Automatic configuration of the Cassandra database using irace

Moisés Silva-Muñoz, Alberto Franzin, Hugues Bersini

18 December 2020

Abstract

Database systems play a central role in modern data-centered applications. Their performance is thus a key factor in the efficiency of data processing pipelines. Modern database systems expose several parameters that users and database administrators can configure to tailor the database settings to the specific application considered. While this task has traditionally been performed manually, in the last years several methods have been proposed to automatically find the best parameter configuration for a database. Several of these methods, however, use statistical models that require high amounts of data and fail to represent all the factors that impact the performance of a database, or implement complex algorithmic solutions. In this work we study the potential of a simple model-free general-purpose configurator, irace, to automatically find the best parameter configuration of a database. In our experiments, conducted configuring the Cassandra NoSQL database using the YCBS benchmark, we first establish a reliable experimental setup, then we evaluate irace under different scenarios, with different tuning budgets. We report speedups of up to 30% over the default configuration in terms of throughput, and we provide an analysis of the configurations obtained.

1 Introduction

With the continuously increasing deployment of data-centered applications, the performance of the data collection and processing pipeline is a key factor to consider when developing high performing applications. In particular, databases form a crucial part of this pipeline, as they operate on the slowest part of the pipeline, the storage. The performance of a database depends on several factors, including the hardware it operates on, the data stored, and its parameter configuration [10]. Databases, in fact, expose several choices that impact their performance, so that users and practitioners can choose the most appropriate configuration for their scenario [28]. Databases are usually provided with a high quality default configuration, adjusted by the developers over the years, to accommodate as many use cases as possible. In many practical applications, however, the structure or amount of data, or the workload, are known, or can be estimated in advance. It is therefore possible to improve the performance of the database on a specific scenario, by tailoring the most appropriate configuration based on the data and workload of each specific case [10, 59].

While historically the task of finding the best configuration has been mostly a manual task, researchers began to explore automatic methods since at least 2003, at first trying to explicitly model the relationship of database parameters [25]. Along with the rise of automatic configuration methods in optimization and machine learning, and in general with the adoption of artificial intelligence techniques also in databases, the last years have seen a surge in automatic methods proposed also for databases [61]. Several techniques have been used, from design of experiments to neural networks, to search algorithms [28]. Automatic configuration methods offer several advantages [46]. First, they unburden users and practitioners of the tedious and time-consuming manual task of finding the best configuration for a certain scenario. Second, users can only evaluate a limited set of alternatives in manual experiments, injecting their own biases in the process and likely obtain very suboptimal results. Consequently, obtaining good results with a manual configuration process requires a high expertise on both the database and the specific scenario, while automatic configuration methods make this task accessible also to non-experts. Automatic methods, therefore, can be expected to find better configurations, with significantly less manual effort [46].

The majority of the works proposed attempt to either evaluate configurations that are representative of the entire parameter space, or to learn some statistical model of the relationship between the parameter configuration of the database and performance obtained [28, 35, 14, 16, 41, 60, 31]. These application of these methods, however, requires great care. The evaluation of a database configuration is a heavy task: it requires a considerable amount of operations to be considered reliable, due to the stochastic nature of the task and to the hardware and software techniques to improve the performance, such as caching. Very short experiments may not be representative enough of the database performance [62]. Therefore, configuring a database is an extremely computationally heavy task, that requires users to carefully balance the computational effort with the quality of the results, both in terms of efficiency and reliability of the final configuration obtained. Furthermore, attempts to model the relationship between parameters and the results cannot possibly include, or measure, all the factors that impact the performance of the database. For example, the specific hardware specifications, or the network characteristics in case of a distributed database, have an impact on the database performance, yet it is impossible to both model all such factors, and to maintain feasible computational requirements for the tuning task [24]. In other words, we want to invest as little time and computing power as possible, to find a configuration that consistently gives the best results; and this unfortunately imposes a limit the number of evaluations we can perform, which in turn prevents the observation of enough data to learn a good statistical model of the database performance. Attempts to devise a clever methodology that satisfies all these requirements can easily turn into complex algorithms, where several different techniques are combined. Last, a model of said relationships for a specific task or database is not transferable to, for example, a different database [5, 24].

A certain number of works consider instead the database configuration problem as an optimization problem over a (potentially) mixed space of variables [59, 26]. While these methods are, in principle, database-independent, they are very complex. Developing an efficient optimization algorithm requires also to evaluate design choices and to fine tune its parameters, a task that effectively requires to repeatedly evaluate the performance of the algorithm in configuring a database, with an explosion in computational demands. Moreover, design choices and parameter values that are good for one database or scenario are not necessarily good for a different one.

We instead advocate an alternative, model-free approach that makes use of an easy-to-use, well-tested, general-purpose configurator to find the best settings for a database. Finding the best configuration for a database is, conceptually, no different than finding the best configuration for any other software or algorithm for which the quality of the output can be measured [8, 6]. This problem arises in several other fields, and has been the subject of several lines of research, that produced powerful methods [19, 7, 20]. In particular, we use the irace R package to find the best configuration of the Cassandra database for given scenarios [9, 27]. Irace is a state-of-the-art configurator, used to improve the performance for several applications, from optimization algorithms to machine learning, from compilers to the automatic design of algorithms. Irace is as a model-free black-box stochastic optimizer, so it does not need to assume any statistical model of the relationship between the parameters and the target algorithm performance (the database, in our case), and it can be applied to any parameterized algorithm or system. It is also easy to use, since it requires only to set up few scripts and configuraton files, without the need to implement complex algorithmic solutions. We show how irace can improve over the already high-quality default configuration, and that the potential for improvement increases the more specific the scenario is.

Our contribution is three-fold. We first separate the tasks of evaluating a configuration and configuring a database, and devise an experimental setup that is optimized for both speed and consistency. We perform systematic experiments to determine the best tradeoff between consistency of the results and speed. We also show that experiments performed on our setup generalize well to heavier scenarios. In our second contribution we show that irace is a valid method for automatically configuring a database with little manual intervention, and we show how to obtain speedups of up to 30% with respect to the default configuration with a relatively limited number of evaluations. Then, we can analyze the configurations obtained in the various cases, to give indications about how to perform future configuration tasks. Our experimental setup is provided as part of the Supplementary Material. Our approach is completely general, and can be applied to different databases with little effort. In this work, however, we study the performance of irace on the Cassandra database, and use the YCSB benchmark to measure the performance in terms of throughput [11, 53, 13].

In the next section we review existing automatic approaches to configure databases. In Section 3 we describe Cassandra and YCSB, together with the irace configurator and the general experimental setup. In Section 4 we report our experiments to find a consistent experimental setup and to evaluate irace as configurator for databases. We then conclude outlining future research directions in Section 5.

2 Background and literature review

In this section we briefly review the parameter tuning problem and the methods proposed in the literature to automatically configure database systems, with their limitations. We can classify the existing works in two main categories, model-based and search-based methods.

2.1 Parameter tuning

The task of parameter tuning, algorithm configuration, or (in machine learning) hyperparameter tuning consists in finding the configuration of a given software or algorithm that can obtain the best results for a given scenario [8]. Formally, for an algorithm $\mathcal{A}_\mathcal{P}$ for a certain problem $\mathcal{P}$, with k parameters $\Theta = \theta_1, \theta_2, \ldots, \theta_k$, a set of instances $\mathcal{I}$ of $\mathcal{P}$, and a measure of the performance $m : \Theta \times \mathcal{I} \mapsto \mathbb{R}$ to be (without loss of generality) maximized, the goal is to find $\Theta^* = \arg\max m(\Theta, \mathcal{I})$.

It is a stochastic black-box optimization problem that arises in many fields where the performance of a certain algorithm of software is crucial, and can be controlled by the user with the algorithm or software parameters. Databases are an example where this task is clearly relevant. They come with a set of parameters that impact their performance at various levels, such as the size of caches and buffers, or the number of concurrent operations allowed.

Parameter tuning is a hard problem in practice. It involves a mixed space of variables, as parameters can be of integer type, real-valued, categorical, representing unordered alternative choices, or ordinal, where a relationship between the parameter values can be established (for example, a parameter may take a value among {low, medium, high}). Parameters can also be dependent on each other, where for example parameters p_a is used only if another parameter p_b takes a certain value; and even when this is not the case, parameters interact in ways there are difficult to understand even for domain experts. A tuning is also strongly dependent on the scenario, such as the metric we use to evaluate the configurations, or the experimental setup.

We distinguish between *offline* parameter tuning, where the goal is to find one configuration of parameters to apply when first deploying the database, and *online* parameter tuning where instead the task is to modify the configuration at runtime, to adapt the database to a changed scenario (e.g. a different workload).

2.1.1 Scope of parameter tuning for databases

The use of artificial intelligence (AI) techniques has been increasingly replacing traditional methods based on human effort and heuristics in the deployment and management of database systems. In [61], AI techniques for databases have been classified in five broad areas, three of which are related to the optimization of the performance. These are database configuration, database optimization, and database design, listed in increasing order of complexity.

Database configuration refers to the fine-tuning of a certain set of operations to improve the performance. This includes the configuration of the database parameters [62, 52, 59, 26], but also the index selection [51, 42, 37] and view advisor [64, 21, 57] that optimize the performance when accessing the data, and the query rewriting that manipulates the queries provided by the user in order to make them more efficient [30, 12, 4].

Database optimization involves several components of the database at the same time. It includes for example cardinality and cost estimation [17, 32], that is, estimating how many rows will be accessed when executing a query, and how many resources will be used, join order selection [44, 23], to find the most efficient way of performing a set of join operations.

AI-based techniques for database design can instead be used to assist or replace human operators when designing a database, exploring a wider set of alternatives than what could be possible in a manual task. This includes using machine learning algorithms to learn indexes and data structures [22, 56]. Another important application is learning the patterns of transactions, in order to predict future operations and schedule them more efficiently [29, 43].

In this works we consider (offline) parameter tuning, one of the basic of database configuration. Before presenting our approach in detail, we first review the existing techniques proposed in the literature.

2.2 Model-based methods

The first methods proposed for automatic database configurations tried to devise an explicit relationship between database parameters, or a subset of them, and the database performance. Kwan et al. explicitly provide a mathematical formulation of this relationship for DB2, and suggest an initial configuration [25]. It is however unclear from their work if they consider all the parameters, or only a subset. Sullivan et al. devise an influence diagram involving four parameters of BerkeleyDB, and recommend a plausible interval for those parameters [47]; Dias et al. propose an analogous approach for the Oracle database [15]. The main limitation of these works is the impossibility to include in the analysis all the possible factors that impact the performance, such as the hardware configuration, the data, or the workload.

Identifying the parameters that affect the performance the most is a very important task for both obtaining an efficient database and being able to interpret the results. Several methods based on statistics and design of experiments have been proposed to this task. Oh and Lee use the Pearson correlation coefficient to determine the four parameters that impact the memory efficiency of Oracle the most [35]. Debnath et al. use Plackett-Burman design to rank the parameters of PostgreSQL according to their importance [14]. Babu et al. use adaptive sampling to tune two parameters related to the memory use of PostgreSQL [3]. Duan et al. combine Latin Hypercube Sampling and Gaussian Processes to configure a database, and provide the first experimental results of a method on two different databases [16]. n particular, they configure up to eleven parameters for PostgreSQL and MySQL. In 2017, Van Aken et al. instead propose a model based on Gaussian Processes and gradient descent to configure the parameters of PostgreSQL, MySQL and ActianVector [52].

Parameters can also be altered while the database is in operation. Storm et al. use cost-benefit analysis and control theory to tune and adapt the buffer pool size of DB2 [45]. Tran et al. apply memory replacement policies based on an analytical model to configure four parameters that impact the buffer size of PostgreSQL [50]. Fuzzy logic has been also used to configure in an online fashion the buffer parameters of Oracle [58, 55].

Following their successes on other fields, (deep) neural networks have been used also for database configuration. Rodd et al. and Zheng at al. both use neural networks to tune parameters related to the memory management in Oracle [41, 60]. Tan et al. propose an analogous approach for online buffer tuning of PolarDB-X [49]. Maghoub et al. use deep learning to generate a surrogate model of the performance of the database, and combine it with a genetic algorithm to configure at runtime both Cassandra and ScyllaDB, identifying the most important parameters to which focus on [31]. Zhang et al. and Li et al. both apply deep reinforcement learning to configure MySQL, PostreSQL and MongoDB [59, 26].

2.3 Search-based methods

While several model-based approaches have been proposed, not as many model-free works exist in the literature. Zhu et al. propose two dabatase-independent methods, the first one combining the divide-and-diverge sampling method with the bound-and-search algorithm, and the second one combining the Latin Hypercube Sampling method with the Random Search algorithm to configure the parameters of several databases, both relational and NoSQL, including a big data one [62, 63]. These methods, however, are very complex and not easy to implement.

2.4 Discussion

Improving the performance of a database by configuring its parameters is an approach that has received significant attention in the last years, similarly to analogous works in thriving fields such as machine learning. In fact, the majority of the works we have collected rely on some statistical modeling of the relationship between the parameter configuration and the performance of the database.

However, model-based methods have some significant limitations, that hamper their applicability on different scenarios [28]. Simple methods can be used to configure only few parameters, while more complex approached require large amounts of data to be effective. Unfortunately, as we are going to see in Section 4.1, a consistent experimental setup is quite computationally expensive, yet it is necessary to obtain reliable measurements. Modeling an explicit relationship between parameters and performance omits relevant factors such as the hardware configuration used; and including those factors is likely to make the model complex to implement, cumbersome to train and difficult to understand. Finally, the performance measured after training a model on a specific scenario cannot be expected to apply to different scenarios, such as different data, different workloads, or a different hardware configuration [10, 24].

On the other hand, the search-based methods proposed do not depend on a specific database, but are also complex ad-hoc methodologies. However, the configuration of a database is conceptually no different than the configuration of any parameterized algorithm or software, a problem that has received significant attention in the last years [9, 6]. In particular, a lot of interest followed the explosion of machine learning methods and the recent trend in automated machine learning, that includes (hyper-)parameter optimization among its tasks [19, 7, 20, 27].

We therefore follow a third approach, to use an off-the-shelf, easy-to-use, general purpose tuner, namely the irace configurator, described in the next section.

3 Materials

3.1 The Cassandra database

Cassandra is a distributed NoSQL database designed to handle large amounts of unstructured data [11, 53]. Initially designed at Facebook by combining some characteristics of Google's Bigtable and Amazon's Dynamo, Cassandra has then become one of the most popular NoSQL databases. Cassandra has a wide-column data store model, using columns as the basic data unit and the structuring the data following the concept of column families.

One of the most important characteristics of Cassandra is its peer-to-peer distribution architecture where each node acts as both master and server, without a single point of failure. All the operations can therefore be performed in a decentralized manner, and the nodes periodically exchange information on their status. This architecture, coupled with a fault-tolerant writing process and the replication of the data, grants a high service availability.

Cassandra provides a SQL-like language called Cassandra Query Language (CQL). Using this language it is possible to create and update the schema of the database, and access the data.

We use Cassandra for two main reasons. First, Cassandra is one of the most studied and best performing NoSQL databases today. Second, the existing documentation is very complete, and it allows to easily replicate and generalize the experiments carried out in this work.

3.2 The YCSB benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) is one of the most popular benchmarks to assess the performance of relational and NoSQL database management systems [13, 2, 1]. This tool consists of two parts: the record generator and the workload generator. The record generator allows to define the characteristics of the data stored in the database, specifying for example the length and type of the records and the distribution of the data when inserted. In this way it is possible to replicate a specific database scenario, by using the same kind and amount of data of real cases, without using the actual data.

The workload generator defines the quantity and type of operations to be executed. Like for the record generator, it is possible to replicate real-world cases by specifying the proportion of different operations from the following set: (i) read, which gets the value contained in a record, (ii) update, that overwrites the value contained in a record, (ii) scan, that reads all or a subset of the records in a table, (iv) insert, which adds a new record to the database, and (v) read-modify-write, which alters the value of a record by executing three successive operations as an atomic one. While the user can specify any workload, YCSB has a set of six default workloads.

- Workload A has a 50/50 read and update ratio.
- Workload B is a "read mostly" workload, with a read and update ratio of 95% and 5% respectively.
- Workload C is a read-only workload.
- Workload D is a "read latest" workload. This workload has 95% read and 5% insert operations, and the most recently inserted records are the most likely ones to be read.
- Workload E has 95% scan and 5% insert operations.
- Workload F is a read-modify-write workload. Half of the operations are read, while the other half consist in read-modify-write operations.

For different applications, it may be possible that some records are accessed more frequently than other ones. Thus, in YCSB it is possible to select the most probable distribution of the records that will be accessed by the operations. The three options are: (i) uniform, where each record has the same probability of being accessed; (ii) Zipfian, that increases the probability of accessing records selected in the past, and (iii) latest, that increases the probability of accessing the most recently inserted records. We use the default policy of YCSB, the Zipfian distribution, for the majority of the operations. The exceptions are Workload D, where a latest policy is used, the scan operations, that select a subset of the records in a table with a uniform distribution. In YCSB it is possible to run operations in parallel using multi-threading.

The use of YCSB has several advantages. First, it is possible to recreate various situations, by using alternative combinations of data and workload specifications. Second, using YCSB it is possible to easily generalize our approach to different databases. Finally, YCSB generates a comprehensive report, than can be used to analyze the status of the database.

3.3 The irace configurator

We use the general purpose configurator irace, an implementation of the *iterated racing* algorithm freely available as an R package [27, 9]. Initially developed as a tool for optimization algorithms, it has been applied also to different domains, such as machine learning, automatic algorithm generation, or the configuration of the GCC compiler. In this work we investigate its efficiency on the task of configuring a database; the peculiar aspects of this task will be presented in Section 4.1.

To perform a tuning, irace needs a *target algorithm*, its list of parameters with their type and possible values, and a set of *instances* to benchmark the configurations. In our case, the algorithm will be the Cassandra database, and the instances will be sets of database operations as defined by different YCSB workloads.

The basic routine of irace, called *race*, can be thought as a competition to select the best configuration among a set of candidate ones [33]. The candidate configurations are evaluated on the same benchmarks, and a statistical test is used to eliminate from the race the ones performing poorly. The rationale for this process is that evaluations are expensive, and it is better to focus the limited budget of evaluation on the most promising candidates, rather than wasting it on poorly performing ones.

More precisely, irace starts with a set of configurations generated uniformly at random (possibly including some configurations provided by the user, such as a default configuration); each configuration is benchmarked on a sequence of instances (in our case, of workloads) until a certain amount of results have been collected for each configuration. At this point, a statistical test identifies the worse configurations, and removes them from the lot. The race continues repeating this process of evaluation and statistical elimination, until either the budget of evaluation for the race is exhausted, or a minimum number of configurations remain. The surviving configurations are then used as seeds to generate a new set of candidate configurations around them, to be benchmarked with a new race. This process iterates until the total amount of experiments is used. The process is represented in Figure 1. For a more detail description of irace, we refer to [27].

We choose irace for several reasons. First, its efficiency has been demonstrated on several different scenarios. While developed primarily for the tune of optimization algorithms, irace has been successfully used also to configure machine learning algorithms, optimize code compilers, and automatically design

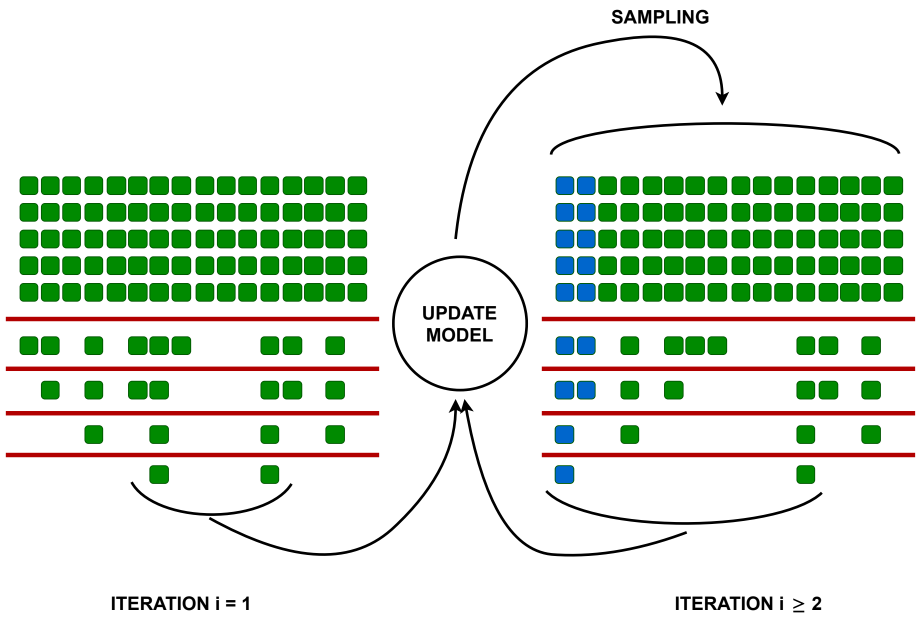


Figure 1: Representation of the racing process of irace. Each column is a different configuration, and each row is an instance evaluated. After a certain amount of instances, the algorithm discards the configurations that perform statistically worse than the best one, according to a Friedman analysis of ranks test (corresponding to the red lines). Instance after instance, the algorithm proceeds to benchmark surviving configurations, and discard the ones that perform poorly, until either the experiment budget for the race is exhausted, or a minimum number of surviving configurations remain. These surviving configurations, called *elites*, are used to update the model, to sample new candidate configurations around them. The elites (in blue) and the new configurations are evaluated in a second race. This process iterates until the total experimental budget of irace is consumed.

new state-of-the-art optimization algorithms with several hundreds of parameters [34, 38, 36]. Second, its ease of use makes its use suitable also to non-experts. Being a black-box optimizer, irace does not require any insight about the inner working of our target algorithm. Therefore, we can seamlessly apply it to configure Cassandra, but also any other database we should need to tune. In this regard, irace has an advantage over model-based methods, as it does not need to assume statistical models about the impact of the parameter configuration, nor large amounts of data to train them.

3.4 General experimental setup

We run our experiments in the Google Cloud infrastructure using `n1-standard-8` machines, with 8 virtual CPUs, 30GB of memory and a 20GB persistent disk.

We use Cassandra version 3.6, that exposes 23 parameters that can impact the performance of the database, of categorical, integer, and real-valued type. The list of parameters and the ranges considered is reported in Table 1. Using YCSB version 0.17, for each experiment we create a table with ten `varchar` fields of 100 bytes each.

The measure we use to evaluate the configurations is the throughput, that is, the number of transaction per second: a good configuration yields a higher throughput than a poor one, so our goal is to find the configuration that maximizes the throughput.

The version of irace is 3.4, with budgets of 500, 1000 and 2000 experiments per tuning. Real-valued parameters use a precision of 2 decimal digits. An instance is a pair (YCSB workload, random seed). The statistical test used to discard poor-performing configurations is the Friedman test. The additional details on the setup are determined experimentally and discussed in Section 4.1.

The basic setup of irace consists of three main components: (i) the parameter file, that contains the definition of the Cassandra parameters we configure; (ii) the scenario file, with the configuration of irace, including the tuning budget; and (iii) a script called `target-runner` that performs an atomic evaluation of Cassandra with a given configuration. The `target-runner` takes in input the candidate parameter configuration and the instance, evaluates it, and returns the throughput recorded. Optionally, a file containing user-defined configurations can also be provided: we use this file to include the default Cassandra configuration in some of our experiments. A full experimental setup to reproduce our experiments is included in the Supplementary Material.

Following the guidelines for evaluating a database, we can claim that our evaluation is fair [40]. We test the same database, under the same benchmark, on the same hardware. The significance of the evaluations is determined experimentally. Our goal is to evaluate the potential of a general purpose configurator to find configurations that improve over the default one, so each test is performed in the same way, and we report statistical comparisons of the results.

3.5 List of experiments

Here we briefly introduce the list of experiments performed. In Section 4.1 we complete the experimental setup, by determining experimentally how much data and how many operations should be used, and what is the impact of some choices about how experiments are run on the results obtained. The goal is to obtain a setup that obtains consistent and scalable results, in the shortest possible time. Since a tuning consists in a sequence of evaluations performed under partially different conditions, the assurance about the meaningfulness of the results is even more important than the throughput obtained.

In Section 4.2 we evaluate irace as a configurator for Cassandra. We consider different scenarios: (i) the six default YCSB workloads, (ii) YCSB Workload A, and (iii) YCSB Workload E. With the first one, we try to find a configuration that can improve over the default one for several cases at the same time. With the second and third scenario, we want to observe what is the impact of focusing on a specific case, respectively one for which Cassandra is well suited, and one where Cassandra struggles. Our set of scenarios is representative of different real world cases, where the user needs either flexibility over several possible scenarios, or, on the contrary, the user knows precisely the load of the database and needs a configuration tailored for the specific case. We test the potential of irace with increasing tuning budget, on the whole set of parameters and on a restricted one, and observe when and how it is convenient to exploit the high quality of the default configuration. Each configuration obtained is also applied to a heavier load, to test the validity of our experimental setup, and the general applicability of the tuning process.

Finally, we analyze the configurations obtained, and discuss them in relation with the default configuration and the scenario for which they were obtained.

4 Experimental results

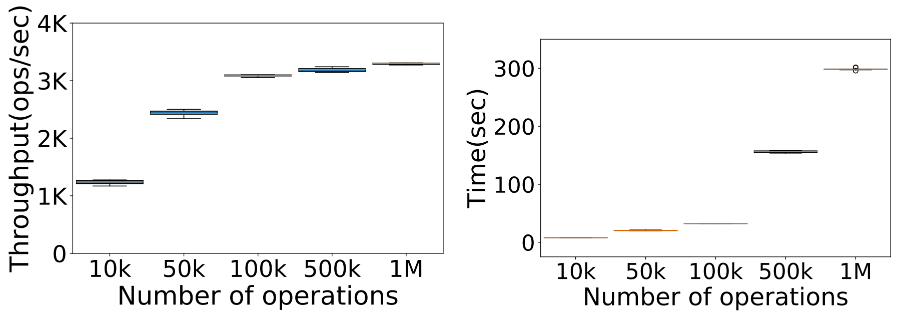
4.1 Finding a reliable experimental setup

Finding the best configuration of a database for a given scenario typically requires several experiments, each one requiring to benchmark the database with a different configuration for some time. These experiments are quite computationally expensive, and we usually cannot or do not want to run each of them at length. Unfortunately, short experiments cannot be considered reliable, as they are likely to give different results when repeated identically.

When configuring an optimization or machine learning algorithm, very often we can afford to run several experiments. In this case, however, each experiment takes a relatively long time, and it is more difficult to parallelize, so we need to use as few experiments as possible.

The first step in configuring a database is therefore to devise a minimal experimental setup that will allow users and practitioners to obtain consistent results. There are in fact several factors that impact the performance of a database. Some of these factors, such as the machine or the workload, are usually part of the problem definition. The parameter configuration of the database that yields the best performance for the given scenario is the factor we want to intervene on. There are however other possible spurious factors that impact our observation of the performance, caused by the intrinsic inner working of the database. In fact, our measurement of the throughput, or some other performance measure of choice, will also be affected by how much data we use, for how long we run our benchmark to measure the performance, or how we configure our machine.

Here we study these spurious factors and how to mitigate their impact, in order to devise an experimental setup that can obtain consistent results. A full



(a) Throughput (b) Time

Figure 2: Throughput obtained with 10K, 50K, 100K, 500K and 1M operations, with 100k rows of data, and the time each experiment takes. The results reported represent the performance of Cassandra measured by testing the default configuration ten times on workload A.

factorial analysis of the possible experimental setups is clearly infeasible, so we break this task down into five basic questions.

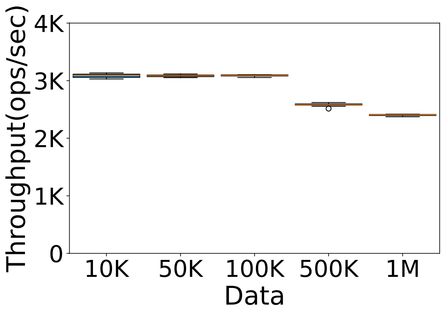
4.1.1 How many operations to use?

The first factor we consider is the number of operations, that is, the minimum number of operations necessary to do a representative experiment. Here we test the default configuration of Cassandra with the YCSB workload A and a fixed amount of $100K$ rows of data in the database, varying the number of operations. Each evaluation is repeated ten times. We seek to find the minimum amount of data that gives results representative of higher amounts.

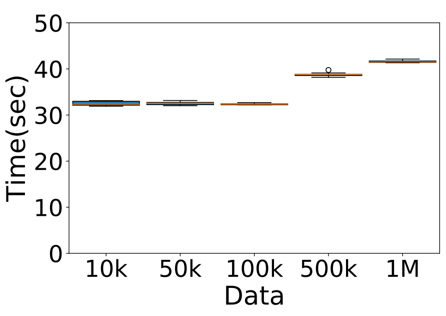
In Figure 2 we show the throughput obtained with 10000, 50000, 100000, 500000 and one million operations, and the time it takes. Clearly, 100K operations are the best choice, obtaining a throughput that is both similar to the one of higher amounts of operations and of low variance, in a time similar to the time it takes to run 50K operations. Additional experiments with a one million rows of data, and with number of operations ranging from 10K to 100K at intervals of 10K are reported in the Supplementary Material, and confirm that 100K is indeed a good value, for which we obtain results both consistent and representative of heavier loads, in a relatively short time.

4.1.2 How much data to use?

The next step is to determine the proper amount of data, in terms of rows in the database. We use again the Cassandra default configuration on workload A. In Figure 3 we show the throughput and relative time for 10000, 50000, 100000, 500000 and one million rows. The results indicate that using 100K rows of data we can obtain a good and consistent throughput, for the same time of lower



(a) Throughput



(b) Time

Figure 3: Throughput obtained with 10K, 50K, 100K, 500K and 1M rows, with 100K operations per experiment, and the time each experiment takes. The results reported represent the performance of Cassandra measured by testing the default configuration ten times on workload A.

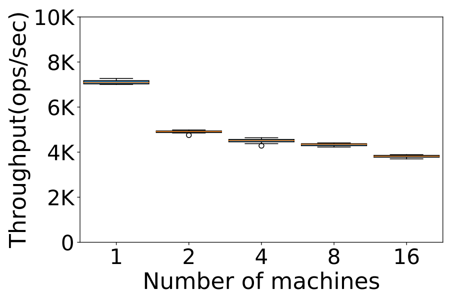


Figure 4: Throughput obtained with 100K rows and 100K operations per experiment on 1, 2, 4, 8 and 16 machines. The results reported represent the performance of Cassandra measured by testing the default configuration ten times on workload A.

amounts of data.

4.1.3 How many machines to use?

The number of machines we use in our experiments is another very important factor to consider to establish the framework where we run the experiments. We use the default configuration on Workload A, and 100K rows and operations. We evaluate the performance on 1, 2, 4, 8 and 16 machines. The throughput obtained is reported in Figure 4.

The best results are obtained using a single machine. This is perhaps surprising, considering that Cassandra is a distributed database designed to linearly increase its performance along with the number of machines, but we note how our observations are consistent with what reported by other authors. Haugh-

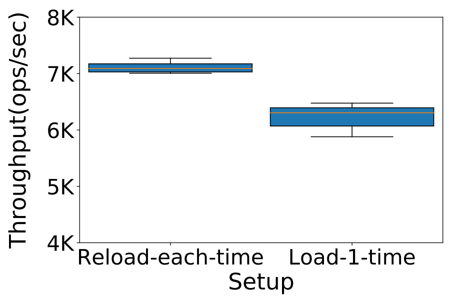


Figure 5: Throughput obtained with 100K rows and 100K operations per experiment by loading the data only once without deleting it between experiments, and destroying and reloading the data for each run. The results reported represent the performance of Cassandra measured by testing the default configuration ten times on workload A.

ian et al. reports that Cassandra's replication models degrade in both performance and consistency when more than one machine is used, in different cluster sizes and different workloads, due to Cassandra's multi-master architecture [18]. Wang et al. also argue that a high replication factor in Cassandra can affect the performance [54]. Finally, Swaminathan concludes that for databases of the moderate to large sizes the excessive distribution of data over different nodes and the overhead associated with the communication protocol can lead to a decrease in the performance of Cassandra [48]. The results we obtained can also be in part explained by the use of a workload with a large portion of read operations, while a more write-heavy workload would probably benefit from a higher number of machines [18].

On the other hand, we note that the absence of a single point of failure of a distributed architecture renders the database more robust, and may still be preferred in critical applications.

4.1.4 Is it better to reload the data base each time?

Another factor that can influence run performance is how data is loaded between runs. With these experiments we want to observe what effect this has on the configuration task, where several different configurations have to be evaluated sequentially. In Figure 5 we observe the impact on the throughput of the reloading the database each time for the Cassandra default configuration, on a database with 100K rows and operations, on a single machine. The ten replications obtain more consistent results when destroying and reloading the database for each experiment.

The effect of parameters that cannot be altered at runtime has another consequence. In Figure 6 we observe the results obtained by nine different configurations, chosen according to a design of experiment strategy to cover the parameter space as much as possible, on the same experimental setup, when

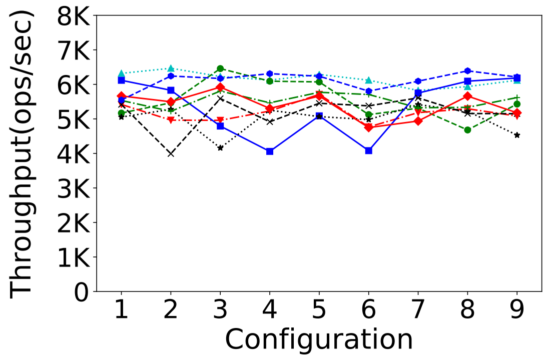


Figure 6: Throughput obtained with 100K rows and 100K operations per experiment. The results reported represent the performance of Cassandra measured by running a series of nine different setups nine times, changing the order of the configurations each run, without destroying the database after each experiment.

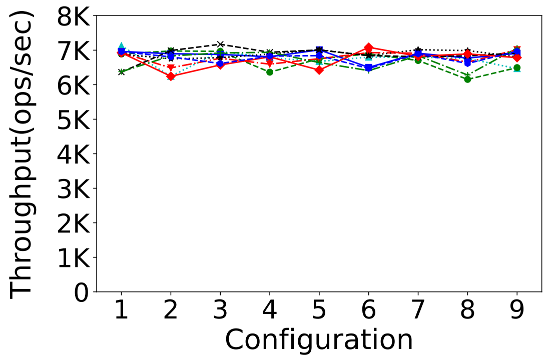


Figure 7: Throughput obtained with 100K rows and 100K operations per experiment. The results reported represent the performance of Cassandra measured by running a series of nine different setups nine times, changing the order of the configurations each run, destroying and reloading the database for each run.

ran in different order without destroying the database between evaluations. In Figure 7 instead, the same configurations are run in the same orders, this time destroying the database after each evaluation. The results indicate how the order of execution has a dramatic influence on the throughput obtained.

In this case, the destruction and reloading of the database is not merely a feature of the experimental setup that affects the performance, but it can invalidate the whole tuning process. It is therefore necessary to destroy and reload the database before evaluating every new configuration.

In database jargon, we perform only cold evaluations, that is, that include the comparatively slow startup phase [40]. In our case, however, this choice is

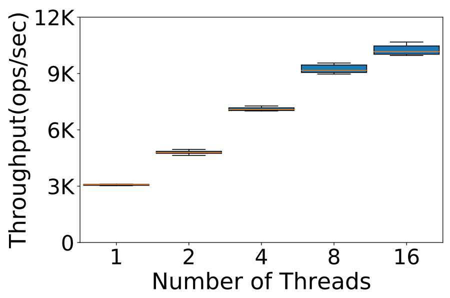


Figure 8: Throughput obtained with 100K rows and 100K operations per experiment running the experiments with 1, 2, 4, 8 and 16 threads. The results reported represent the performance of Cassandra measured by testing the default configuration ten times on workload A.

not due to the impact of the startup phase, but by the impossibility of performing a meaningful tuning task if only warm runs are executed. In fact, the infeasibility of the modification of the configuration in between runs essentially devoids of meaning the throughput we measure.

This reliability comes, however, at the cost of the time needed for each evaluation. Warm evaluations on this setup take on average 50.2 seconds to complete, while cold ones take on average 64.5 seconds, circa 28% more. Overall, however, the evaluation time remains the 77% of the total time of each experiment, and this can be considered a reasonable price to pay for, in exchange, the assurance of a valid and consistent tuning process. We also note how, by virtue of their higher reliability, cold experiments obtain a higher throughput than warm ones.

4.1.5 Does the number of users impact the results?

YCSB can simulate the use of the database from different users by executing the operations in a multithreaded fashion. So far we used 1 thread to run our experiments, but we want to test whether the performance of Cassandra can depend also on the number of concurrent threads used. In Figure 8 we report the throughput obtained testing 1, 2, 4, 8 and 16 YCSB threads with the default Cassandra configuration on Workload A, on a database with 100K rows and 100K operations. We see that Cassandra responds very well to the different number of threads. We choose to use 4 threads because of the very low variance in the results.

4.1.6 Conclusions

While the proportion of read and write operations is defined by the workload, their amount will determine the length of each experiment. Unfortunately, the impact of an inadequately low amount of operations will be overly affected by the initial loading of the database, something that can instead be considered negligible in a production setting. Likewise, for too little data the effect of caching will give unreliable observations.

The final experimental setup that we will use in the following Section is therefore composed of $100K$ rows and operations, for a database on a single machine. We use four threads in YCSB to simulate four users querying concurrently the database. Finally, the destruction and reloading of the database before every configuration evaluation will ensure a correct tuning process.

Based on our experiments, this setup is representative of heavier loads and is likely to scale well, while requiring a computational effort that is not much greater than lighter loads.

4.2 Automatic configuration of Cassandra

Having obtained a reliable experimental setup, we can proceed with the automatic configuration of Cassandra using irace. For a tuning to be effective in a production environment, the training phase needs to resemble as much as possible the conditions of the production. We therefore consider three different cases, represented by three different workload conditions.

W6 All the six YCSB workloads. With this experiment we aim to observe the capability of irace to obtain a configuration that outperforms the default one on a variety of applications.

WA YCSB workload A. This is a workload with only read and update operations, where the default configuration of Cassandra performs particularly well.

WE YCSB workload E. This is a scan-heavy workload, a case where the default Cassandra configuration does not excel.

We evaluate irace on every workload condition by configuring two sets of parameters: the whole set of Cassandra parameters that impact the performance, reported in Table 1, and the five most important parameters, as determined in [31] (in boldface in the table). In total we have therefore six tuning scenarios. We note that, using 23 parameters, the number of possible configurations is roughly 1.18×10^38. For every scenario we perform three tuning tasks with increasing tuning budget, 500, 1000 and 2000 experiments, including the default Cassandra configuration in the initial set of irace candidate configurations, and three tunings with the same budget without including the default configuration. The tuning process with irace is described in Section 3; the material to reproduce the experiments is reported in the Supplementary Material.

The final configurations obtained are tested on the same workload of the relative tuning, under the same experimental setup (100K rows in the database and operations, testing scenario TS1) and on a heavier setup (one million rows and operations, testing scenario TS2). Each final configuration is tested ten times. The results are reported in the following, in terms of speedup obtained with respect to the default Cassandra configuration under the same testing

Parameter	Type	Values
concurrent writes	i	[8, 64]
file cache size	i	[256, 2048]
memtable cleanup	r	[0.1, 0.9]
concurrent compact	i	[2, 16]
compaction strategy method	c	(Leveled, SizeTiered)
num tokens	i	[2, 256]
concurrent reads	i	[8, 64]
replication factor	i	[2, 11]
memtable heap space	i	[1024, 3072]
memtable allocation	c	(heap_buffers, offheap_buffers, offheap_objects)
row cache size in mb	i	[0, 16]
sstable open interval	i	[0, 100]
trickle fsync	c	(True, False)
inter dc stream	i	[100, 400]
key cache size	i	[0, 200]
stream throughput	i	[100, 400]
row cache save	i	[0, 120]
column index size	i	[32, 128]
compaction throughput	i	[16, 64]
memtable offheap space	i	[1024, 3072]
commitlog segment	i	[16, 64]
mem flush writers	i	[2, 16]
index summary	i	[75, 150]

Table 1: The 23 parameters that impact the performance of Cassandra. Parameters of type `c` are categorical, representing alternative choices; parameters of type `i` and `r` take, respectively, integer and real values. Parameters in boldface are the most important ones for the performance of Cassandra, as determined in [31].

conditions. A higher boxplot thus means better results, while a boxplot whose average is below 0% indicates that the relative configuration performs worse than the default one.

4.2.1 Workshop W6

5 parameters The results for TS1 are reported in Figure 9, divided by workload. Each plot includes the results obtained with and without the default configuration (boxplots D-x and ND-x, respectively, where x is the tuning budget used) using the three different tuning budgets.

In most cases irace finds a configuration that performs, on average, 5.9% better than the default Cassandra one. In general, there is not much difference between the tuning budgets 500 and 1000, but a budget of 2000 experiments can find configurations that are both better performing and that are more consistent in their results.

The throughput improves in the heavier testing scenario TS2, reported in Figure 10, where the average speedup is around 13%, albeit with higher variability. The main exception is observed for workload E, the most dissimilar from

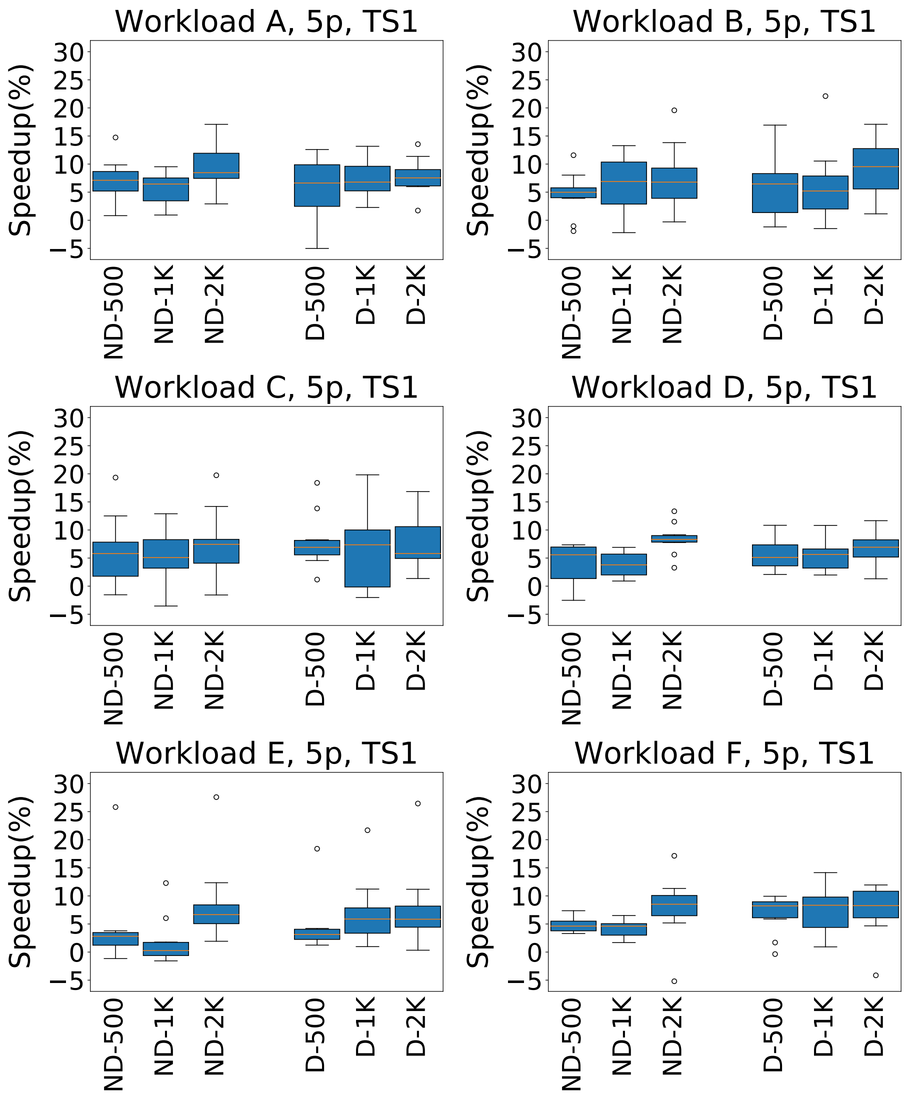


Figure 9: Speedup with respect to the default configuration obtained when tuning 5 parameters for all the six YCSB workloads (scenario TS1). The results reported represent the throughput measured testing the final configuration over each workload, repeating each experiment ten times. The boxplots report the speedup obtained by, from left to right, irace without the default configuration (ND-x, for budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (D-x, for budgets of 500, 1000 and 2000 experiments).

the other workloads, where the configurations found by irace remain below 10% of speedup. In this case, however, there is a lower variance, and the speedup is

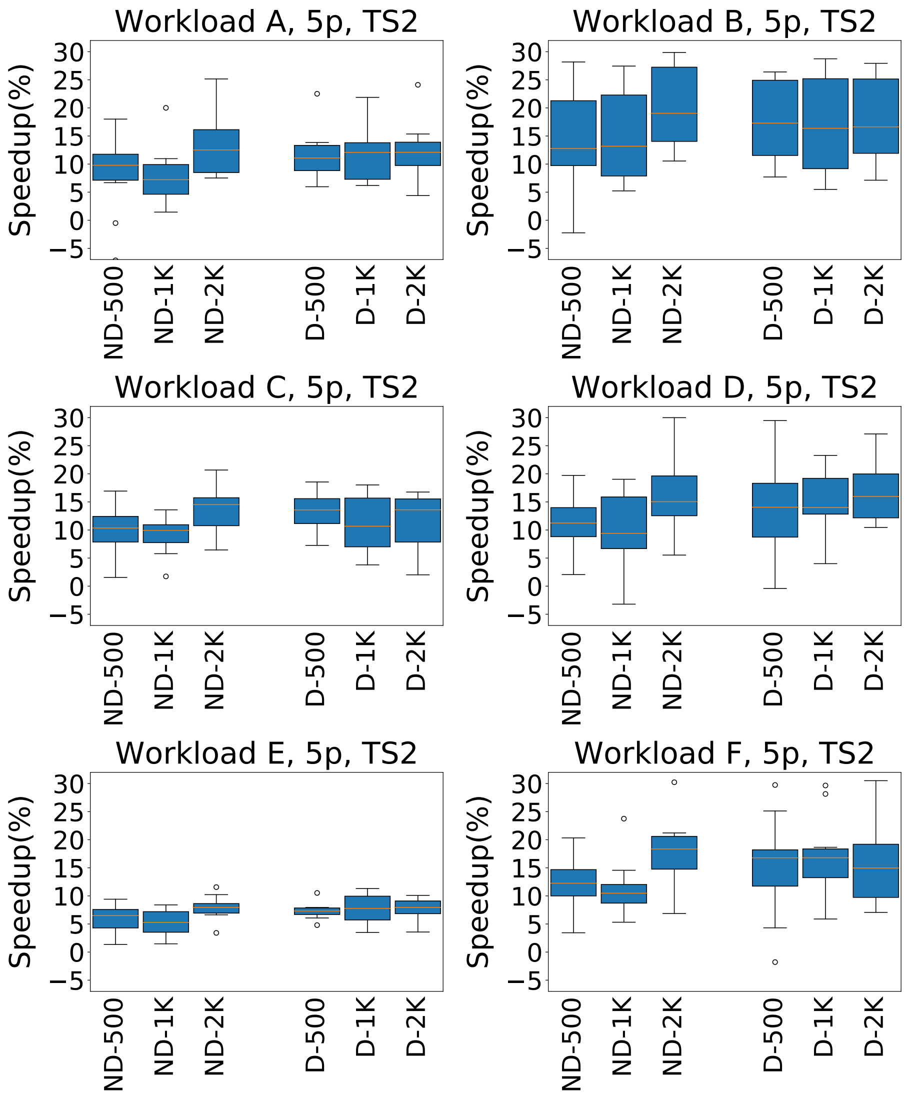


Figure 10: Speedup with respect to the default configuration obtained tuning 5 parameters for all the 6 YCSB workloads (scenario 2). The results reported represent the throughput measured testing the final configuration over each workload, repeating each experiment ten times. The boxplots report the speedup obtained by, from left to right, irace without the default configuration (budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (budgets of 500, 1000 and 2000 experiments).

consistently positive.

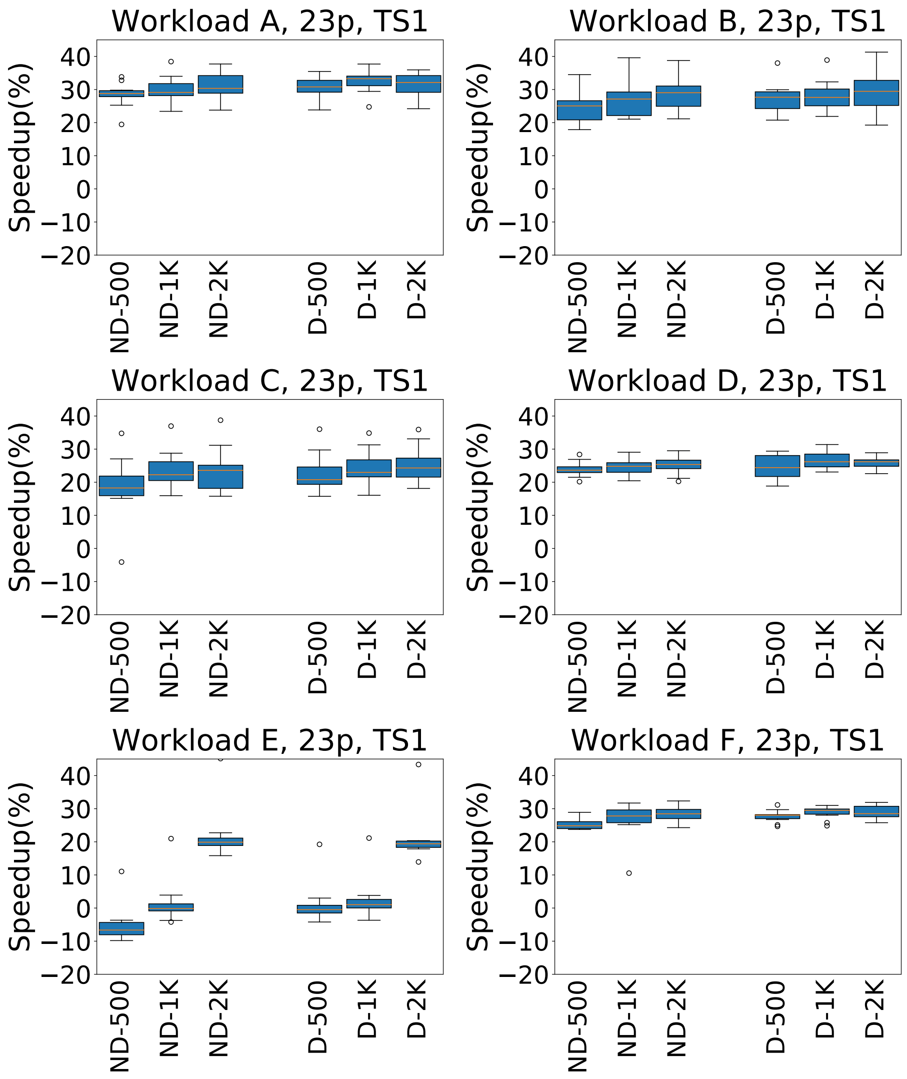


Figure 11: Speedup with respect to the default configuration obtained tuning 23 parameters for all the six YCSB workloads (scenario 1). The results reported represent the throughput measured testing the final configuration over each workload, repeating each experiment ten times. The boxplots report the speedup obtained by, from left to right, irace without the default configuration (budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (budgets of 500, 1000 and 2000 experiments).

23 parameters In Figures 11 and 12 we report the results for TS1 and TS2 respectively obtained across the 6 YCSB workloads when tuning the 23 Cas-

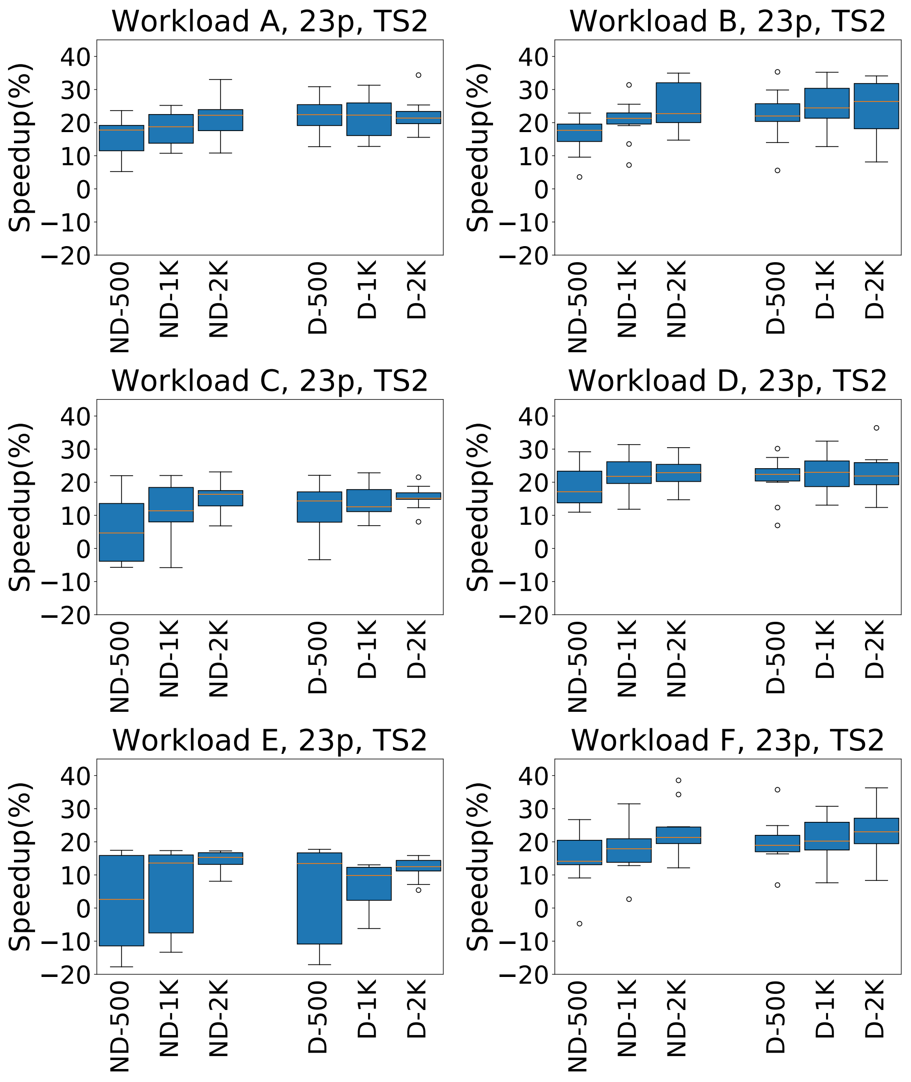


Figure 12: Speedup with respect to the default configuration obtained tuning 23 parameters for all the 6 YCSB workloads (scenario 2). The results reported represent the throughput measured testing the final configuration over each workload, repeating each experiment ten times. The boxplots report the speedup obtained by, from left to right, irace without the default configuration (budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (budgets of 500, 1000 and 2000 experiments).

sandra parameters. For TS1, on five workloads the average speedup is always between 20% and 30% over the default configuration, even for a tuning budget of 500 experiments. The only exception to this is workload E, where for the lower budgets the speedup is negligible, or even absent (in the ND-500 case); However, even for this workload the configuration found with 2000 experiments of budget obtain consistently a speedup or around 20%.

For TS2, instead, the speedup is lower, and less consistent, than for TS1, but still higher than when tuning 5 parameters. We explain this with the fact that while configuring 23 parameters allows to obtain, in general, better results than tuning only 5 parameters, this higher potential comes with the risk of overfitting. In particular, in this case the configurations obtained perform better when tested on the same amount of data and operations considered in the tuning.

Configurations The configurations obtained are represented graphically in Figure 13, and compared against the default configuration. We clearly see that there is a very diverse set of configuration obtained, and there is no value, for each parameter, that can be considered "the best" in itself. When tuning 23 parameters we observe the same effect (figure in the Supplementary Material, for space reasons). The value for the most important parameters in the extended set of parameters differs, sometimes significantly, than when tuning 5 parameters. This happens because of the interplay of the parameters, that is more important than the specific value of several of the parameters.

4.2.2 Workload WA

5 parameters The results for tuning 5 parameters over Workload A are reported in Figure 14, for both TS1 and TS2. Tuning for a single workload proves to be an easier scenario. Irace improves over the default configuration even for a budget of 500 experiments, and higher budgets result in higher improvements. As workload A is one where Cassandra performs particularly well with its default configuration, including this one in the tuning allows to obtain immediately very good results. The heavier testing scenario TS2 emphasizes these observations, with speedups up to 28.7% over the default configuration, whereas on TS1 we obtain at most an average of 12% improvement in the throughput.

23 parameters The results obtained tuning 23 parameters on Workload A are reported in Figure 15 for TS1 and TS2. In this case the results are slightly less good than when tuning five parameters. The most noticeable difference happens for ND-500, that is even on average almost equivalent (for TS1) and worse (for TS2) than the default configuration. In this case the much larger parameter space than on the 5 parameter case makes it more difficult to obtain a good configuration. The inclusion of the default configuration makes it instead possible to immediately converge the search to a good area of the parameter space, and to obtain consistently good results.

Configurations In Figure 16 we show the configurations obtained by irace when tuning 5 parameters. In the Supplementary Material we report the configurations obtained by 23 parameters. In both scenarios the configurations are

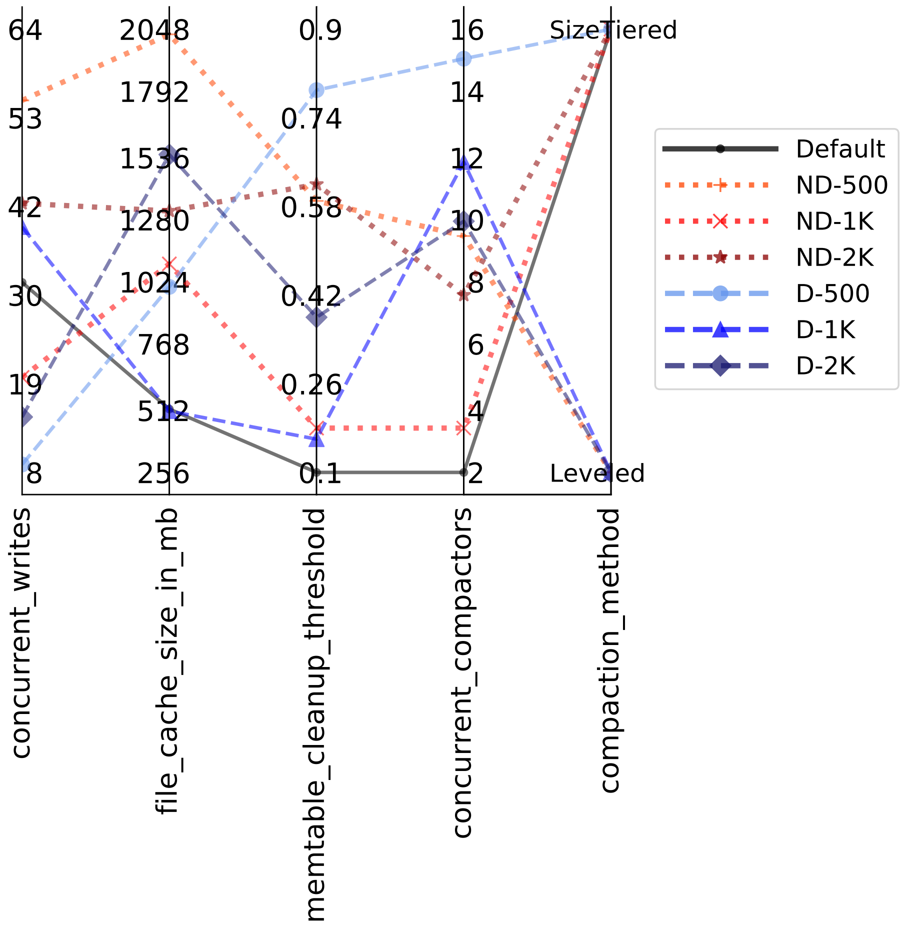
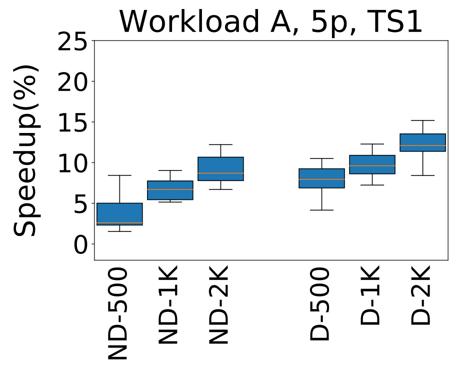


Figure 13: Parallel coordinates plot obtained tuning 5 parameters for all the 6 YCSB workloads. The reported results represent the final configuration obtained after the tuning process. The plot report the configuration obtained by, from top to bottom, the default configuration, irace without the default configuration (budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (budgets of 500, 1000 and 2000 experiments).

more uniform than in the previous case, and more similar to the default configuration. This reflects the suitability of the default configuration for scenarios with a balanced amount of read and write operations.

4.2.3 Workload WE

5 parameters In Figure 17 we report the results of the configurations obtained by irace when tuning 5 parameters under Workload E, evaluated on TS1



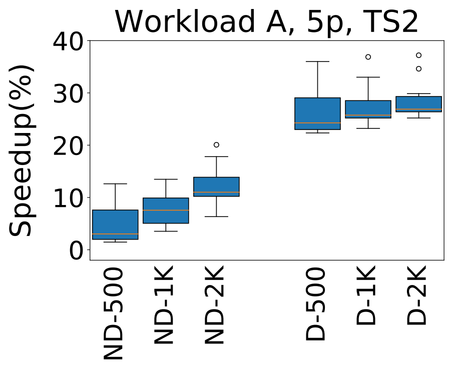
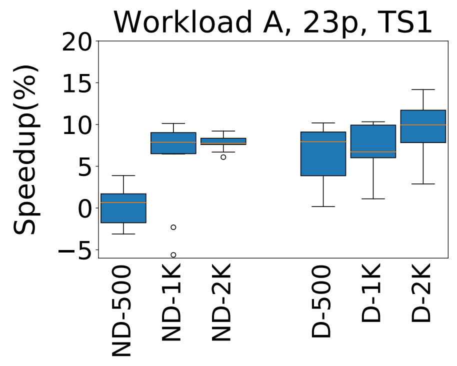


Figure 14: Speedup with respect to the default configuration obtained tuning 5 parameters for workload A in test scenario 1 and test scenario 2. The results reported represent the throughput measured testing the final configuration over workload A, repeating each experiment ten times. The boxplots report the speedup obtained by, from left to right, irace without the default configuration (budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (budgets of 500, 1000 and 2000 experiments).



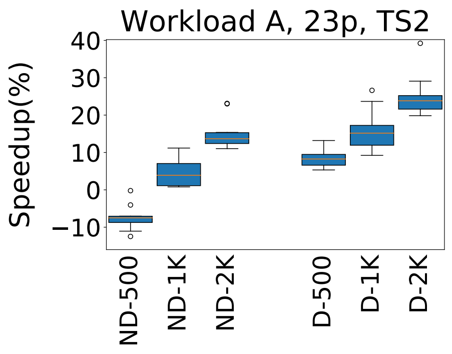


Figure 15: Speedup with respect to the default configuration obtained tuning 23 parameters for workload A in test scenario 1 and test scenario 2. The results reported represent the throughput measured testing the final configuration over workload A, repeating each experiment ten times. The boxplots report the speedup obtained by, from left to right, irace without the default configuration (budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (budgets of 500, 1000 and 2000 experiments).

and TS2. Worklaod E is one where Cassandra does not excel, and this is reflected by the fact that the configurations ND-x perform better than the D-x ones for the same tuning budget. Irace finds configurations that obtain speedups of up to 6.6% for TS1, and 12.5% for TS2. When including the default config-

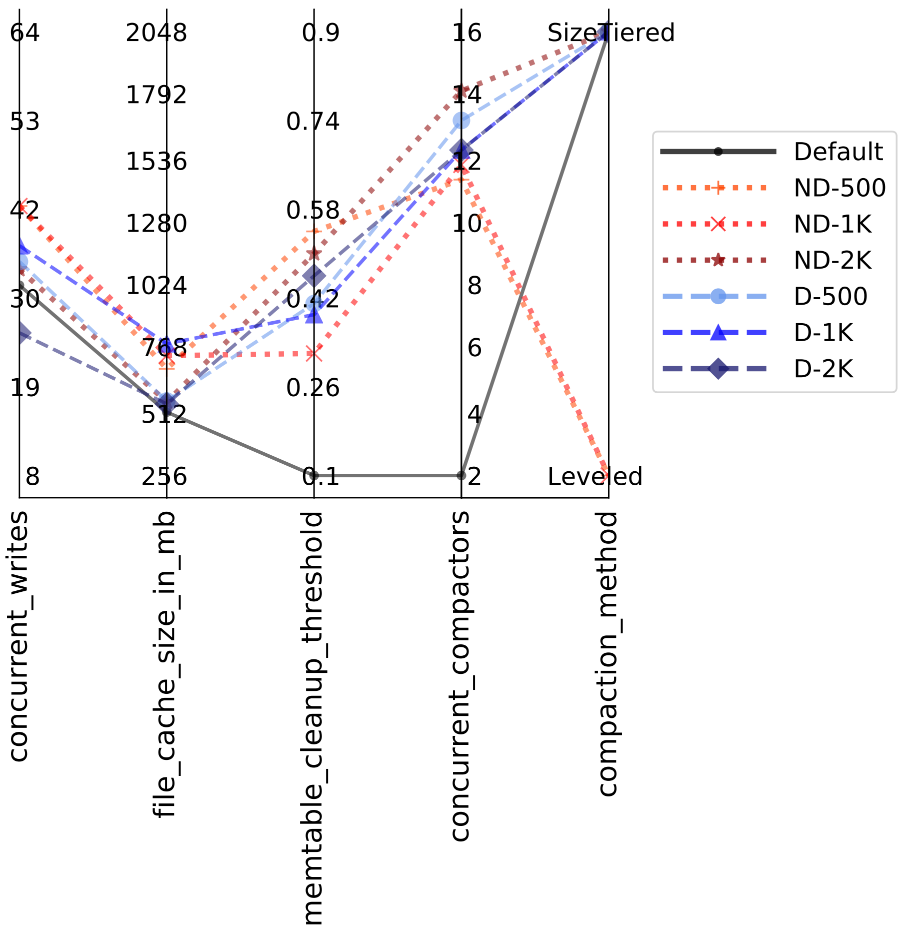


Figure 16: Parallel coordinates plot obtained tuning 5 parameters for workload A. The reported results represent the final configuration obtained after the tuning process. The plot report the configuration obtained by, from top to bottom, the default configuration, irace without the default configuration (budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (budgets of 500, 1000 and 2000 experiments).

uration in the tuning setup, it is still possible to obtain very configurations as good as in the ND-x case, but it takes a higher budget for the search to steer away from the area of relative poor quality around the default configuration.

23 parameters In Figure 18 we report the speedups of the configurations found by irace tuning 23 parameters under Workload E, tested on TS1 and TS2. In this case the inclusion of all the parameters does not result in overfitting, but

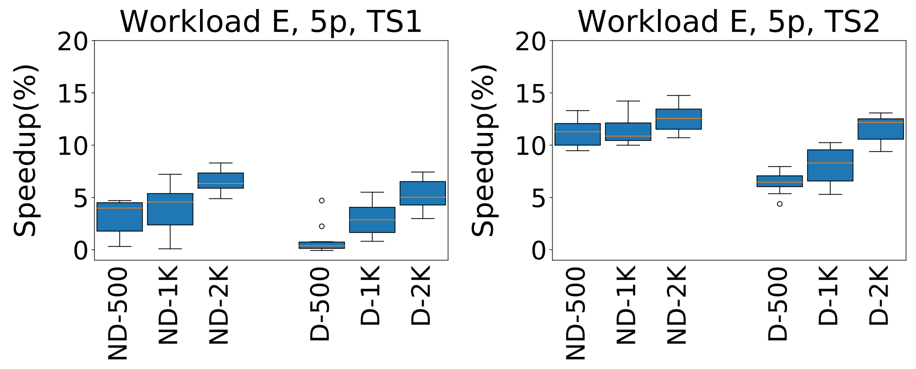


Figure 17: Speedup with respect to the default configuration obtained tuning 5 parameters for workload E in test scenario 1 and test scenario 2. The results reported represent the throughput measured testing the final configuration over workload A, repeating each experiment ten times. The boxplots report the speedup obtained by, from left to right, irace without the default configuration (budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (budgets of 500, 1000 and 2000 experiments).

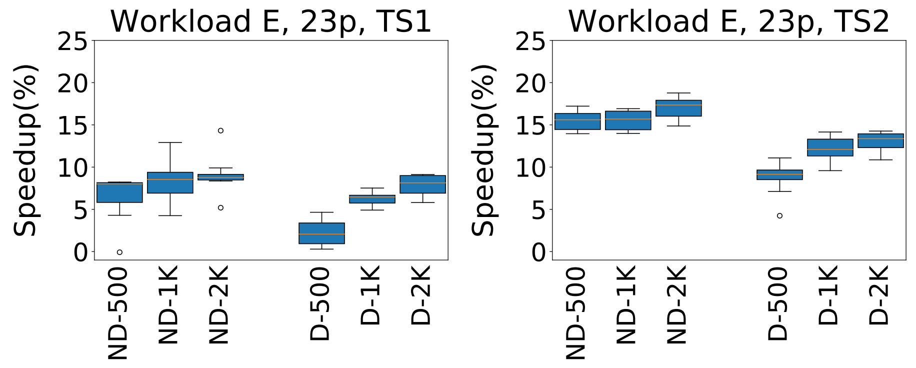


Figure 18: Speedup with respect to the default configuration obtained tuning 23 parameters for workload E in test scenario 1 and test scenario 2. The results reported represent the throughput measured testing the final configuration over workload A, repeating each experiment ten times. The boxplots report the speedup obtained by, from left to right, irace without the default configuration (budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (budgets of 500, 1000 and 2000 experiments).

makes instead possible to obtain better results than when considering only five parameters, because finding a configuration that outperforms the default one in this case is easier than for other workloads. We obtain speedups of 9.1% for ND-2000 for TS1, and 17% for the same configuration for TS2. It is to be noted

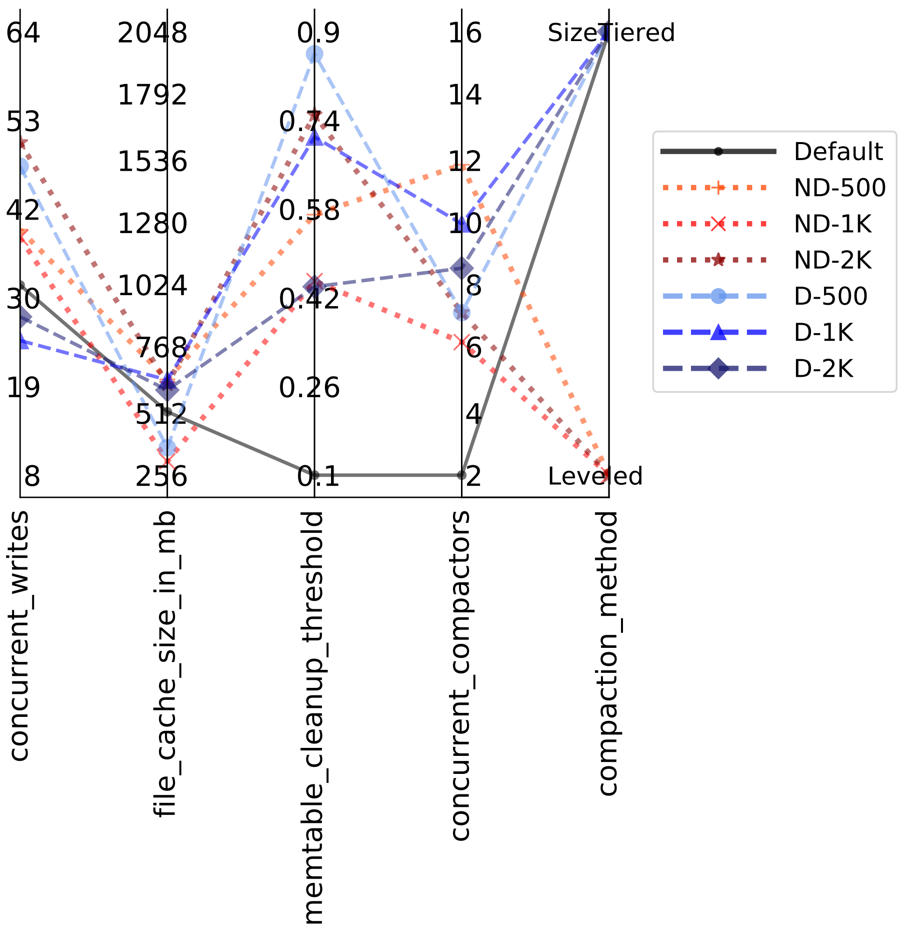


Figure 19: Parallel coordinates plot obtained tuning 5 parameters for workload E. The reported results represent the final configuration obtained after the tuning process. The plot report the configuration obtained by, from top to bottom, the default configuration, irace without the default configuration (budgets of 500, 1000 and 2000 experiments), and irace with the default configuration (budgets of 500, 1000 and 2000 experiments).

that ND-500 reports a speedup of 6.5% for TS1, which increases to 15.5% on the heavier testing scenario TS2. As when tuning five parameters, a larger budget allows configurations D-x to still obtain good results for higher budgets, even if they do not scale as much as the ND-x on TS2, reaching at most a speedup of 13%.

Setup	**W6, 5**	**W6, 23**	**WA, 5**	**WA, 23**	**WE, 5**	**WE, 23**
ND-500	8h45m	8h34m	8h54m	9h26m	10h25m	10h54m
ND-1K	18h45m	19h23m	17h50m	19h6m	21h58m	24h1m
ND-2K	41h14m	38h19m	35h57m	37h20m	44h1m	47h28m
D-500	9h21m	9h38m	8h57m	8h12m	10h40m	12h56m
D-1K	19h29m	19h20m	17h20m	17h37m	21h44m	24h5m
D-2K	37h36m	37h49m	36h3m	37h6m	44h14m	48h1m

Table 2: Time of each irace experiment.

Configurations In Figure 19 we report the configurations obtained for the workload scenario WE for 5 parameters; the analogous plot for the 23 parameters is in the Supplementary Material. Again there is some uniformity in the configurations obtained; this time, the configurations are less similar to the default configuration than in the WA scenario, but not as much on the most important parameters as on the other parameters.

4.3 Discussion

The experiments presented in this section show how irace is effective in finding parameter configurations for Cassandra that outperform the default one for single and multiple different workloads. As expected, a higher budget finds better configurations. In many cases, a budget of 500 experiments is sufficient to improve over the default configuration. This does not always happen, in particular when tuning the whole set of 23 parameters relying on the default configuration in scenarios where it is not well suited, or, on the contrary, not relying on it in cases where it is well suited. On the other hand, we a budget of 1000 experiments we always found improving configurations.

While the computational cost of each evaluations is relatively high, this is, however, a remarkably low amount of evaluations, for a set of few dozens of configurations evaluated. Very similar configurations will obtain very similar results, and in this case the noise introduced by the stochasticity will probably have a higher impact on the throughput (which reinforces the need of a proper experimental setup). There is however evidence that the parameter landscapes are somehow “easy” to traverse for automatic configurators [39], while being clearly impossible to understand for human operators.

The time taken for each tuning on the machine described in Section 3.4 is reported in Table 2. By doubling the tuning budget, the total tuning time roughly doubles too.

Whether including or excluding the default configuration from the tuning setup is beneficial for the final results depends on the workload considered. Cassandra, and its default configuration, obtain the best results for workloads with a mix of read and write operations: in these cases, the default configuration provides a good starting point that can be improved. In cases (such as Workload E) where Cassandra has lower performance, the default configuration slows down the tuning process.

Similar mixed indications come from the the comparison between the results obtained when tuning 5 and 23 parameters. In many cases the inclusion of all the parameters makes it possible to fine-tune the database in a way that is not possible by using only five parameters, even the ones with the highest impact; this is a consequence of the interaction of the parameters, as observed also by authors of previous works [16, 52, 28].

It is however possible that, in certain scenarios, the reduced search space generated by only five parameters makes it easier to find a good configuration, especially considering a limited tuning budget.

Therefore, while configurators, and generic-purpose ones such as irace in particular, make the configuration process accessible also for non-experts, a certain degree of domain expertise remains necessary to exploit the full potential of the tuning process, in particular when a limited tuning budget is used. By having reliable estimates of the data and workload, users can tailor the tuning for their specific case, in order to obtain the best results.

4.3.1 Analysis of the default configuration of Cassandra

The default configuration of Cassandra is clearly a high-quality one, that has been carefully fine-tuned over the years by the developers to perform well on several different scenarios and hardware configurations. Nonetheless, as we have seen it is possible to consistently improve over it.

The configurations obtained for the workload scenario W6 are very different from each other, yet they consistently outperform the default one, sometimes by a high margin. It is clearly impossible even for an expert database administrator to manually craft such configurations. For example, the parameter `memtable_cleanup_threshold` takes a default value of 1/(`memtable_flush_writers`+ 1), but in our final configurations we do not observe such relationship.

On more focused benchmarks, in particular WA, we can instead observe more consistent configurations, because of the narrower scope of the tuning, and some parameter values are more similar to the ones in the default configuration. For example, on WA the parameters `concurrent_writes` and `file_cache_size_in_mb` always takes values close to the default setting of 32 and 512, respectively. On the other hand, `memtable_cleanup_threshold` and `concurrent_compactors` take very different values from the default configuration. These two parameters have an effect on the performance of read and write operations, and the different values are probably caused by the specific data format used in our tests. A higher value of `memtable_cleanup_threshold` translates in larger and less frequent flushes. The different value for `concurrent_compactors` reflects the fact that we have a lot of disk space in our machines; this cannot be assumed for a generic Cassandra installation, and this is the likely reason for the default conservative value of 2.

On WE The `concurrent_writes` and `file_cache_size_in_mb` parameters take consistent values across all configurations. We note how the high range of values now taken by parameters `memtable_cleanup_threshold` and `concurrent_compactors` reflects their reduced importance in this scenario, where there are no write operations. Similarly, when not starting the tuning from the default configuration the value of the compaction method in this case is `Leveled`, which is a better choice for reading operations, while starting from the default configuration, the value remains `SizeTiered`.

5 Conclusions

In modern data-centric applications, the performance of the database is paramount, since a poor-performing database will affect the entire data processing pipeline. Finding the best parameter configuration for the database in use is therefore a very important task. Benchmarking a database configuration, however, is not a trivial task, since a balance between computational effort and significance of the experiments has to be found.

Configuring a database is far from a trivial task, due to the high computational cost of each experiment, the stochasticity of the results, the dependency from the hardware configuration and the benchmarking environment, and the intricacies arising from the internal workings of the database. While several methods have been proposed to this task, they require high amounts of data, are inadequate to model all the interactions between the factors that affect the performance of a database, or implement very complex algorithms.

In this work we have instead studied the potential of a general purpose automatic configurator, irace, to find high quality configurations for databases, and have reported experiments on different scenarios using the popular Cassandra NoSQL database. As a preliminary phase, we have devised an experimental setup that allows to obtain consistent results and is representative of heavier loads, at the minimum possible cost. We have observed how, while the performance that can be obtained obviously depends on the amounts of experiments performed, irace can find good quality configurations even with a limited tuning budget, in a huge configuration space. Thanks to the experimental setup we used, the configurations found can also scale very well to heavier loads. The results we obtained indicate than with a budget of 2000 experiments it is possible to improve for up to nearly the 30% over the default configuration of Cassandra, for selected workloads, and lower budgets can anyway be used to find good quality configurations. Finally, we have analyzes the configurations obtained, and compared them with the default configuration, to understand how the better performance could be obtained.

Aside from its efficiency, irace has several other advantages: it is freely available as an R package and it is easy to use, requiring only few scripts to run, thus unburdening users and database administrators from the need of implementing complex algorithmic solutions; and it is model-free, which makes it possible to apply it to any database, benchmark or scenario with limited effort.

Several lines of research are possible to continue this work. We plan to test irace with other databases and benchmarks. In particular, we want to use it to automatically configure entire data processing pipelines of real world applications. On one hand, this will improve the performance of entire systems, not just parts of them; on the other one, we will be able the theoretical question of whether the simultaneous configuration of a single complex system can be more effective than the separate configuration of the components of the system.

Another development is the adaptation of this methodology to dynamically configure databases, periodically changing the configuration to respond to changes in the workload.

The computational cost of each evaluation is the main hurdle for a wider deployment of automatic configurators in practical applications. Thus, we want to also explore the possibility of developing cost-aware measures that can manage the configuration by estimating the cost of the experiments performed, and the potential savings entailed by the performance increase in the production environment, and autonomously determine the extent to which a tuning should be performed. Another possibility is to develop surrogate benchmarks, that can be used to estimate the performance of a candidate configuration without actually performing the evaluation.

Acknowledgements

MS Acknowledges support from the CHIST-ERA project CHIST-ERA-17-BDSI-001 ABIDI "Context-aware and Veracious Big Data Analytics for Industrial IoT". AF acknowledges support from the Innoviris project 2018-SHAPE-25a "PHILEAS: smart monitoring par détection de comportements anormaux appliquée aux objets connectés". We thank Guillaume Levasseur and Thomas Stützle for useful discussions.

References

[1] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. Evaluating cassandra scalability with YCSB. In *International Conference on Database and Expert Systems Applications*, pages 199–207. Springer, 2014.

[2] Yusuf Abubakar, Thankgod Sani Adeyi, and Ibrahim Gambo Auta. Performance evaluation of nosql systems using YCSB in a resource austere environment. *Performance Evaluation*, 7(8):23–27, 2014.

[3] Shivnath Babu, Nedyalko Borisov, Songyun Duan, Herodotos Herodotou, and Vamsidhar Thummala. Automated experiment-driven management of (database) systems. In *HotOS*, 2009.

[4] Christopher Baik, Hosagrahar V Jagadish, and Yunyao Li. Bridging the semantic gap with sql query logs in natural language interfaces to databases. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 374–385. IEEE, 2019.

[5] Liang Bao, Xin Liu, Ziheng Xu, and Baoyin Fang. Autoconfig: Automatic configuration tuning for distributed message systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 29–40. IEEE, 2018.

[6] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24:2546–2554, 2011.

[7] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[8] Mauro Birattari. *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. PhD thesis, IRIDIA, École polytechnique, Université Libre de Bruxelles, Belgium, 2004.

[9] Mauro Birattari, Thomas Stützle, Luís Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In W. B. Langdon et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[10] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 893–907, 2018.

[11] Apache Cassandra. Apache cassandra. *Website. Available online at https://www.datastax.com/cassandra*, 13, 2014.

[12] Mahendra Chavan, Ravindra Guravannavar, Karthik Ramachandra, and S Sudarshan. Dbridge: A program rewrite tool for set-oriented query execution. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1284–1287. IEEE, 2011.

[13] 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.

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

[15] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic performance diagnosis and tuning in oracle. In *CIDR*, pages 84–94, 2005.

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

[17] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment*, 12(9):1044–1057, 2019.

[18] Gerard Haughian, Rasha Osman, and W. Knottenbelt. Benchmarking replication in cassandra and mongodb nosql datastores. In *DEXA*, 2016.

[19] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[20] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[21] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment*, 11(7):800–812, 2018.

[22] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[23] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[24] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking scalability and elasticity of distributed database systems. *Proceedings of the VLDB Endowment*, 7(12):1219–1230, 2014.

[25] Eva Kwan, Sam Lighthouse, Berni Schiefer, Adam Storm, and Leanne Wu. Automatic database configuration for db2 universal database: Compressing years of performance expertise into seconds of execution. In *BTW 2003–Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW Konferenz*. Gesellschaft für Informatik eV, 2003.

[26] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[27] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. The `irace` package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[28] Jiaheng Lu, Yuxing Chen, Herodotos Herodotou, and Shivnath Babu. Speedup your analytics: Automatic parameter tuning for databases and big data systems. *Proceedings of the VLDB Endowment*, 12(12):1970–1973, 2019.

[29] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645, 2018.

[30] Divya Mahajan, Cody Blakeney, and Ziliang Zong. Improving the energy efficiency of relational and nosql databases via query optimizations. *Sustainable Computing: Informatics and Systems*, 22:120–133, 2019.

[31] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: a middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 28–40, 2017.

[32] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019.

[33] O. Maron and A. W. Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Research*, 11(1–5):193–225, 1997.

[34] Péricles Miranda, Ricardo M. Silva, and Ricardo B. Prudêncio. Fine-tuning of support vector machine parameters using racing algorithms. In *Proceedings of 22th European Symposium on Artificial Neural Networks, ESANN 2014, Bruges, Belgium, April 23-25, 2014*, pages 325–330, 2014.

[35] Jeong Seok Oh and Sang Ho Lee. Resource selection for autonomic database tuning. In *21st International Conference on Data Engineering Workshops (ICDEW'05)*, pages 1218–1218. IEEE, 2005.

[36] Federico Pagnozzi and Thomas Stützle. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. *European Journal of Operational Research*, 276:409–421, 2019.

[37] Wendel Góes Pedrozo, Júlio Cesar Nievola, and Deborah Carvalho Ribeiro. An adaptive approach for index tuning with learning classifier systems on hybrid storage environments. In *International conference on hybrid artificial intelligence systems*, pages 716–729. Springer, 2018.

[38] Leslie Pérez Cáceres, Federico Pagnozzi, Alberto Franzin, and Thomas Stützle. Automatic configuration of GCC using irace. In Evelyne Lutton, Pierrick Legrand, Pierre Parrend, Nicolas Monmarché, and Marc Schoenauer, editors, *Artificial Evolution: 13th International Conference,*

Évolution Artificielle, EA 2017, Paris, France, October 25-27, 2017, Revised Selected, volume 10764 of *Lecture Notes in Computer Science*, pages 202–216. Springer, Heidelberg, Germany, 2017.

[39] Yasha Pushak and Holger Hoos. Algorithm configuration landscapes: more benign than expected? In *International Conference on Parallel Problem Solving from Nature*, pages 271–283. Springer, 2018.

[40] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. Fair benchmarking considered difficult: Common pitfalls in database performance testing. In *Proceedings of the Workshop on Testing Database Systems*, pages 1–6, 2018.

[41] Sunil F Rodd, Umakanth P Kulkarni, and Anil R Yardi. Adaptive neuro-fuzzy technique for performance tuning of database management systems. *Evolving Systems*, 4(2):133–143, 2013.

[42] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. On-line index selection for shifting workloads. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 459–468. IEEE, 2007.

[43] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. Scheduling oltp transactions via machine learning. *arXiv preprint arXiv:1903.02990*, 2019.

[44] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. Leo-db2's learning optimizer. In *VLDB*, volume 1, pages 19–28, 2001.

[45] Adam J Storm, Christian Garcia-Arellano, Sam S Lightstone, Yixin Diao, and Maheswaran Surendra. Adaptive self-tuning memory in db2. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1081–1092, 2006.

[46] Thomas Stützle and Manuel López-Ibáñez. Automatic (offline) configuration of algorithms. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 893–918, 2013.

[47] David G Sullivan, Margo I Seltzer, and Avi Pfeffer. Using probabilistic reasoning to automate software tuning. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):404–405, 2004.

[48] Surya Narayanan Swaminathan and Ramez Elmasri. Quantitative analysis of scalable nosql databases. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 323–326. IEEE, 2016.

[49] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. ibtune: Individualized buffer tuning for large-scale cloud databases. *Proceedings of the VLDB Endowment*, 12(10):1221–1234, 2019.

[50] Dinh Nguyen Tran, Phung Chinh Huynh, Yong C Tay, and Anthony KH Tung. A new approach to dynamic self-tuning of database buffers. *ACM Transactions on Storage (TOS)*, 4(1):1–25, 2008.

[51] Gary Valentin, Michael Zuliani, Daniel C Zilio, Guy Lohman, and Alan Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of 16th International Conference on Data Engineering (Cat. no. 00CB37073)*, pages 101–110. IEEE, 2000.

[52] 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.

[53] Guoxi Wang and Jianfeng Tang. The nosql principles and basic application of cassandra model. In *2012 international conference on computer science and service system*, pages 1332–1335. IEEE, 2012.

[54] Huajin Wang, Jianhui Li, Haiming Zhang, and Yuanchun Zhou. Benchmarking replication and consistency strategies in cloud serving databases: Hbase and cassandra. In *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pages 71–82. Springer, 2014.

[55] Zhijie Wei, Zuohua Ding, and Jueliang Hu. Self-tuning performance of database systems based on fuzzy rules. In *2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pages 194–198. IEEE, 2014.

[56] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1223–1240, 2019.

[57] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. Automatic view generation with deep learning and reinforcement learning. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1501–1512. IEEE, 2020.

[58] Gaozheng Zhang, Mengdong Chen, and Lianzhong Liu. A model for application-oriented database performance tuning. In *2012 6th International Conference on New Trends in Information Science, Service Science and Data Mining (ISSDM2012)*, pages 389–394. IEEE, 2012.

[59] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432, 2019.

[60] Conghuan Zheng, Zuohua Ding, and Jueliang Hu. Self-tuning performance of database systems with neural network. In *International Conference on Intelligent Computing*, pages 1–12. Springer, 2014.

[61] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

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

[63] Yuqing Zhu, Jianxun Liu, Mengying Guo, Wenlong Ma, and Yungang Bao. Acts in need: Automatic configuration tuning with scalability guarantees. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–8, 2017.

[64] Daniel C Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M Lohman, Roberta J Cochrane, Hamid Pirahesh, Latha Colby, Jarek Gryz, Eric Alton, et al. Recommending materialized views and indexes with the ibm db2 design advisor. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 180–187. IEEE, 2004.