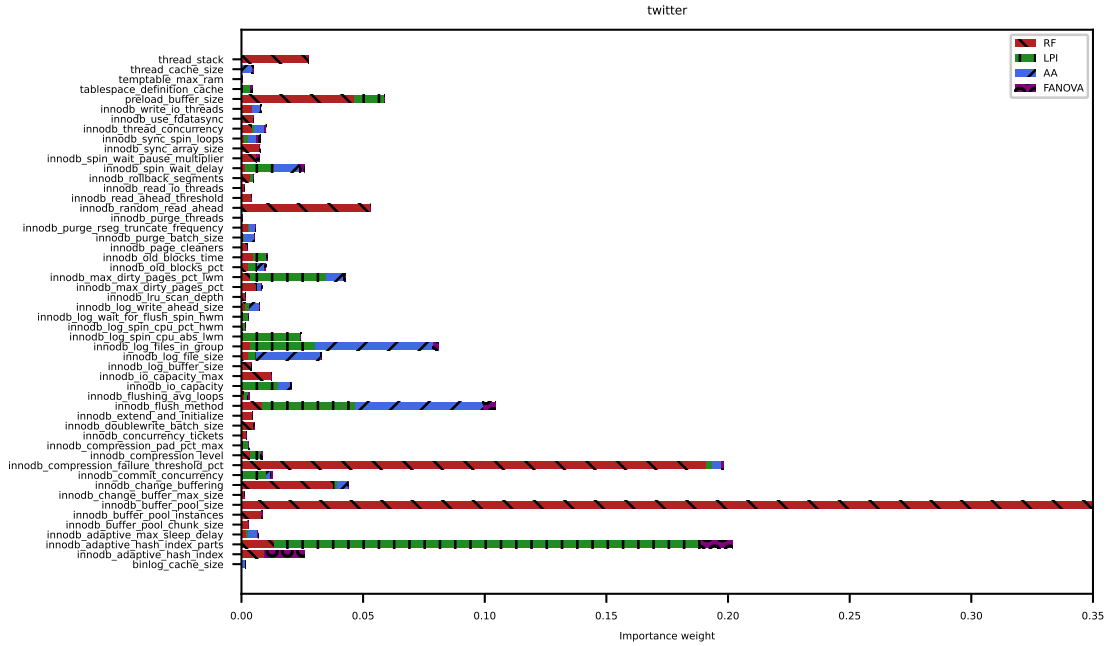

Figure 4.13: Knob importance weight for Twitter according to RF, LPI, AA, fANOVA. This figure is identical to figure A.3 but zoomed in.

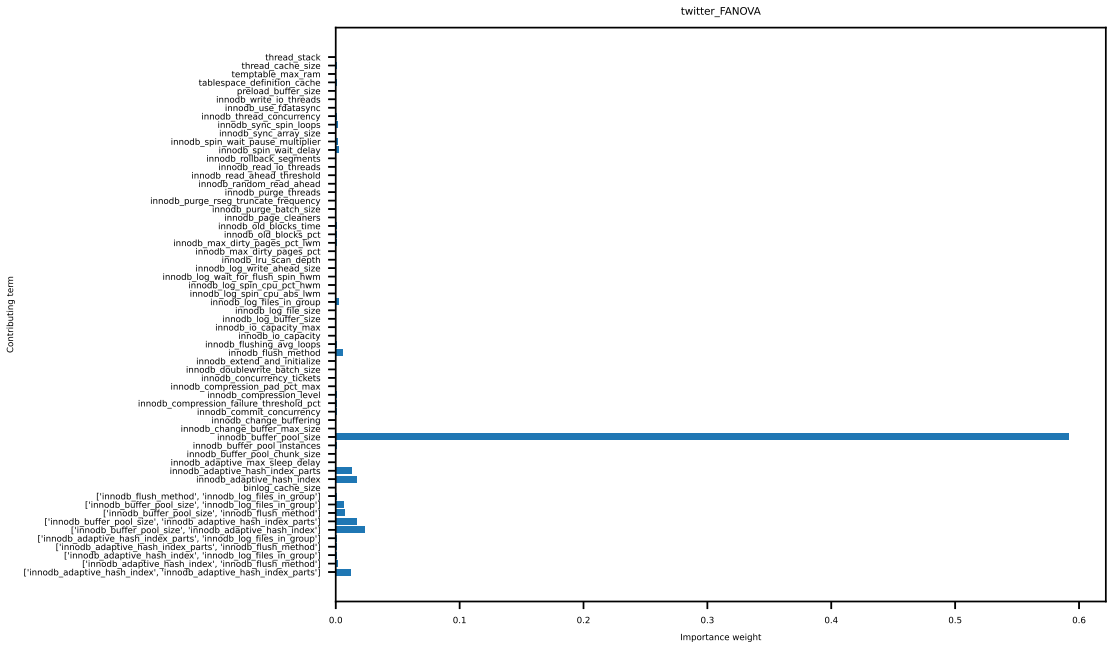

Figure 4.14: fANOVA with first and seconder interactions between knobs for Twitter benchmark.

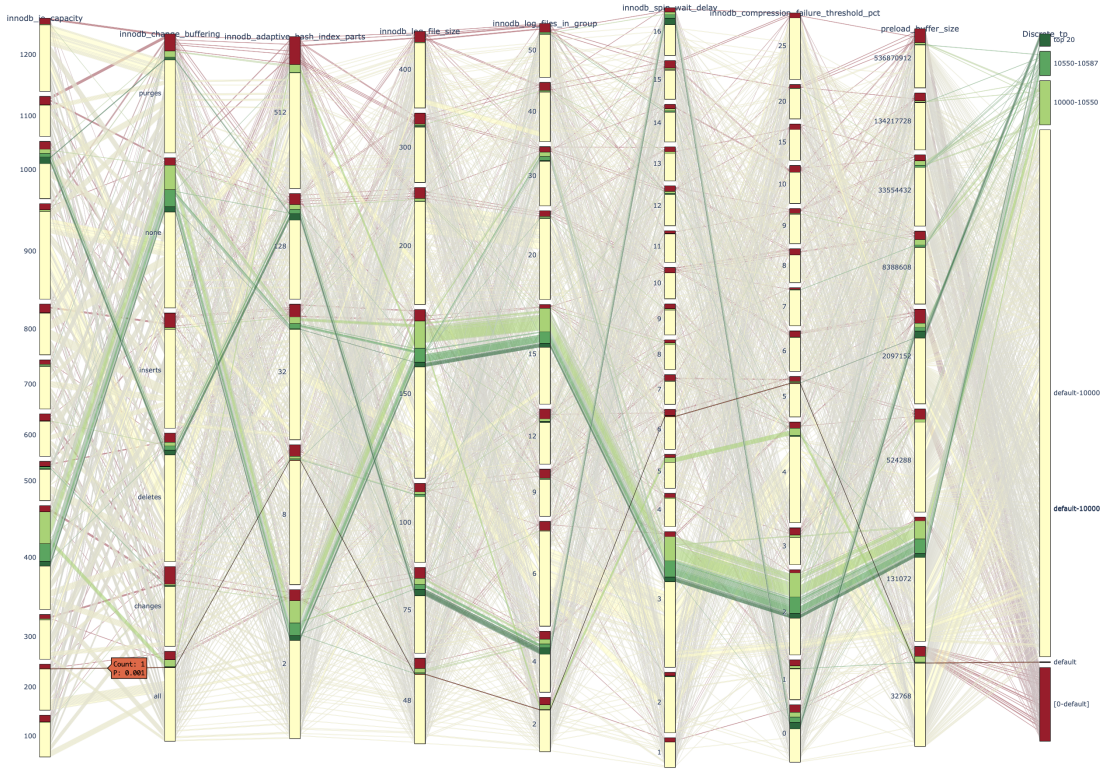

Figure 4.15: Parallell coordinates plots of knobs we decided to investigate further for the Twitter benchmark, giving some indication of which combinations of values resulted in which throughput.

The same three plot are shown in Figures 4.13 to 4.15, now for YCSB. The parallel coordinates plot again contains different knobs.

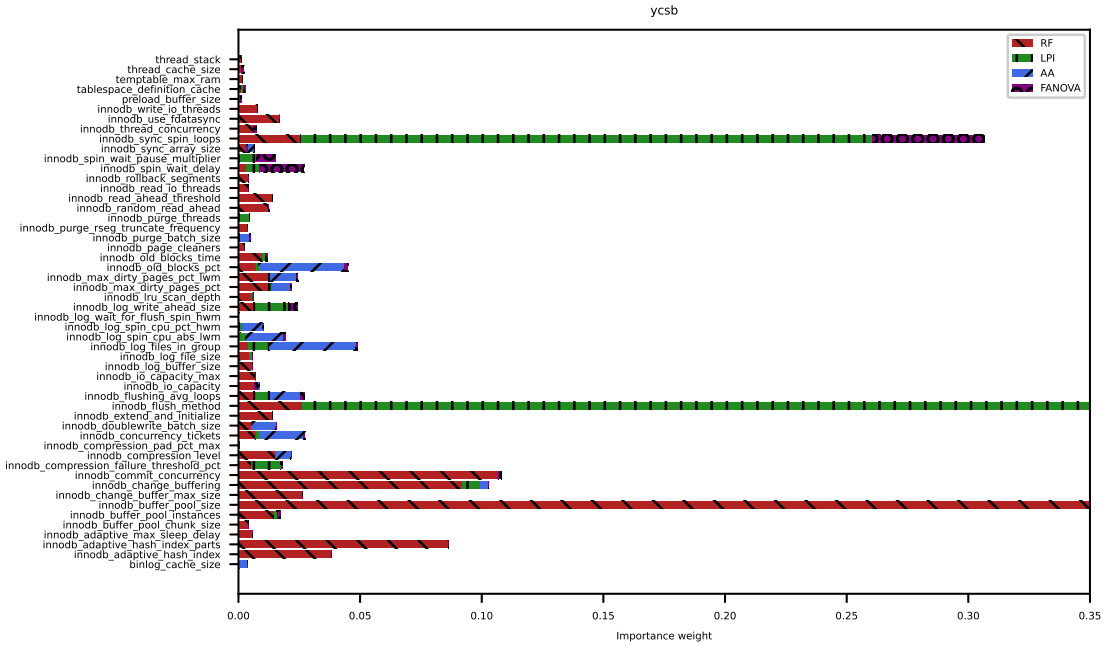

Figure 4.16: Knob importance weight for YSCB according to RF, LPI, AA, fANOVA. This figure is identical to figure A.1 but zoomed in.

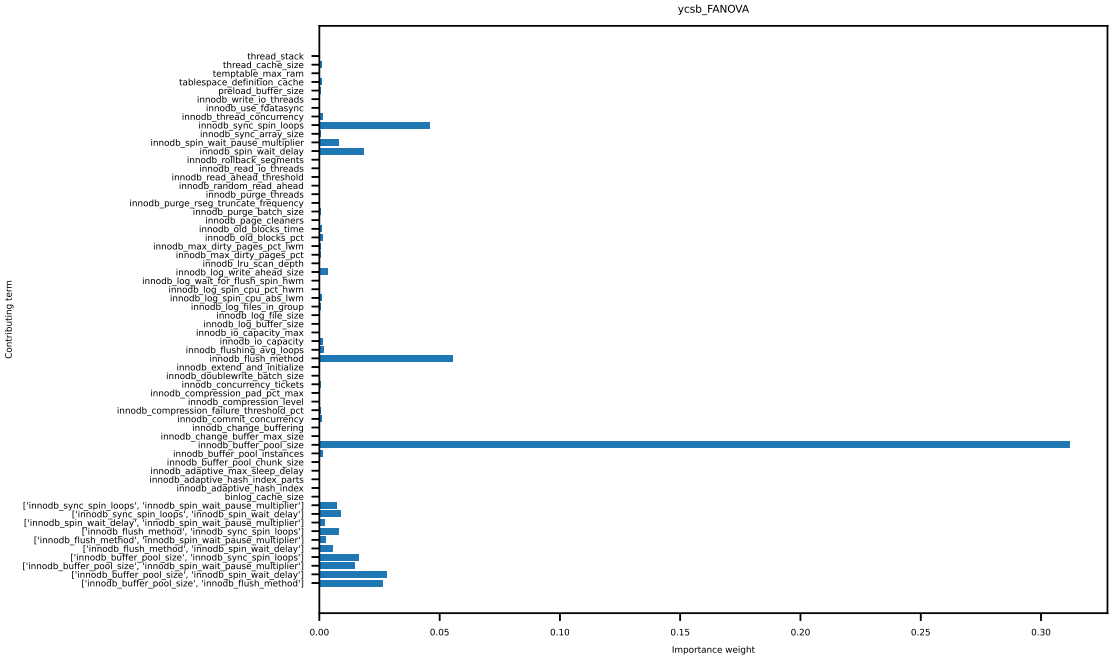

Figure 4.17: fANOVA with first and seconder interactions between knobs for YCSB benchmark.

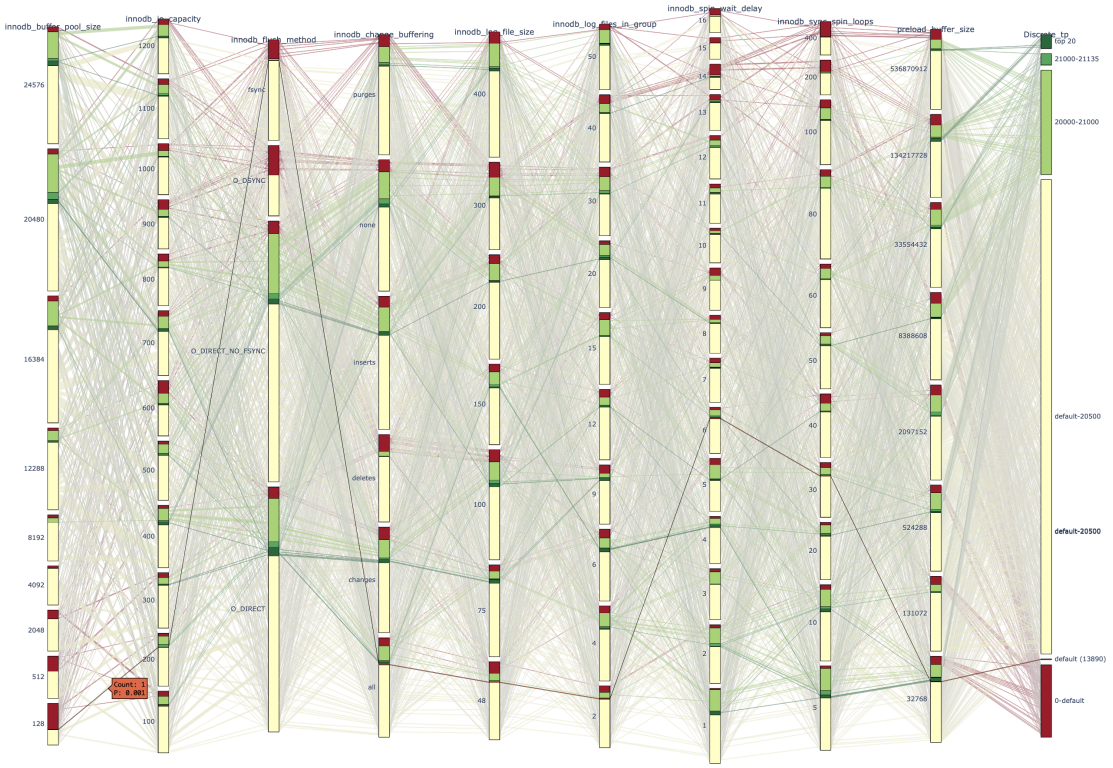

Figure 4.18: Parallell coordinates plots of knobs we decided to investigate further for the YCSB benchmark, giving some indication of which combinations of values resulted in which throughput.

4.3.1 The most important knobs

We here go through the knobs that show some clear indication of being important for one or more benchmarks, with one or more feature importance methods. We consider a knob with good importance features values on all four feature importance methods more important than a knob with higher cumulative sum but concentrated on only one feature importance method. In total we look deeper into 16 knobs, and also mention five others that possibly could have had an impact but that would not make it into our final search space, all of which presented in the Table 4.4.

Part of final search space	Honourable mentions	The remaining of the 16 knobs
preload_buffer_size	thread_stack	innodb_read_ahead_threshold
innodb_sync_spin_loops	thread_cache_size	innodb_old_blocks_pct
innodb_spin_wait_delay	innodb_random_read_ahead	innodb_lru_scan_depth
innodb_log_files_in_group	innodb_old_blocks_time	innodb_compression_failure_threshold_pct
innodb_log_file_size	innodb_commit_concurrency	innodb_buffer_pool_instances
innodb_io_capacity		innodb_adaptive_hash_index_parts
innodb_flush_method		innodb_adaptive_hash_index
innodb_change_buffering		
innodb_buffer_pool_size		

Table 4.4: The left column represents the parameters that are part of the 16 knobs that we considered interesting candidates and later also included in the final search space. The middle column represents the 5 knobs that had lower impact overall but still promising. The right column represents the remaining 16 knobs that ended up filtered out due to not enough impact.

- The knob preload_buffer_size is not exceptionally high anywhere but it has some moderate contribution from several feature importance methods in all the benchmarks. In Figure 4.15 we notice that the optimizer seemed to prefer slightly higher values than the low default value, but overall we don't notice much structure in our parallel coordinates plots.

- The knob innodb_sync_spin_loops seems to be the third most important knob for YCSB, with three of the feature importance methods indicating that it is of some importance. For TPC-C three of the methods produce above zero values, although the sum is not as high. In the parallell coordinates plot for TPC-C 4.12 we see that higher values than default seems to be preferred, and judging from Figure 4.18 we see that lower values are preferred for YCSB. This knob has to do with the number of times threads waits for a mutex lock to be freed before the thread is suspended. We believe the difference between TPC-C and YCSB may depend on the complexity of the queries, where TPC-C has more complex queries that should get more time to complete. There is limited information about this knob on the internet and the relationship between it and the other knobs that has to do with mutex locks and threads going to sleep is unclear, but we believe that after this number of times the thread has waited for a mutex lock the query is aborted, and the user needs to try again. And so in practice the user may wants to set a higher value on this knob than what is optimal for throughput to make sure that more queries go through. That said, at some points queries need to give up, and so optimizing this knob for throughput might not be so bad. Either way we conclude that this knob probably has some impact on throughput.

- The knob innodb_spin_wait_delay has a sum of the four methods above 0.02 for all benchmarks, with contributions from three of the four methods. It contributes more to TPC-C and YCSB than Twitter as in the case of innodb_sync_spin_loops. All three benchmarks seem to prefer smaller values on this knob than the default value six. We conclude that this knob probably has some small impact on throughput.

- The knob innodb_read_ahead_threshold is given a lot of importance by the LPI method

in TPC-C, and less than **1%** by fANOVA, but not much anywhere else. In Figure 4.12 we notice good and bad points all over. The values 0, 16, 20 and the default 56 all contain top 20 points. Here 0 actually mean that read ahead is turned off, which is more similar to higher values such as 64 than low ones. Higher values means that read ahead is only triggered when most of a page has already been read, which means that it is triggered less often. In any case we don't notice enough evidence that optimizing this knob impacts throughput in any significant way.

- In Figure A.2 we notice that innodb_old_blocks_pct is considered to be very important for TPC-C by LPI and AA. It importance weight for LPI and AA is approximately 16 % and 10 %. From Figure 4.12 we can observe the default value is quite good, it contains some of the top 20 points. Other good values are 60 % and 30 %. Many of top 20 point prefer a specific combination of innodb_old_blocks_pct and innodb_buffer_instances: innodb_buffer_instances values 6.2 combined with innodb_old_blocks_pct values 37.30 respectively. The top 20 points seem to be in the 30 to 60 range. Default might be good enough, and whats important is to stay away from the two extremes. Since we did not notice any importance in twitter and YCSB either, we choose to not include this knob in the final search space.

- The knob innodb_lru_scan_depth is mostly important for TPC-C, a bit for YCSB according to the AA method and not at all for Twitter. In Figure 4.12 we notice that most of the top points for TPC-C uses the default value or slightly lower, and in Figure 4.18 we notice that the top points are spread out everywhere, including the default value. Beyond that we can't conclude much from those plots. We stick to the default value and don't include in the final search space.

- We notice that innodb_log_files_in_group might have some importance for all three benchmarks, mainly Twitter and YCSB. Especially the AA method in Twitter and YCSB might have detected something. In Figure 4.15 we notice quite clearly that most of the better points in Twitter use one of the values 4 or 15, both larger than the default 2. Also in Figure 4.18 we believe that values other than the default might lead to higher throughputs, although less clearly than Twitter.

- The knob innodb_log_file_size has some importance according to LPI for TPC-C and some according to AA for Twitter. To get a better picture what is going on we investigate the parallel coordinates plots. We notice that for TPC-C there might be some preference for values such as 200, higher than the default 48 and for Twitter for the values 75 and 150. If there is space this knob could be included in the final search.

- For the knob innodb_io_capacity we notice some small signs of importance according to RF in TPC-C and Twitter, complemented by AA, suggesting that something small might be going on. In the parallel coordinates plots we notice that most of the very best points use some value higher than the default 200 for both TPC-C (800 and 1200) and Twitter (400 and 1000). Why it has been given such low importance overall is bit of a mystery. This knob is responsible for the IOPS rate on background tasks and we believe tuning this knob could be important. Possibly tuning it doesn't start to matter before the really high throughputs, making it invisible to the feature importance methods.

AA could have catched it though. If there is space this knob might be included in the final search space.

- The knob innodb_flush_method is clearly important for all benchmarks and it is included without question.
- The knob innodb_compression_failure_threshold_pct is given a lot of importance according to RF for Twitter, possibly supported by less than 1% from the other methods. For YCSB, LPI gives it about 2% of the importance. In Figure 4.15 we notice that for Twitter the optimizer has choosen to explore more around the smaller values 0-4 than the default 5 and higher values, and it is also among those small numbers that the best points where found. It might just be that other knobs contributed to these higher throughputs, making the Bayesian optimization algorithm stick around in this area for no particular reason. For YCSB the best points where found at 5 and 15. We just stick to the default and don't include it in the final search space.
- For innodb_change_buffering the situation is similar to the above knob, with weight being given by RF, but now it is for all three benchmarks, and for YCSB it is supported by some importance from the LPI method. Looking at the parallel coordinates plots we notice that more good points seem to use values other than the default value "all" in all the benchmarks. We include it in the final search space.
- The knob innodb_buffer_pool_size has high numbers in several feature importance methods for all three benchmarks and is therefore without hesitation included in the final search space.
- The knob innodb_buffer_pool_instances is given significant importance by AA and LPI for TPC-C, insignificant amount by all importance methods for Twitter and just around 2% by RF for YCSB. From the parallel coordinates plot for TPC-C, Figure 4.12, we don't get a lot of information since good and bad points are spread out across all values. Overall the sum of the feature importance methods on TPC-C is not high enough and we also don't notice anything in the other benchmarks.
- The knob innodb_adaptive_hash_index_parts seems important for Twitter since it is more than 15% of LPI, supported by significant chunks from RF and fANOVA. On YCSB it is given almost 10% of the weight from RF. In Figure 4.15 the best points are using the values 2 and 128 for Twitter. This large difference is slightly strange, unless there are dependencies with other knobs that would explain why 2 and 128 both sometimes are optimal. For YCSB, Figure 4.18 both good and points seem to evenly spread out. Further investigation found that innodb_adaptive_hash_index was mostly chosen to be turned off by the optimizer (ie. by setting innodb_adaptive_hash_index to false), which definitely makes this knob have zero impact when that is true. We therefore guess that the significant importance given by LPI is mostly due to randomness. If adaptive hash indexing was mostly turned on then we would have more strongly believed that this knob was important.
- The knob innodb_adaptive_hash_index is given a lot of importance by the AA method in TPC-C and only some by RF and fANOVA for Twitter and some from RF for YCSB. The optimizer keeps trying out both OFF and ON for TPC-C, and both seem to have

both good and bad values. We can't determine that this is an import enough knob to include it.

Other honorable mentions would be thread_stack, thread_cache_size, innodb_random_read_ahead, innodb_old_blocks_time and innodb_commit_concurrency but these did not quite have high enough importance values to make it into our final search space.

We conclude that preload_buffer_size,innodb_sync_spin_loops, innodb_spin_wait_delay,innodb_log_files_in_group, innodb_log_file_size, innodb_io_capacity, innodb_flush_method, innodb_change_buffering and innodb_buffer_pool_size seem to be the most likely top 9 important knobs and are therefore included in our final search space for testing.

These knobs are followed by innodb_read_ahead_threshold, innodb_old_blocks_pct, innodb_lru_scan_depth, innodb_compression_failure_threshold_pct, innodb_buffer_pool_instances, innodb_adaptive_hash_index_parts and innodb_adaptive_hash_index which all very well could have a great impact that we just did not notice enough of in our experiments. The knobs thread_stack, thread_cache_size, innodb_random_read_ahead, innodb_old_blocks_time and innodb_commit_concurrency could also be impactful, but with lower probability.

4.4 The final search space

Here we present the results of optimizing over our final search space, containing the 9 most important knobs. The value ranges are the same as when we did the feature importance, and so might be wider than one would want them in a real world setting. We ran each optimization three times for all three benchmarks, with 10 design of experiment iterations and 90 optimization iterations. We also look at parallel categories plots again and discuss whether or not the nine seems to actually have been important in the final search space.

4.4.1 TPC-C

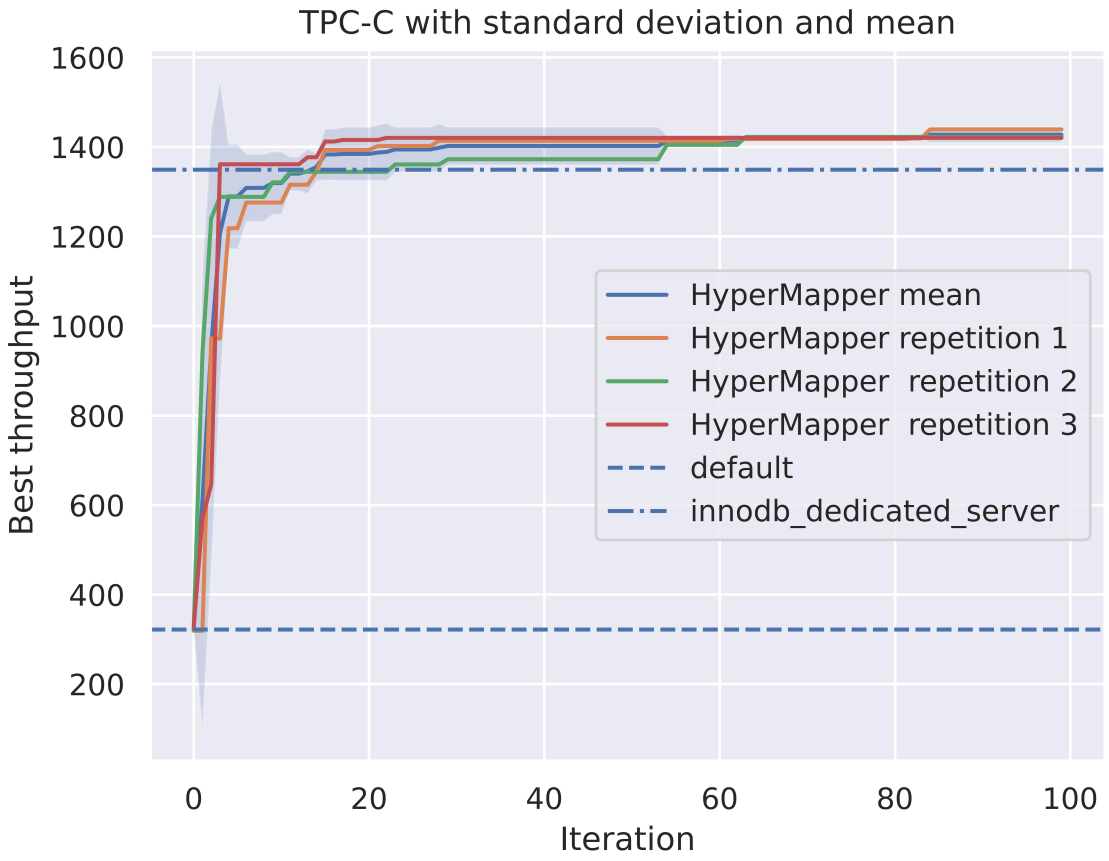

Figure 4.19: Throughput over best configuration found over time for all 3 repetitions with default configuration and innodb_dedicated_server=ON. Benchmark : TPC-C.

Figure 4.19 shows the convergence for TPC-C, as well as the default and also default with innodb_dedicated_server turned on as baseline reference points. We see that after around 60 iterations all three repetitions seems to have converged. From Table 4.5 we see that optimal throughput in the final search space are close to best point found in the larger 52 knob search space. The optimal throughput from the final search space is between 98 and 99 % of the optimal throughput in the larger search space, while innodb_dedicated_server=ON is around 93 %.

Table 4.5 shows the knob values and throughputs of the best point for the three repetitions. We see that only innodb_flush_method, innodb_change_buffering and innodb_buffer_pool_size are the same across all three runs. Figure 4.20 shows a parallel categories plot using all 300 points from the three optimization repetitions. The throughputs are to the far right, where darker green means higher throughput. The default configuration is highlighted in red.

We can see that for some knobs the best configurations use values significantly different from default, indicating that the knob was important to include in the search space. For others the best configurations are spread out across all the possible values, meaning that

the value put on the knob doesn't seem to matter very much. For innodb_flush_method, innodb_change_buffering and innodb_buffer_pool_size there seems to be a clearly prefered value, and beyond that it seems that values higher than default for preload_buffer_size is preferred. The optimal points for this point as seen in the table are all using one of three highest values, much larger than default. Both innodb_log_files_in_group and innodb_log_file_size seem to create higher throughputs when set to something significantly different from their defaults. The three knobs innodb_spin_wait_delay, innodb_sync_spin_loops and innodb_io_capacity however seem less important since good configurations uses values spread out across many values, both high and low. It could be that some combinations of innodb_spin_wait_delay and innodb_sync_spin_loops are better than others, maybe high innodb_spin_wait_delay combined with low innodb_sync_spin_loops or low innodb_spin_wait_delay combined with medium innodb_sync_spin_loops.

From	default	rep 1	rep 2	rep 3	innodb_dedicated_server=ON	original
preload_buffer_size	32768	536870912	134217728	33554432	-	536870912
innodb_sync_spin_loops	30	400	5	10	-	200
innodb_spin_wait_delay	6	1	8	8	-	1
innodb_log_files_in_group	2	50	12	50	18	30
innodb_log_file_size	48M	300M	200M	300M	1024M	200M
innodb_io_capacity	200	1100	400	600	-	800
innodb_flush_method	fsync	O_DIRECT	O_DIRECT	O_DIRECT	O_DIRECT_NO_FSYNC	O_DIRECT_NO_FSYNC
innodb_change_buffering	all	purges	purges	purges	-	purges
innodb_buffer_pool_size	128M	24576M	24576M	24576M	24576M	20480M
Throughput	322	1439	1422	1420	1349	1447

Table 4.5: Benchmark: TPC-C. The table contains the optimal configuration from each repetition of the final search space test, optimal configuration from the original search space test, the default configuration and configuration from when innodb_dedicated_server is enabled. The remaining of the 52 knobs are set to default values except for the original optimal configuration which can be viewed in Table B.1 in appendix. A hyphen in innodb_dedicated_server=ON column indicate that the knob is not affected by innodb_dedicated_server and have the same value as the default column. K = 1024^1, M = 1024^2, G = 1024^3.

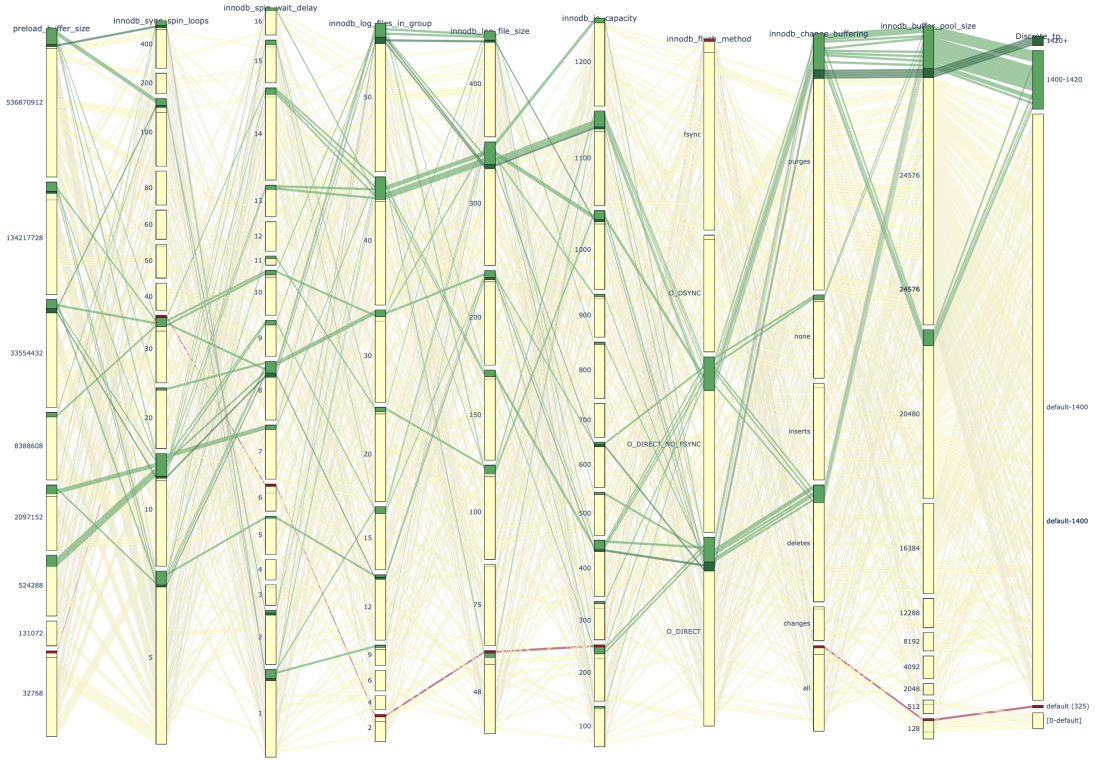

Figure 4.20: Parallel categories plot of all 300 different configurations and their resulting throughputs from the three optimization runs on TPC-C. The default configuration is highlighted in red. The knobs innodb_buffer_pool_size and innodb_log_file_size are expressed in the unit of M = 1024^2.

4.4.2 Twitter

In Figure 4.21 we see that HyperMapper optimizations converge before the 20:th iteration. This seems to be because only some knobs matter and beyond that there is not much optimization to be done in our search space.

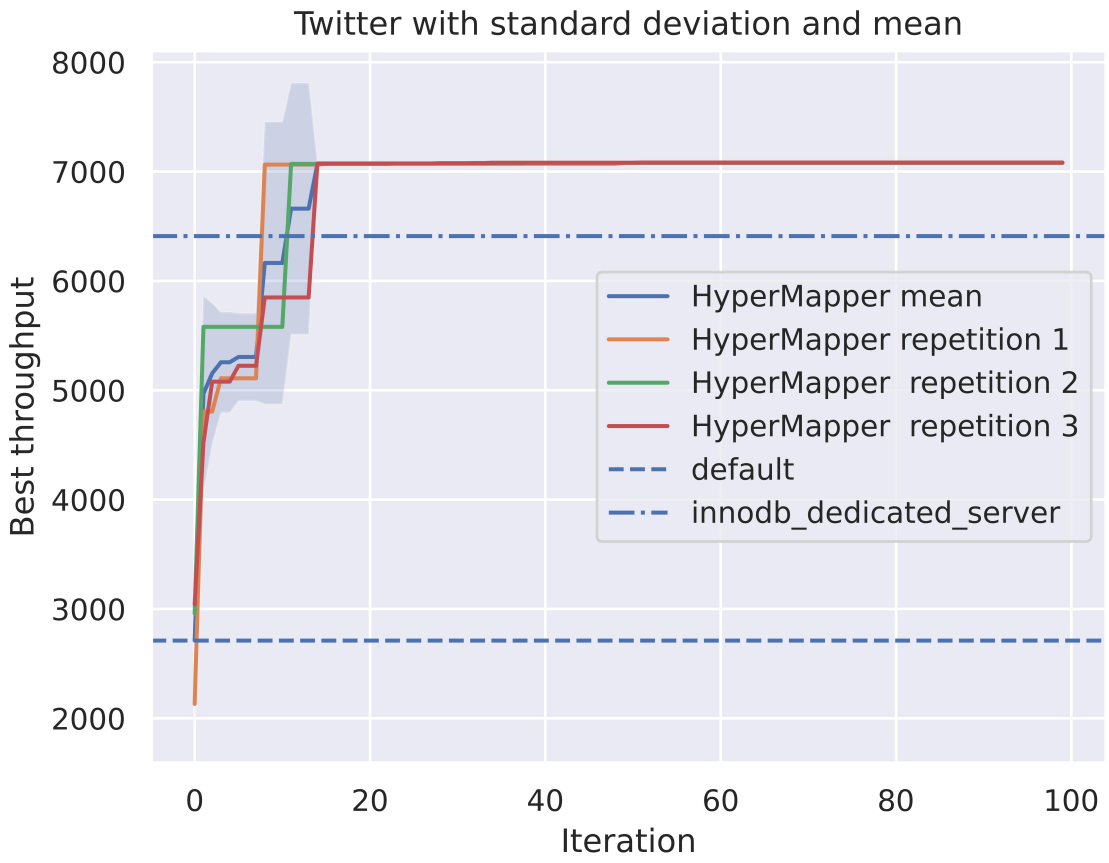

Figure 4.21: Throughput over best configuration found over time for all 3 repetitions with default configuration and innodb_dedicated_server=ON. Benchmark : Twitter.

In Table 4.6 we see that the best configuration found in the original 52 knob-search space has a considerably higher throughput than the optimum found in the final search space. This is probably because there are knobs with high impact on Twitter that based on our experiments did not quite make it into the final search space. Further testing would need to be done to see if any of the "honorable mentions" could increase the throughput up to above 10,000. This could be for example the knob innodb_compression_failure_threshold_pct that had high importance according to the random forest feature importance method for twitter but was left out because it wasn't supported enough by the other methods and benchmarks. The optimal values are here between 66 and 67% of the best point found in the larger search space, while innodb_dedicated_server=ON got around 60%.

From Figure 4.22 we can see that for the best points innodb_buffer_pool_size and innodb_flush_methods are set to 20480 MB and O_DIRECT. We see also that there are many points with almost the same high value, and that the optimizer couldn't find anything better beyond that. For all the remaining seven knobs any value seems to be able to contribute to these highest throughputs, and so they probably don't matter much. We see that high innodb_sync_spin_loops combined with high innodb_spin_wait_delay never leads to above 7000 throughput, but other than that many combinations seem to be reasonable.

From	default	rep 1	rep 2	rep 3	innodb_dedicated_server=ON	original
preload_buffer_size	32768	134217728	131072	524288	-	131072
innodb_sync_spin_loops	30	80	30	50	-	400
innodb_spin_wait_delay	6	10	15	9	-	3
innodb_log_files_in_group	2	20	15	2	18	15
innodb_log_file_size	48M	48M	75M	150M	1024M	150M
innodb_io_capacity	200	400	900	100	-	400
innodb_flush_method	fsync	O_DIRECT	O_DIRECT	O_DIRECT	O_DIRECT_NO_FSYNC	O_DIRECT
innodb_change_buffering	all	inserts	deletes	changes	-	none
innodb_buffer_pool_size	128M	20480M	20480M	20480M	24576M	24576M
Throughput	2710	7077	7078	7081	6409	10620

Table 4.6: Benchmark: Twitter. The table contains the optimal configuration from each repetition of the final search space test, optimal configuration from the original search space test, the default configuration and configuration from when innodb_dedicated_server is enabled. The remaining of the 52 knobs are set to default values except for the original optimal configuration which can be viewed in table B.1 in appendix. A hyphen in innodb_dedicated_server=ON column indicate that the knob is not affected by innodb_dedicated_server and have the same value as the default column. K = 1024^1, M = 1024^2, G = 1024^3.

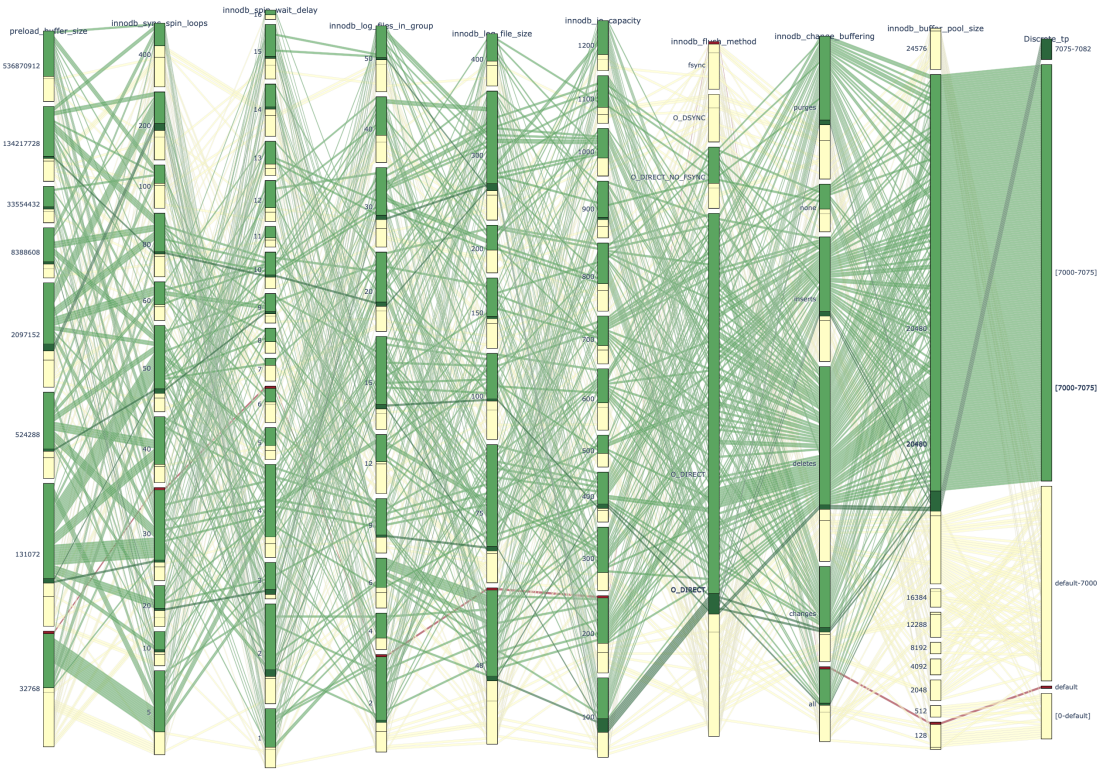

Figure 4.22: Parallel categories plot of all 300 different configurations and their resulting throughputs from the three optimization runs on Twitter. One default configuration is highlighted in red. The knobs innodb_buffer_pool_size and innodb_log_file_size are expressed in the unit of M = 1024^2.

4.4.3 YCSB

In Figure 4.23 we see that all three repetitions has converged by iteration 50 in the YCSB optimization.

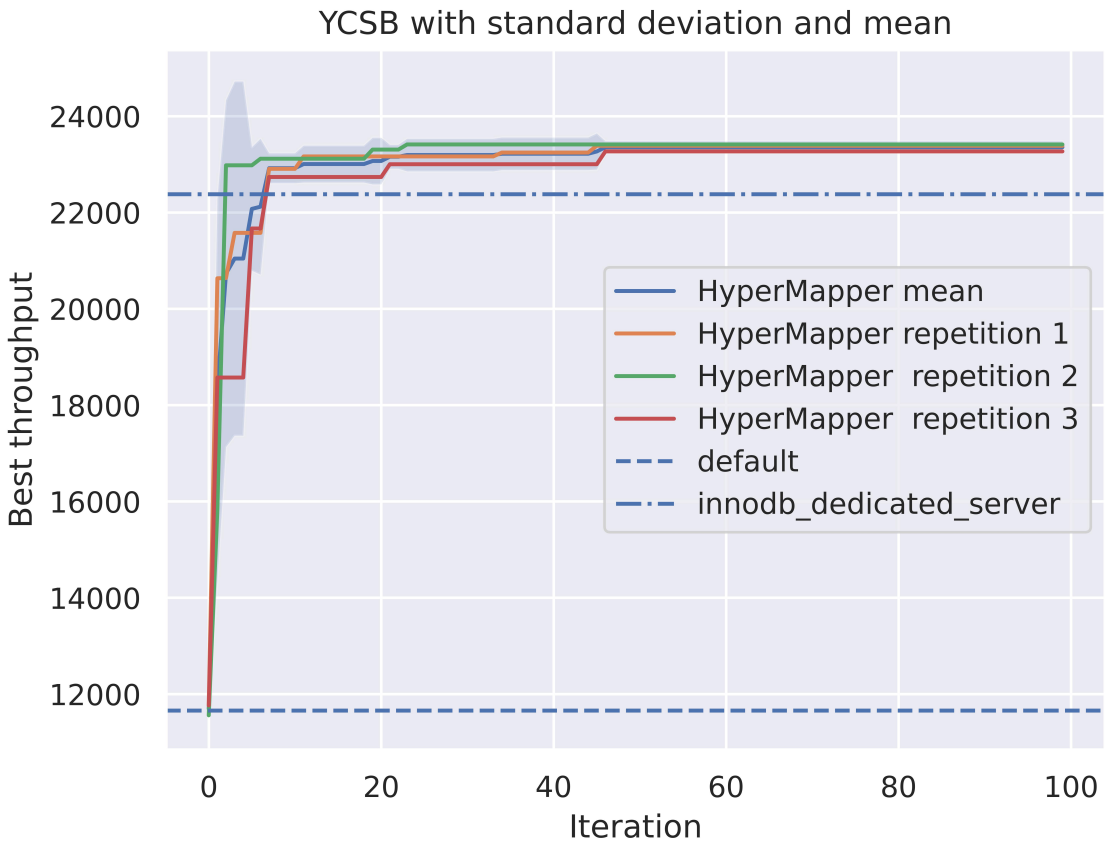

Figure 4.23: Throughput over best configuration found over time for all 3 repetitions with default configuration and innodb_dedicated_server=ON. Benchmark : YCSB.

Based on Table 4.7 we note that that the optimal points are higher than the best point from the original search space. This indicates that we have picked out at least some important knobs. From Figure 4.24 we see that innodb_flush_method and innodb_buffer_pool_size clearly are important. Some combinations of innodb_sync_spin_delay and innodb_spin_wait_delay seem to be better than others and for innodb_log_files_in_group, innodb_log_file_size and innodb_io_capacity values higher than default seem to be better. For innodb_change_buffering possibly "deletes" and "purges" are slightly better than default.

Iteration	default	rep 1	rep 2	rep 3	innodb_dedicated_server=ON	original
preload_buffer_size	32768	536870912	32768	536870912	-	33554432
innodb_sync_spin_loops	30	200	10	10	-	5
innodb_spin_wait_delay	6	1	7	3	-	6
innodb_log_files_in_group	2	40	50	12	18	20
innodb_log_file_size	48M	75M	75M	300	1024M	100M
innodb_io_capacity	200	700	500	1100	-	300
innodb_flush_method	fsync	O_DIRECT	O_DIRECT_NO_FSYNC	O_DIRECT_NO_FSYNC	O_DIRECT_NO_FSYNC	O_DIRECT
innodb_change_buffering	all	purges	deletes	deletes	-	none
innodb_buffer_pool_size	128M	20480M	20480M	20480M	24576M	24576M
Throughput	11692	23386	23413	23269	22600	21775

Table 4.7: Benchmark: YCSB. The table contains the optimal configuration from each repetition of the final search space test, optimal configuration from the original search space test, the default configuration and configuration from when innodb_dedicated_server is enabled. The remaining of the 52 knobs are set to default values except for the original optimal configuration which can be viewed in table B.1 in appendix. A hyphen in innodb_dedicated_server=ON column indicate that the knob is not affected by innodb_dedicated_server and have the same value as the default column. K = 1024^1, M = 1024^2, G = 1024^3.

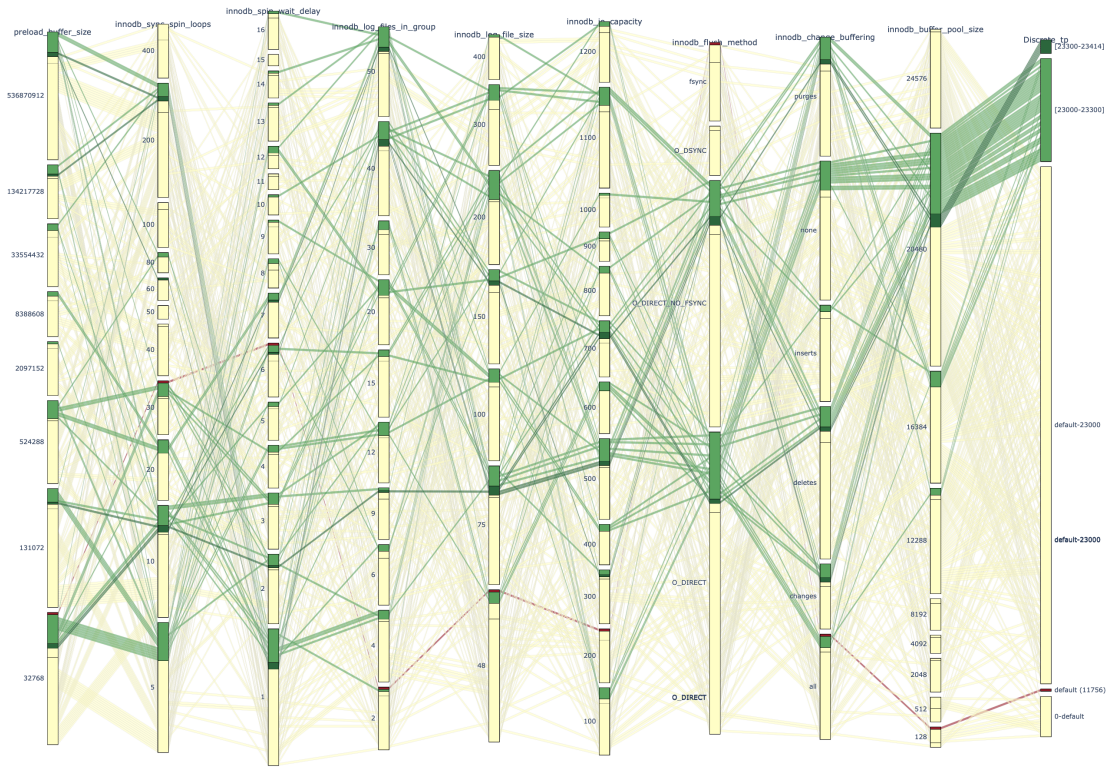

Figure 4.24: Parallel categories plot of all 300 different configurations and their resulting throughputs from the three optimization runs on Twitter. One default configuration is highlighted in red. The knobs innodb_buffer_pool_size and innodb_log_file_size are expressed in the unit of M = 1024^2.

4.5 Discussion

In this section we describe the results of our warm-up and measurement time investigation, the feature importance and the tests on the final search space. For the warm-up and measurement time we went with 240 + 120 seconds for TPC-C and Twitter, and 300 + 300 seconds for YCSB. These times are specific to the workloads we used, and with a different benchmark or in a real world setting a different choice might be more suitable.

We then presented the feature importance results on the three benchmarks. We identified nine knobs that stood out in terms of having higher feature importance numbers than others and according to more feature importance methods and on more than one of the benchmarks. Beyond the top nine we identified twelve honorable mentions who also have a reasonable chance of being important. Our data set was due to time constraints not large enough for the number of parameters, which is why we wanted to see some indication of importance from more than one method on one benchmark.

After identifying the top nine knobs we tested those in a final search space, letting the optimizer run for a total of 100 iterations. The best throughputs found on the three benchmarks are presented in Table 4.8, compared to the default configuration, default with innodb_dedicated_server turned on and also the best point found from the data set with the original large search space with 52 knobs as baseline references. We also present the speedups as a factor of the three baselines. We see that the optimizer was able to find configurations between 3 and 10% better than the default with innodb_dedicated_server turned on. The improvement compared to default was between 2x and 4.43x, but it seems that most of the improvement comes from the four knobs that innodb_dedicated_server sets automatically. For the Twitter benchmark however we see that in the large search configurations generating a throughput of at least 10,620 are possible, more than 65% better than innodb_dedicated_server. This indicates that we missed some important knobs in our final search space, and that through further refinement it should be possible to achieve better results compared to innodb_dedicated_server, at least on some workloads.

From our feature importance investigation there was always going to be things we were not able to see. We used just three workloads on one hardware configurations, and with changes to those different knobs were going to appear more important or less important. Not many knobs relating to thread concurrency made it into our final search space or as honorable mentions, and it might be that with a much higher number of connections, or users some of those might have been more important.

There is also the possibility that we missed one or two knobs that should have been included in the search space. Some niche knobs were intentionally left out, for example ones relating to full-text tables and full-text search were not included at all in the search space, but those might of course be important on some workloads. It is likely that we did not design the value ranges optimally. They are somewhat wide and sparse, especially the ones whose values we increase exponentially with a factor of two. This was to not completely miss the best values, but it might be that with a narrower and more dense range of reasonable values some knob that require precision would have shown greater importance because better values were found.

	TPC-C	Twitter	YCSB
Throughput			
default	322	2710	11692
innodb_dedicated_server=ON	1349	6409	22600
large search space	1447	10620	21775
final search space	1427 ± 9	7078 ± 2	23356 ± 63
Speedup			
default	4.43 ± 0.03	2.61 ± 0.00	2.00 ± 0.01
innodb_dedicated_server=ON	1.06 ± 0.01	1.10 ± 0.00	1.03 ± 0.00
large search space	0.99 ± 0.01	0.67 ± 0.00	1.07 ± 0.00

Table 4.8: The **Throughput** section presents the throughputs for the default configuration, default with innodb_dedicated_server=ON, the best configuration from the large search space test and from the final search space runs. The **Speedup** section presents the speedups gained from the final search space with respect to default, innodb_dedicated_server=ON and the large search space. The speedup section and the throughputs from the 3 final search space runs are expressed with one standard deviation.

It was discovered afterward that BenchBase have a high CPU utilization. For TPC-C, YCSB and Twitter respectively we observed that roughly 200, 250, 100 % (where 100 % equals full utilization of one core) was utilized by Benchbase itself according to the output of the command `top`. Having BenchBase and MySQL run from the same machine could potentially have affected the results in various ways.

Chapter 5
Conclusion and Future Work

5.1 Conclusion

In conclusion, after testing our final search space it still seems as if the nine knobs included in the final search space are among the most important ones. The knobs were preload_buffer_size, innodb_sync_spin_loops, innodb_spin_wait_delay, innodb_log_files_in_group, innodb_log_file_size, innodb_io_capacity, innodb_flush_method, innodb_change_buffering and innodb_buffer_pool_size. The possible performance gain in less than 100 iterations when optimizing these knobs is between 99.0 and 346.7% better than default, and between 3.0 and 10.5% better than default configuration with innodb_dedicated_server turned on.

However, in order to squeeze out more performance some changes can be made before settling on a search space suitable for an unknown workload. This should especially improve throughput for the Twitter benchmark where we know from the data collected for the feature importance that higher throughputs are possible. Four of the knobs, namely preload_buffer_size, innodb_flush_method, innodb_change_buffering and innodb_buffer_pool_size could probably be set to predetermined values and four or more additional knobs out of our "honorable mentions" could have be included. The seven honorable mentions that we looked deeper into, and that very well could have a great impact that we just did not see as much of in our experiments, were innodb_read_ahead_threshold, innodb_old_blocks_pct, innodb_lru_scan_depth, innodb_compression_failure_threshold_pct, innodb_buffer_pool_instances, innodb_adaptive_hash_index_parts and innodb_adaptive_hash_index. Of these innodb_compression_failure_threshold_pct and innodb_adaptive_hash_index_parts had the highest feature importance values for Twitter and should therefore probably be included in an adjusted search space.

Beyond these seven, the five knobs thread_stack, thread_cache_size, innodb_random_read_ahead, _old_blocks_time and innodb_commit_concurrency could potentially also be impactful, but with lower probability. Of these we would try including innodb_random_read_ahead because of its relevance to the Twitter benchmark.

5.2 Future work

Since we know that even better performance is possible, more work is needed to refine the final search space. As previously described this would include setting some knobs to their obvious best values and including other knobs in the search space. It would also include narrowing down the value ranges to include the relevant areas only. Since there is also the possibility of setting a prior distribution for the knobs, this could be done to guide the optimization into the likely good areas in order to shorten the number of iterations needed for convergence.

In order to become more certain of the universal applicability of the search space one could also gather data and do feature importance using other computers and hardware configurations. For example there are "storage optimized" instances for AWS, which as opposed to the "general purpose" instance m5d.2xlarge that we used is optimized specifically for high sequential read and write access to local storage. Those could be relevant for database applications, and could possibly produce different feature importance results. There is of course also the possibility of testing workloads other than the three used in this project.

In our experiments the workload was always the same, but in real world scenario the workload is likely going to change over time. When the workload remains relatively stable for several hours there is time to optimize, but optimization is difficult if the workload shifts to much and the throughput and general behaviour of the system changes. For this reason some sort of workload categorisation or identification is a topic for the future. Additionally, in this thesis we did not investigate to what degree different workloads require different configurations to run optimally. Whenever the workload changes drastically it might be worth it to go through the optimization process again, and save the optimal points for when the same workload is present again in the future.

Another idea is to tailor the search space to the specific workload. If it is known that some knobs are only relevant in certain situations, one could then leave those out of the search space and leave them at their default value when the workload does not call for them.

Another idea is to investigate the possibility of changing the warm-up and measurement time as the optimization goes on. All alternatives are possible here. For one it might be easier to differentiate between configurations early on in the optimization, and so when guiding the optimizer into the right areas (exploration) to explore further (exploitation) one could maybe get away with shorter times. On the other hand, when the optimizer is in the right area and is fine-tuning knobs (exploitation) the behaviour of different configurations might be so similar that it is instead easier to differentiate between slightly worse and slightly better ones. Or it might take longer to differentiate between them because of measurement noise being more significant in relation to the differences in throughput. It is unclear if one should start or end with longer warm-up and measurement times. Through further investigation into this it might be possible to reduce the total optimization time.

When investigating which warm-up and measurement time is suitable, it might be of interest to put a number on the similarity between different warm-up and measurement times, instead of doing it visually as in this project. As previously mentioned one alternative is to put all distances in a vector and compare to the "true" distance vector, for example using Euclidean distance or cosine similarity. Cosine similarity would have the advantage of being a value between -1 and 1, and one could pick a value such as 0.9 that would be considered good enough.

References

[1] Amazon. Benchmarking databases, June 2022. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html.

[2] Amazon. Benchmarking databases, June 2022. URL: https://aws.amazon.com/ec2/instance-types/.

[3] André Biedenkapp, Joshua Marben, Marius Lindauer, and Frank Hutter. Cave: Configuration assessment, visualization and evaluation. In *International Conference on Learning and Intelligent Optimization*, pages 115–130. Springer, 2018.

[4] Zhen Cao, Geoff Kuenning, and Erez Zadok. Carver: Finding important parameters for storage system tuning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, 2020.

[5] CAVE. Input format, 2021. URL: https://automl.github.io/CAVE/stable/manualdoc/fileformats.html#id1.

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

[7] Transaction Processing Performance Council. Tpc benchmark, February 2010. URL: https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[8] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[9] Eldawebi, Osama. Efficient Optimization of Databases Using Parameter Importance Methods, 2022. Student Paper.

[10] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[11] Hypermapper. Home, June 2022. URL: https://github.com/luinardi/hypermapper/wiki.

[12] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. Too many knobs to tune? towards faster database tuning by pre-selecting important knobs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[13] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. Too many knobs to tune? towards faster database tuning by pre-selecting important knobs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[14] Ceshine Lee. Feature importance measures for tree models — part i. URL: https://medium.com/the-artificial-impostor/feature-importance-measures-for-tree-models-part-i-47f187c1a2c3.

[15] Luigi Nardi, Bruno Bodin, Sajad Saeedi, Emanuele Vespa, Andrew J Davison, and Paul HJ Kelly. Algorithmic performance-accuracy trade-off in 3d vision applications using hypermapper. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1434–1443. IEEE, 2017.

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

[17] MySQL official documentation. Innodb architecture. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-architecture.html.

[18] MySQL official documentation. Innodb in-memory structures. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-in-memory-structures.html.

[19] MySQL official documentation. Innodb on-disk structures. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-on-disk-structures.html.

[20] Oracle. Mysql enterprise edition. URL: https://www.mysql.com/products/enterprise/scalability.html.

[21] Oracle. Tpc-c, June 2022. URL: https://www.mysql.com/customers/.

[22] solid IT. Tpc-c, June 2022. URL: https://db-engines.com/en/ranking.

[23] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

Appendices

Appendix A

Importance plots

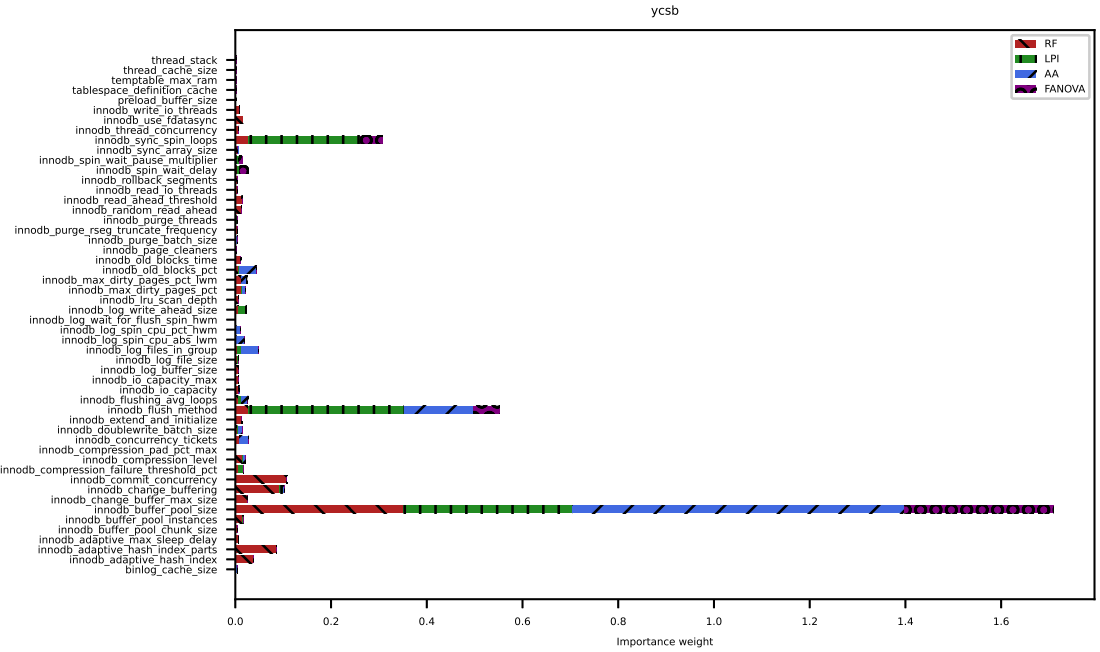

Figure A.1: Knob importance weight for YCSB according to RF, LPI, AA, fANOVA. A zoomed in plot can be found in A.1

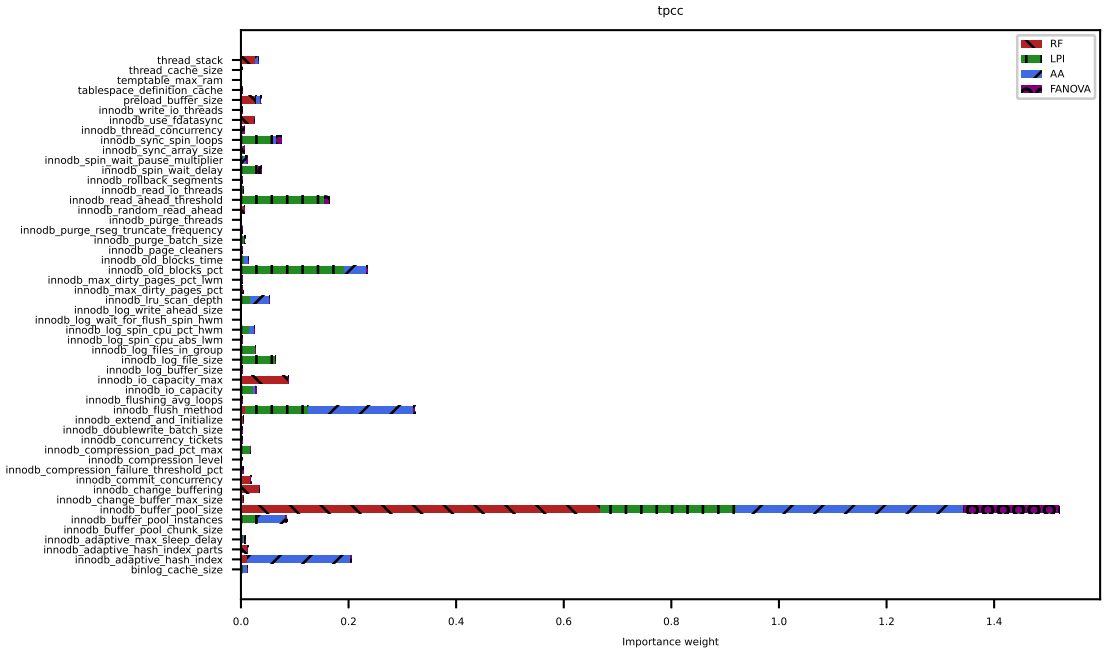

Figure A.2: Knob importance weight for TPC-C according to RF, LPI, AA, fANOVA. A zoomed in plot can be found in 4.10.

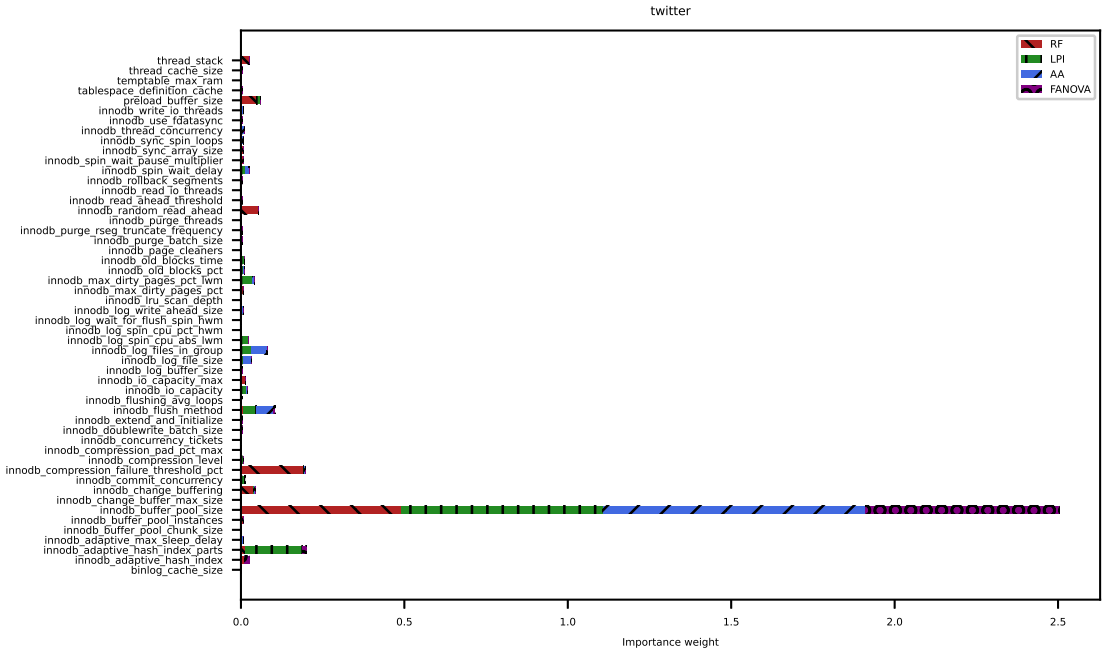

Figure A.3: Knob importance weight for Twitter according to RF,LPI ,AA , fANOVA. A zoomed in plot can be found in 4.13.

Appendix B

Knob value tables

benchmark	*TPC-C*	*Twitter*	*YCSB*
innodb_buffer_pool_size	20480	24576	24576
innodb_buffer_pool_instances	2	10	6
innodb_buffer_pool_chunk_size	64	512	256
innodb_old_blocks_time	1000	100	1000
innodb_random_read_ahead	OFF	OFF	OFF
innodb_old_blocks_pct	30	5	40
innodb_read_ahead_threshold	24	40	64
innodb_page_cleaners	8	6	4
innodb_max_dirty_pages_pct	85	70	65
innodb_max_dirty_pages_pct_lwm	0	20	20
innodb_lru_scan_depth	1024	128	2048
innodb_flushing_avg_loops	200	10	500
innodb_io_capacity	800	400	300
innodb_io_capacity_max	1200	2800	4400
innodb_flush_method	O_DIRECT_NO_FSYNC	O_DIRECT	O_DIRECT
innodb_use_fdatasync	OFF	OFF	OFF
innodb_change_buffer_max_size	15	15	5
innodb_change_buffering	purges	none	none
innodb_adaptive_hash_index	ON	OFF	ON
innodb_adaptive_hash_index_parts	2	2	32
innodb_log_buffer_size	512	512	16
innodb_rollback_segments	128	64	32
innodb_purge_rseg_truncate_frequency	64	8	128
innodb_extend_and_initialize	ON	OFF	ON
innodb_doublewrite_batch_size	1	1	2
innodb_log_file_size	200	150	100
innodb_log_files_in_group	30	15	20
innodb_log_write_ahead_size	512	1024	16384
innodb_thread_concurrency	513	169	482
innodb_adaptive_max_sleep_delay	300000	20000	40000
innodb_concurrency_tickets	28100	8900	15800
innodb_read_io_threads	8	4	4
innodb_write_io_threads	8	64	64
innodb_spin_wait_delay	1	3	6
innodb_spin_wait_pause_multiplier	10	70	50
innodb_sync_array_size	1	64	64
innodb_sync_spin_loops	200	400	5
innodb_commit_concurrency	700	50	300
innodb_compression_failure_threshold_pct	3	2	15
innodb_compression_level	1	6	3
innodb_compression_pad_pct_max	65	65	65
innodb_log_spin_cpu_abs_lwm	320	400	80
innodb_log_spin_cpu_pct_hwm	40	30	70
innodb_log_wait_for_flush_spin_hwm	1600	409600	409600
innodb_purge_batch_size	400	800	700
innodb_purge_threads	16	24	4
thread_cache_size	20	4	50
binlog_cache_size	2097152	8388608	134217728
preload_buffer_size	32768	131072	33554432
thread_stack	2097152	8388608	4194304
tablespace_definition_cache	8192	2048	256
temptable_max_ram	4096	3072	2048
Throughput	1447.252185	10620.104755	21774.879133

Table B.1: The optimal configuration from original knob search space.

parameter	*parameter_default*	*parameter_type*	*values*	*unit*	*F/H*
innodb_buffer_pool_size	128	ordinal	[128, 512, 2048, 4092, 8192, 12288, 16384, 20480, 24576]	M	F
innodb_buffer_pool_instances	8	ordinal	[1, 2, 4, 6, 8, 10, 14, 18, 24]		H
innodb_buffer_pool_chunk_size	128	ordinal	[32, 64, 128, 256, 512]	M	
innodb_old_blocks_time	1000	ordinal	[0, 10, 32, 100, 316, 1000, 3160, 10000, 31600, 100000]		H
innodb_random_read_ahead	OFF	categorical	['ON', 'OFF']		H
innodb_old_blocks_pct	37	ordinal	[5, 10, 20, 30, 37, 40, 50, 60, 70, 80, 90, 95]		H
innodb_read_ahead_threshold	56	ordinal	[0, 8, 16, 24, 32, 40, 48, 56, 64]		H
innodb_page_cleaners	4	ordinal	[1, 2, 4, 6, 8, 10, 14]		
innodb_max_dirty_pages_pct	90	ordinal	[50, 55, 60, 65, 70, 75, 80, 85, 90, 95]		
innodb_max_dirty_pages_pct_lwm	10	ordinal	[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]		
innodb_lru_scan_depth	1024	ordinal	[128, 256, 512, 1024, 2048, 4096, 8192, 16384]		H
innodb_flushing_avg_loops	30	ordinal	[5, 10, 20, 30, 40, 50, 60, 80, 100, 200, 500]		
innodb_io_capacity	200	ordinal	[100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200]		F
innodb_io_capacity_max	2000	ordinal	[1200, 2000, 2800, 3600, 4400]		
innodb_flush_method	fsync	categorical	['fsync', 'O_DSYNC', 'O_DIRECT', 'O_DIRECT_NO_FSYNC']		F
innodb_use_fdatasync	OFF	categorical	['ON', 'OFF']		
innodb_change_buffer_max_size	25	ordinal	[5, 15, 25, 35, 45]		
innodb_change_buffering	all	categorical	['none', 'inserts', 'deletes', 'changes', 'purges', 'all']		F
innodb_adaptive_hash_index	ON	categorical	['ON', 'OFF']		H
innodb_adaptive_hash_index_parts	8	ordinal	[2, 8, 32, 128, 512]		H
innodb_log_buffer_size	16	ordinal	[16, 32, 64, 128, 256, 512]	M	
innodb_rollback_segments	128	ordinal	[8, 16, 32, 64, 128]		
innodb_purge_rseg_truncate_frequency	128	ordinal	[8, 16, 32, 64, 128]		
innodb_extend_and_initialize	ON	categorical	['ON', 'OFF']		
innodb_doublewrite_batch_size	0	ordinal	[0, 1, 2, 4, 8, 16, 32, 64, 128]		
innodb_log_file_size	48	ordinal	[48, 75, 100, 150, 200, 300, 400]	M	F
innodb_log_files_in_group	2	ordinal	[2, 4, 6, 9, 12, 15, 20, 30, 40, 50]		F
innodb_log_write_ahead_size	8192	ordinal	[512, 1024, 2048, 4096, 8192, 16384]		
innodb_thread_concurrency	600	integer	[1, 600]		
innodb_adaptive_max_sleep_delay	150000	ordinal	[5000, 10000, 20000, 40000, 80000, 150000, 300000]		
innodb_concurrency_tickets	5000	ordinal	[500, 890, 1580, 2810, 5000, 8900, 15800, 28100, 50000]		
innodb_read_io_threads	4	ordinal	[4, 8, 16, 32, 64]		
innodb_write_io_threads	4	ordinal	[4, 8, 16, 32, 64]		
innodb_spin_wait_delay	6	ordinal	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]		F
innodb_spin_wait_pause_multiplier	50	ordinal	[1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]		
innodb_sync_array_size	1	ordinal	[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]		
innodb_sync_spin_loops	30	ordinal	[5, 10, 20, 30, 40, 50, 60, 80, 100, 200, 400]		F
innodb_commit_concurrency	1000	ordinal	[50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]		H
innodb_compression_failure_threshold_pct	5	ordinal	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25]		H
innodb_compression_level	6	ordinal	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]		
innodb_compression_pad_pct_max	50	ordinal	[30, 35, 40, 45, 50, 55, 60, 65, 70]		
innodb_log_spin_cpu_abs_lwm	160	ordinal	[0, 40, 80, 120, 160, 200, 240, 280, 320, 360, 400]		
innodb_log_spin_cpu_pct_hwm	50	ordinal	[30, 40, 50, 60, 70, 80, 90, 100]		
innodb_log_wait_for_flush_spin_hwm	400	ordinal	[100, 400, 1600, 6400, 25600, 102400, 409600]		
innodb_purge_batch_size	300	ordinal	[100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]		
innodb_purge_threads	4	ordinal	[4, 8, 16, 24, 32]		
thread_cache_size	9	ordinal	[1, 2, 4, 6, 8, 9, 10, 12, 15, 20, 30, 50]		H
binlog_cache_size	32768	ordinal	[32768, 131072, 524288, 2097152, 8388608, 33554432, 134217728, 536870912]		
preload_buffer_size	32768	ordinal	[32768, 131072, 524288, 2097152, 8388608, 33554432, 134217728, 536870912]		F
thread_stack	1048576	ordinal	[262144, 524288, 1048576, 2097152, 4194304, 8388608]		H
tablespace_definition_cache	256	ordinal	[256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288]		
temptable_max_ram	1024	ordinal	[512, 1024, 2048, 3072, 4096]	M	

Table B.2: The original knob search space. The columns ***parameter_type***, ***values***, ***parameter_default*** corresponds to the properties used for defining a input parameter in HyperMapper. If the unit entry is used then the parameter's values is multiplied with that unit. Blanks unit entry means no units is used. The last field indicate if a knob is part of the final knobs (***F***) or honorable mentions (***H***) described in section 4.3.1 "The most important knobs". Possible units are K = 1024^1, M = 1024^2, G = 1024^3.

knob/point	0	1	2	3	4	5	6	7	8	9
innodb_buffer_pool_size	24576	24576	24576	24576	24576	24576	24576	24576	24576	24576
innodb_buffer_pool_instances	14	2	6	4	2	6	24	8	24	8
innodb_buffer_pool_chunk_size	32	64	256	32	512	32	256	512	64	128
innodb_old_blocks_time	100	100	31600	31600	1000	31600	100000	100000	100000	1000
innodb_random_read_ahead	OFF	ON	ON	ON	OFF	ON	ON	OFF	ON	OFF
innodb_old_blocks_pct	30	90	90	30	90	70	90	30	60	37
innodb_read_ahead_threshold	24	32	32	0	24	64	56	0	56	56
innodb_page_cleaners	4	14	14	2	6	10	14	8	4	4
innodb_max_dirty_pages_pct	70	85	75	65	55	65	95	70	85	90
innodb_max_dirty_pages_pct_lwm	40	5	20	30	40	0	25	5	10	10
innodb_lru_scan_depth	4096	2048	1024	512	16384	16384	8192	2048	16384	1024
innodb_flushing_avg_loops	40	40	80	5	10	100	40	20	5	30
innodb_io_capacity	1100	100	900	700	200	400	500	1200	900	200
innodb_io_capacity_max	2000	2800	2800	2000	1200	2000	1200	1200	2800	2000
innodb_flush_method	O_DSYNC	O_DIRECT	O_DSYNC	fsync	O_DSYNC	O_DIRECT_NO_FSYNC	fsync	fsync	fsync	fsync
innodb_use_fdatasync	ON	ON	OFF	OFF	ON	OFF	OFF	OFF	OFF	OFF
innodb_change_buffer_max_size	25	45	45	5	25	15	35	25	45	25
innodb_change_buffering	deletes	all	changes	inserts	inserts	changes	inserts	changes	purges	all
innodb_adaptive_hash_index	OFF	ON	OFF	OFF	ON	ON	ON	ON	OFF	ON
innodb_adaptive_hash_index_parts	8	32	32	128	8	8	8	32	2	8
innodb_log_buffer_size	512	256	512	32	64	64	128	512	256	16
innodb_rollback_segments	128	8	128	32	8	64	32	16	32	128
innodb_purge_rseg_truncate_frequency	16	8	64	64	128	32	16	64	32	128
innodb_extend_and_initialize	OFF	OFF	OFF	ON	OFF	OFF	ON	ON	ON	ON
innodb_doublewrite_batch_size	8	2	8	16	2	2	0	2	1	0
innodb_log_file_size	150	100	100	75	200	200	150	200	150	48
innodb_log_files_in_group	9	50	50	12	9	15	6	12	4	2
innodb_log_write_ahead_size	16384	1024	16384	512	2048	512	1024	2048	8192	8192
innodb_thread_concurrency	127	62	572	184	530	494	412	573	57	600
innodb_adaptive_max_sleep_delay	80000	300000	40000	300000	20000	80000	80000	150000	150000	150000
innodb_concurrency_tickets	28100	15800	28100	2810	890	2810	15800	890	28100	5000
innodb_read_io_threads	4	16	8	64	32	4	4	16	4	4
innodb_write_io_threads	4	64	64	64	64	32	32	16	32	4
innodb_spin_wait_delay	5	3	15	13	5	1	7	8	15	6
innodb_spin_wait_pause_multiplier	20	30	30	100	1	20	30	10	70	50
innodb_sync_array_size	1	1	4	4	512	64	128	16	128	1
innodb_sync_spin_loops	200	50	10	50	60	20	200	100	10	30
innodb_commit_concurrency	200	700	300	200	800	800	200	200	100	1000
innodb_compression_failure_threshold_pct	1	7	3	4	5	25	8	1	1	5
innodb_compression_level	8	6	7	0	2	1	9	6	4	6
innodb_compression_pad_pct_max	30	55	40	30	65	35	70	60	65	50
innodb_log_spin_cpu_abs_lwm	210	90	210	30	270	30	60	60	180	120
innodb_log_spin_cpu_pct_hwm	50	30	100	60	40	40	70	60	100	50
innodb_log_wait_for_flush_spin_hwm	409600	102400	1600	400	1600	102400	1600	409600	6400	400
innodb_purge_batch_size	1000	900	600	400	200	1000	1000	100	200	300
innodb_purge_threads	32	24	8	16	8	16	16	8	16	4
thread_cache_size	4	4	12	1	10	20	4	15	30	9
binlog_cache_size	2097152	2097152	2097152	134217728	134217728	8388608	524288	32768	134217728	32768
preload_buffer_size	524288	2097152	2097152	2097152	32768	2097152	134217728	131072	134217728	32768
thread_stack	262144	2097152	4194304	4194304	262144	2097152	1048576	8388608	4194304	1048576
tablespace_definition_cache	512	524288	256	4096	512	4096	65536	1024	256	256
temptable_max_ram	4096	3072	512	1024	4096	3072	2048	1024	512	1024
TPC-C throughput	1312.36	1298.56	1303.04	1197.86	1125.93	1301.11	1216.95	1298.10	1205.54	694.30
Twitter throughput	5335.97	5945.00	5941.73	3607.23	4470.90	7278.07	5518.72	3400.97	4951.89	4945.64
YCSB throughput	19610.53	20579.03	17609.30	16946.67	17410.11	21523.39	19692.18	19742.13	20145.50	18709.85

Table B.3: The 10 random points used in section 4.2 “Warm-up and measurement time”.

Appendix C
Data

The data for the tests can be found here. Only 1040 first samples in the large knob search space tests and only 100 first samples in final search space test were used. The rest were discarded. The date for importance plot can be found here.

Appendix D

Code

```
"""
Script that formats the results and scenario of a HyperMapper optimization procedure to files
    supported by CAVE.
We create a configspace.json file and a folder with some .csv files.
"""
# 1
# CAVE keyword : (->) HM keyword
# cost : [objective name]
# time : runtime to evaluate a configuration - in our case not really applicable so use a constant
# param1: param1...

import os
import pandas as pd
import json
import pathlib
from pathlib import Path
from shutil import copyfile
from textwrap import dedent
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-hmf","--hypermapper-folder",required=True,dest='
    hypermapper_folder',type=str,help="folder that contain hypermapper csv files")
parser.add_argument("-ss","--search-space" ,required=True,type=str,dest="
    search_space_file",help="the json search space file used in hypermapper")
parser.add_argument("-o","--output_folder",required=True,type=str,dest="output_folder"
    ,help="the folder which hm_to_cave will output to")
```

```
24 parser.add_argument("—sf","——scenario—file",default=None, nargs="?", type=str,dest="
       scenario_txt_file",help="a standard scenario.txt file. See hm_to_cave_example")
25 args = parser.parse_args()
26
27 output_folder = args.output_folder
28 hypermapper_folder = args.hypermapper_folder
29 search_space_file = args.search_space_file
30 scenario_txt_file = args.scenario_txt_file
31
32
33
34
35
36
37 for _, file in enumerate([x for x in os.listdir(hypermapper_folder) if x.endswith(".csv")]):
38
39     hm_df = pd.read_csv(os.path.join(hypermapper_folder, file))
40     hm_df = hm_df.drop(columns=["Timestamp"])
41     hm_df = hm_df.rename(columns={"Throughput": "cost"})
42     cave_df = pd.DataFrame(data=hm_df)
43     trajectory_df = cave_df.copy()
44     cave_df["time"] = [0.1] * len(hm_df)
45     cave_df["status"] = ["SUCCESS"] * len(hm_df)
46     cave_df["seed"] = [1] * len(hm_df)
47
48
49
50
51     # Create trajectory file
52     trajectory_df["cpu_time"] = [0.1] * len(hm_df)
53     trajectory_df["wallclock_time"] = [0.1] * len(hm_df)
54     trajectory_df["evaluations"] = [0] * len(hm_df)
55     cost_array = trajectory_df["cost"].tolist()
56     prev_best = cost_array[0]
57     prev_best_index = 0
58     drop_indices = []
59
60
61
62     for ind, cost in enumerate(cost_array):
63         if cost < prev_best:
64             trajectory_df.at[ind,"evaluations"] = ind
65             prev_best = cost
66             prev_best_index = ind
67         else:
68             if ind > 0:
```

```
                    drop_indices.append(ind)
    trajectory_df = trajectory_df.drop(drop_indices)

    filename, extension = os.path.splitext(file)
    instance_path = os.path.join(output_folder,filename)

    pathlib.Path(instance_path).mkdir(parents=True, exist_ok=True)


    cave_df.to_csv( os.path.join(instance_path, "runhistory.csv"), index=False)
    trajectory_df.to_csv(os.path.join(instance_path ,"trajectory.csv"), index=False)
    # copy over scneario file

    if scenario_txt_file is None:
        file = open(os.path.join(instance_path, "scenario.txt"),"w")
        file.write(dedent(
        """
        paramfile = ./configspace.json
        run_obj = quality
        """))
    else:
        copyfile( scenario_txt_file, os.path.join(instance_path, "scenario.txt"))


# 2. Format configuration space
# {hyperparameters: [ {param1}, {param2} ]}



# create configspace
configspace = {"hyperparameters": []}
with open(search_space_file) as f:

    data = json.load(f) # dict
    for param in data.items():
        entry = {}
        entry["name"] = param[0]
        #print(param)
        if param[1]["parameter_type"] in ("integer", "real"):
            if param[1]["parameter_type"] == "integer":
                entry["type"] = "uniform_int"
            else:
                entry["type"] = "uniform_float"
            entry["log"] = False
            values = param[1]["values"]
            entry["lower"] = values[0]
```

```
115                 entry["upper"] = values[1]
116                 entry["default"] = param[1]["parameter_default"]
117             else: # categorical or ordinal --> categorical in CAVE
118                 entry["type"] = "categorical"
119
120                 if param[1]["parameter_default"] in ("false", "true"):
121                     entry["choices"] = ["False", "True"]
122                     entry["default"] = (
123                         "False" if param[1]["parameter_default"] == "false" else "True"
124                     )
125                 else:
126                     entry["choices"] = [str(x) for x in param[1]["values"]]
127                     entry["default"] = str(param[1]["parameter_default"])
128                 entry["probabilities"] = None
129
130             configspace["hyperparameters"].append(entry)
131
132 configspace["conditions"] = []
133 configspace["forbiddens"] = []
134
135
136
137 with open( os.path.join(output_folder, "configspace.json"), "w") as f:
138     json.dump(configspace, f, indent=2, ensure_ascii=False)
```

Listing D.1: The code for converting HyperMapper results to format acceptable by CAVE, provided by our colleague (and slightly modified by us), described in 3.1 “Converting HyperMapper results to CAVE format”.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-08-25

EXAMENSARBETE ML-driven self-tuning MySQL-databases

STUDENTER Asmail Abdulkarim, Filip Johansson
HANDLEDARE Luigi Nardi (LTH)
EXAMINATOR Pierre Nugues (LTH)

Snabba upp världens databaser

POPULÄRVETENSKAPLIG SAMMANFATTNING **Asmail Abdulkarim, Filip Johansson**

Oräkneliga timmar spenderade på att justera databasparametrar, miljarder kWh elektricitet som slösas bort, väntetider oändligt långa. Lösningen på alla problem är automatisk justering av databassystemet med hjälp av maskininlärning!

Många molnleverantörer erbjuder services där kunden kan lagra sin databas på en av deras datacenter, för en månadsavgift där ju fler maskiner som används desto dyrare blir avgiften. Genom optimera databasen så den blir dubbelt så snabb, kan en tung service som tidigare krävde två maskiner utföras med bara en maskin och därmed halva kostnaden.

Men att optimera databasens hastighet är ett svårt problem även för experter på grund av flera faktorer. De främsta faktorn är att de finns hundratals parametrar och varje utvärdering av prestandan kräver en benchmarkkörning som tar minuter att utföra. Dessa faktorer bland många fler gör det svårt att optimera manuellt. Därför har många forskat för att automatisera processen. Det första steget i automatisering är att hitta vilka parameter som har stor betydelse för hastigheten och därefter försöker optimera dessa parameter. Det är vad vi har gjort för världens näst mest populära databashanteringssystem MySQL.

Vi gick igenom strukturen och funktionen av MySQL och alla parametrar som kan ställas in av administratören för att ändra hur systemet fungerar. Av de flera hundra parametrarna identifierade vi som ett första steg 52 st. som hade en viss chans att påverka prestandan. Vi samlade sedan in en datamängd med 1040 datapunkter, vilket innebär en viss konfiguration av de 52 parametrarna och den resulterande genomströmningen. Vi gjorde detta för tre olika "benchmarks", som är artificiella databaser med artificiella arbetsbelastningar av läs- och skrivanrop. Vid benchmarking tar det lite tid för genomströmningen att stabilisera sig, så man måste kasta bort de första minuterna av mätning, och man behöver även mäta över en tillräckligt lång tidsperiod för att få en stabil mätning. Vi undersökte därför först hur kort uppvärmningstid och mättid vi kunde komma undan med och ändå rangordna olika konfigurationer korrekt i förhållande till varandra.

På våra tre datamängder använde vi fyra olika feature importance metoder som uppskattar parametrarnas relativa importans. Genom att aggregera dessa resultat lyckades vi identifiera nio parametrar som verkade vara viktigast för genomströmningen. Vi har också identifierat tolv nämnvärda parameterar som också har en stor chans att påverka. Genom att testa detta sök utrymme med nio parametrar kunde vår optimerare hitta kombinationer av inställningar som uppnår mellan 99% och 346% högre genomströmning än standardinställningen, beroende på arbetsbelastningsstrukturen, på bara några timmar. Vi observerar också att ytterligare ökningar är möjliga för lästunga arbetsbelastningar genom att lägga till några av de nämnvärda till optimeringen, och att detta sannolikt skulle resultera i ett sökutrymme som passar alla arbetsbelastningar.