Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads

Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh
Purdue University

Subrata Mitra
Adobe Research

Wolfgang Gerlach, Travis Harrison, Folker Meyer
Argonne National Laboratory

Ananth Grama, Saurabh Bagchi, Somali Chaterji
Purdue University

Abstract

High performance computing (HPC) applications, such as metagenomics and other big data systems, need to store and analyze huge volumes of semi-structured data. Such applications often rely on NoSQL-based datastores, and optimizing these databases is a challenging endeavor, with over 50 configuration parameters in Cassandra alone. As the application executes, database workloads can change rapidly from read-heavy to write-heavy ones, and a system tuned with a read-optimized configuration becomes suboptimal when the workload becomes write-heavy.

In this paper, we present a method and a system for optimizing NoSQL configurations for Cassandra and ScyllaDB when running HPC and metagenomics workloads. First, we identify the significance of configuration parameters using ANOVA. Next, we apply neural networks using the most significant parameters and their workload-dependent mapping to predict database throughput, as a surrogate model. Then, we optimize the configuration using genetic algorithms on the surrogate to maximize the workload-dependent performance. Using the proposed methodology in our system (Rafiki), we can predict the throughput for unseen workloads and configuration values with an error of 7.5% for Cassandra and 6.9-7.8% for ScyllaDB. Searching the configuration spaces using the trained surrogate models, we achieve performance improvements of 41% for Cassandra and 9% for ScyllaDB over the default configuration with respect to a read-heavy workload, and also significant improvement for mixed workloads. In terms of searching speed, Rafiki, using only 1/10000-th of the searching time of exhaustive search, reaches within 15% and 9.5% of the theoretically best achievable performances for Cassandra and ScyllaDB, respectively—supporting optimizations for highly dynamic workloads.

CCS Concepts **•Software and its engineering → Software configuration management and version control systems;**

Keywords Database automatic tuning, Metagenomics workloads, NoSQL datastores

ACM Reference format:
Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, and Ananth Grama, Saurabh Bagchi, Somali Chaterji. 2017. Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads. In *Proceedings of Middleware '17, Las Vegas, NV, USA, December 11–15, 2017,* 13 pages.
DOI: 10.1145/3135974.3135991

Middleware '17, Las Vegas, NV, USA

DOI: 10.1145/3135974.3135991

1 Introduction

Metagenomics applications, poised alongside other big data systems, have seen explosive data growth, placing immense pressure on overall datacenter I/O [13]. Automatic database tuning, a cornerstone of I/O performance optimization, remains a challenging goal in modern database systems [11, 39, 44]. For example, the NoSQL database engine Cassandra offers 50+ configuration parameter, and each parameter value can impact overall performance in different ways. We demonstrate that the performance difference between the best and worst configuration files for Cassandra can be as high as 102.5% of throughput for a read-heavy workload. Further, the optimal configuration setting for one type of workload is suboptimal for another and this results in as much as 42.9% degradation in database performance in the absence of optimized parameter versions, such as in our system Rafiki. In this paper, we present Rafiki[1], an analysis technique and statistical model for optimizing database configuration parameters to alleviate I/O pressures, and we test it using Cassandra, when handling dynamic metagenomics workloads.

Prior work and the state-of-practice have left several gaps in database configuration optimization that Rafiki addresses. First, most approaches rely on expert knowledge for significant parameter selection or simply include all parameters for tuning. For example, [25] singles out a parameter from prior work surveys (over-simplification), while [6, 7] include most, if not all, parameters, with consequent time-complexity issues. In Rafiki, we utilize analysis of variance (ANOVA [28]) to identify *key parameters* from the set of all parameters for further analysis, thus, reducing the computational complexity and data collection overheads for model training. Second, many techniques utilize optimization approaches that are vulnerable to local maxima, by making linear assumptions about the performance response of each tuning parameter [5], or rely on online tuning techniques. In practice, this results in suboptimal performance and very long wall-clock convergence times due to the overhead of performance metric collection (minutes per trial). Rafiki utilizes *trained surrogate models* for performance to enable rapid searching via stochastic models to mitigate local maxima concerns. Finally, most approaches do not account for *dynamic workload changes* when tuning. For example, [19, 42] require over 30 minutes to adapt to new workloads. Rafiki includes workload characteristics directly in its surrogate model so that large step

[1]Just as Rafiki was the wise monkey from "The Lion King" who always knew how to avoid dead ends, we wish our system to avoid poor-performing dead-spots with its sagacity.

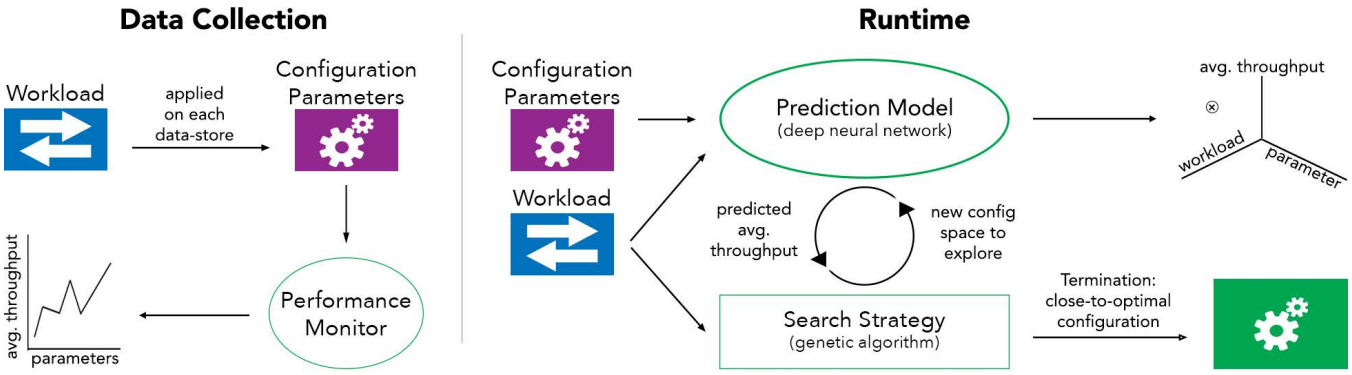

Figure 1. A high-level overview schematic of our proposed system Rafiki that searches through the configuration parameters' space of NoSQL datastores to achieve close-to-optimal performance. Rafiki is agile enough to quickly adapt to changing workload patterns. In the first phase (data collection), varying workloads and configurations are applied to the NoSQL datastore to identify the key configuration parameters and to generate training data for a surrogate model. In the second phase, a prediction model is built and used to identify the relation of configurations and workloads to performance. Finally, a search strategy is applied for a given workload to find close-to-optimal configurations.

changes in workloads are rapidly met with large step changes in configuration parameters. We find that this last piece is crucial in big data applications, such as in metagenomics, where key reuse distance is large, putting large pressures on the hard disk, and the ratio of read-to-write queries changes rapidly during different phases of its data manipulation pipeline.

We apply Rafiki to database traces from Argonne National Lab's MG-RAST[2] system, the most popular metagenomics portal and analysis pipeline. The field of metagenomics encompasses the sequencing and analysis of community microbial DNA, sampled directly from the environment. In recent years, DNA sequencing has become significantly more affordable and widespread, being able to sequence metagenomic samples more efficiently, affording the exploration of microbiomes in different ecosystems, including in the human gut [24] in different clinical manifestations, strongly motivating research in metagenomics [8]. Consequently, metagenomics analysis have come to the fore in the realm of big data systems for personalized medicine applications. Big data systems must deal with ever-growing data volumes and access patterns that are rising fast and exhibit vast changes, respectively. In MG-RAST, for example, user counts and data volumes have grown consistently over the past 9 years, and the repository today has roughly 280k total metagenomes, of which 40,696 are public, containing over a trillion sequences and 131.28 Terabasepairs (Tbp) [43], using about 600 TB resources. Such growth has placed significant pressure on I/O [40]. The system allows multiple users to insert new metagenomes, analyze existing ones, and create new metadata about the metagenomes—operations that mirror many big data applications where new data is inserted and existing data is analyzed continuously. Such processes, especially in a multi-user system like MG-RAST, result in highly dynamic database workloads with extended periods of mixed read-write activity, punctuated by bursty writes, and a dynamic mix of reads and writes during the longest periods. Such accesses are atypical of the archetypal web workloads that are used for benchmarking NoSQL datastores, and consequently, the default configurations woefully under-perform Rafiki's optimized solutions. For example, due to the volume of data and the typical access patterns in MG-RAST, key re-use distance is very large and this puts immense pressure on the disk, while relieving pressure on caches.

We, therefore, felt the need for the design of the Rafiki middleware that can be utilized by NoSQL datastores for tuning their configuration parameters at runtime. Our solution Rafiki, as shown in Figure 1, utilizes supervised learning to select optimal configuration parameters, when faced with dynamic workloads. In the first phase, the configuration files are analyzed for parameters-of-interest selection. Of the 50+ configurations in Cassandra, for example, we find that only 5 significantly impact performance for MG-RAST. To discover this, workload-configuration sets are applied via benchmark utilities, and the resulting performance measures collected from a representative server. We determine significant configuration parameters by applying ANOVA analysis to identify performance-parameter sensitivities. Even with this restricted set of sensitive parameters, the search space is large and an exhaustive search for optimal configuration settings is infeasible. For example, for our selected set of 5 parameter settings, the search space conservatively has 25,000 points. Doing this exhaustive search means running the server for a given workload-configuration combination for a reasonable amount of time (such as, 5 minutes), thus giving a search time of 2,080 hours—clearly infeasible for online parameter tuning.

In the second phase, we train a deep neural network (DNN) to predict performance as a function of workload and configuration. For data collection, we collect benchmark data for a random subset of the possible configurations. Each configuration is run against multiple workloads so that experimentally, a workload+configuration map to a performance metric. We collect a relatively sparse set of samples and utilize regularization techniques to prevent over-fitting of a DNN-based regression model. From this model, we create a surrogate for performance—given a new configuration and a new workload, the DNN will predict the performance of the database system. In the final phase, we use the surrogate model, in conjunction with a genetic algorithm (GA), to search for the optimal configuration. By the domain-specific choice of the fitness function,

[2] http://metagenomics.anl.gov

we are able to get close-to-maximum throughput for any given workload characteristic.

Rafiki is evaluated against synthetic benchmarks that have been tailored to match the query distribution of the sample MG-RAST application. Trace information from MG-RAST, measured over a representative 4 day period, was analyzed to generate accurate representations in the benchmarking tool. Additionally, some subsampling of the trace was used to measure performance in a case study. A second datastore, ScyllaDB, which is based on Cassandra, is used to show the generality of the performance tuning middleware for NoSQL datastores. Rafiki's ability to outperform the ScyllaDB's internal optimizer demonstrates its improvement over the state-of-the-practice.

Rafiki is able to increase Cassandra's throughput compared to the default configurations settings, by 41.4% for read-heavy workloads, 14.2% for write-heavy workloads, and 35% for mixed workloads. The DNN-based predictor can predict the performance with only 5.6% error, on average, when confronted with hitherto unseen workloads and with only 7.5% error for unseen configurations. The improvements for ScyllaDB are more modest because of its internal self-tuning feature—averaging 9% for read-heavy and read-only workloads. Finally, by comparing with an exhaustive grid search, we show that the performances achieved by our technique, using only 1/10,000-th of the searching time of exhaustive search, is within 15% and 9.5% of the theoretically best achievable performances for Cassandra and ScyllaDB, respectively.

The primary claims to novelty of our work can be summarized as follows:

1. We demonstrate that the performance of the NoSQL engines can change significantly with respect to the applied workload characteristics (*e.g.*, read-to-write proportions), due to workload-dependent procedures, such as compaction. Specifically, we identify that for metagenomics workloads, it is important to dynamically and quickly vary a set of configuration parameters to achieve reasonable performance.
2. We create a novel technique to predict the performance of a NoSQL engine for unseen workload characteristics and unseen configurations. This is challenging due to the non-linear nature of the dependence of performance on these and due to the inter-dependent nature of various configuration parameters. This serves as a surrogate model for us to do a search through the large space of configuration parameter settings.
3. Our middleware, called Rafiki, can search efficiently, using genetic algorithms, through the parameter space to derive close-to-optimal parameter settings, while using only 0.28% of the time of an exhaustive search. Our search is agile enough that it can be quickly re-done when the workload characteristics change.
4. We identify that the chief configuration parameters are related to compaction, such as, when to combine multiple data tables on disk into one and how many levels of tables to maintain. The relation between compaction-related configuration parameters and performance is non-monotonic, while the parameter space is infinite with both continuous and integer control variables. Our ANOVA analysis backs up these claims.

The rest of the paper is organized as follows. In Section 2, we cover the fundamental pieces of the optimization problem along with MG-RAST. In Section 3, we describe the pieces of our solution (Figure 1). Section 4 contains the implementation and analysis of our approach. Section 5 covers related work and separates our approach from existing art and Section 6 provides the conclusion.

2 Background

This section covers background material useful to understanding the method by which we improve database performance.

2.1 NoSQL datastores

NoSQL datastores are key value stores that have begun to replace traditional, transactional database systems (MSSQL/MySQL/PostgreSQL) due to performance gains from the relaxation of transaction requirements, *i.e.*, the ACID properties [9], promising scalability beyond what is possible in transaction-based SQL systems. Since metagenomics and many other big data applications can tolerate a certain degree of lack of consistency, we can instead prioritize availability and partition-tolerance [23]. For example, having a slightly outdated copy of a genome may result in less pattern matches, but the matches that are found are still valid results.

In this paper, we focus on Cassandra, an Apache Foundation[3] project, which is one of the leading NoSQL and distributed DBMS driving many of today's modern business applications, including such popular users as Twitter, Netflix, and Cisco WebEx.

2.2 Cassandra Overview: Key Features

This section describes the key features of Cassandra that become prime focus areas for performance optimization.

2.2.1 Write Workflow

Write (or update) requests, a key performance bottleneck in bioinformatics, are handled efficiently in Cassandra using some key in-memory data structures and efficiently arranged secondary storage data structures [10]. We describe them next. As shown in

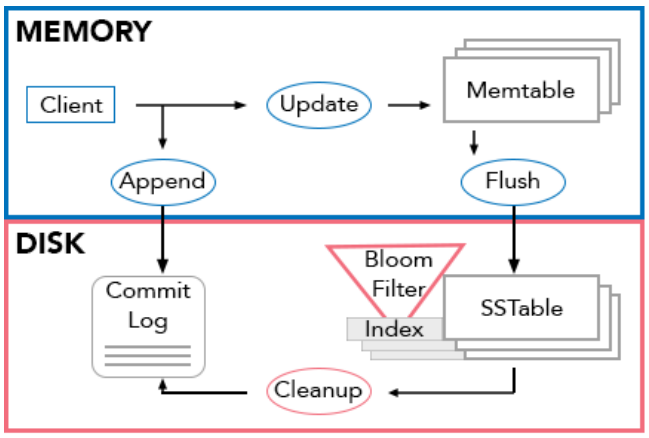

Figure 2. Write workflow overview

figure 2, when a write request arrives, it is appended to Cassandra's CommitLog, a disk-based file where uncommitted queries are saved

[3]Cassandra V3.7 http://cassandra.apache.org

for recovery/replay. Then the result of the query is processed into to an in-memory data structure called the Memtable. A Memtable functions as a write-back cache of data rows that can be looked up by key – that is, unlike a write-through cache, writes are batched up in the Memtable until it is full, when it is flushed (the trigger and manner of flushing are controlled by a set of configuration parameters, which form part of our target for optimization). Each flush operation transfers these contents to the secondary storage representation, called SSTables. SSTables are immutable and every flush task produces a new SSTable.

The data for a given key value may be spread over multiple SSTables. Consequently, if a read request for a row arrives, all SSTables (in addition to the Memtable) have to be searched for portions of that row, and then the partial results combined. This is an expensive process in terms of execution time, especially since SSTables are resident in secondary storage. Over time, Cassandra may write many versions of a row in different SSTables and each version may have a unique set of columns stored with a different timestamp. Cassandra periodically merges SSTables and discards old data in a process called *compaction* to keep the read operation efficient. The compaction process merges keys, combines columns, evicts tombstones, consolidates SSTables, and creates a new index in the merged SSTable.

2.2.2 Compaction

Cassandra provides two compaction strategies, that can be configured on the table level. The default compaction strategy "Size-Tiered Compaction" triggers a new compaction process whenever a number of similar sized SSTables exist, while "Leveled Compaction" divides the SSTables into hierarchical levels. **Size-Tiered Compaction:** This compaction strategy activates whenever a set number of SSTables exist on the disk. Cassandra uses a default number of 4 similarly sized SSTables as the compaction trigger, whereas ScyllaDB triggers a compaction process with respect to each flush operation. At read time, overlapping SSTables might exist and thus the maximum number of SSTables searches for a given row can be equal to the total number of existing SSTables. While this strategy works well with a write-intensive workload, it makes reads slower because the merge-by-size process does not group data by rows. This makes it more likely that versions of a particular row may be spread over many SSTables. Also, it does not evict deleted data until a compaction is triggered. **Leveled Compaction:** The second compaction strategy divides the SSTables into hierarchical levels, say L0, L1, and so on, where L0 is the one where flushes go first. Each level contains a number of equal-sized SSTables that are guaranteed to be non overlapping, and each level contains a number of keys equal to 10X the number of keys at the previous level, thus L1 has 10X the number of keys as in L0. While the keys are non-overlapping within one level, the data corresponding to the same key may be present in multiple levels. Hence, the maximum number of SSTable searches for a given key (equivalently, row to be read) is limited to the number of levels, which is significantly lower than the number of possible searches using the size-tiered compaction strategy. One drawback of the leveled compaction strategy is that the compaction is triggered each time a MEMTable flush occurs, which requires more processing and disk I/O operations to guarantee that the SSTables in each level are non overlapping. Moreover, flushing and compaction at any one level may cause the maximum number of SSTables allowed at that level (say Li) to be reached, leading to a spillover to Level L(i+1). Qualitatively it is known [35] that size-tiered compaction is a better fit for write-heavy workloads, where searching many SSTables for a read request is not a frequent operation.

2.3 Performance metrics: Throughput and Latency

In this paper, we use mean throughput to measure the performance of the datastore. Mean throughput represents the average number of operations the system can perform per second and this is meant to demonstrate the capacity of the database system to support MG-RAST workflow. Some database systems focus on latency metrics that represent the system's response time as observed to the client, but our workload is not latency sensitive, but rather is throughput sensitive. Therefore, our methodology will consider the optimum mean throughput.

2.4 Genomics Workloads

MG-RAST is the world's leading metagenomics sequence data analysis platform developed and maintained at Argonne National Laboratory [43]. Since its inception, MG-RAST has seen growing numbers of datasets and users, with the current version hosting 40,696 public and 279,663 total metagenomes, containing over a trillion sequences amounting to 131.28 Tbp. The derived data products, i.e., results of running the MG-RAST pipeline, are about 10 times the size of the submitted original data, leading to a total of 1.5 PB in a specialized object store and a 250 TB subset in an actively used datastore.

2.4.1 Workload Dynamism

Figure 3 shows the relative ratio of read to write (and update) queries for 4 days of MG-RAST workload. Qualitatively, we observe that there are periods of read heavy, write heavy, and a few mixed during the observed period. More importantly, the transition between these periods is not smooth and often occurs abruptly and lasts for 15 minutes or less. These frequent oscillations make it difficult for online systems to adapt because the transients are very sharp and clearly, static schemes, such as staying with the default configuration settings, will lead to poor performance.

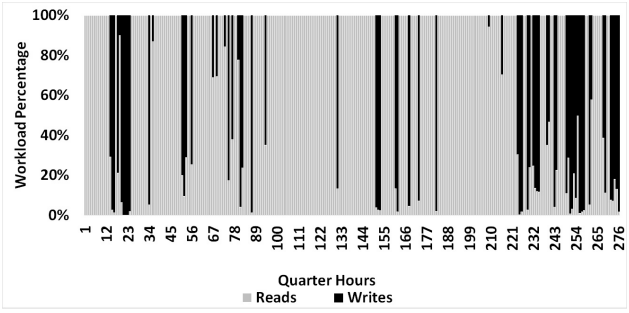

Figure 3. Patterns of workload for MG-RAST. The workload read-/write ratios are shown for 15 minutes intervals.

2.4.2 Processing Pipeline

MG-RAST accepts raw sequence data submission from freely-registered users and pipelines a series of bioinformatics tools to process, analyze, and interpret the data before returning analysis results

to users. Bioinformatics tools in MG-RAST are categorized into: filtering and quality control, data transformation and reduction (such as, gene prediction and RNA detection), and data analysis and interpretation (such as, protein identification and annotation). Many independent users load, process, and store data through these steps, and each user's task may run concurrently with others in the system. Each step accesses elements of a common (genetic) community dataset.

Some pipeline stages break DNA sequences into many overlapping subsequences that are re-inserted into the datastore, increasing processed outputs by a factor of 10 over initial datasets. Moreover, metagenomics often involves decisions that cannot be made until late-stage pipeline passes, creating persistently large working sets. Thus, in this analysis of NoSQL systems, we are driven by the pressing needs of metagenomics platforms, such as MG-RAST.

3 Methodology

This section describes our proposed methodology for creating the middleware for configuration tuning of NoSQL datastores. We start off with the end-to-end workflow, which incorporates both the offline training and the online optimal configuration parameter choice in response to workload changes. Then, we describe the details of each step of the workflow.

3.1 Rafiki Workflow

The Rafiki middleware has the following workflow:

1. **Workload Characterization**: The database workloads are parametrized and expected workloads are injected into the system during data collection. The output metrics, in response to these injected workloads, are measured. In our domain, throughput is the metric of interest.
2. **Important Parameter Identification**: All of the performance related control parameters are identified, and each control parameter is independently varied, measured, and ranked in order of importance using the ANOVA technique. The top ranking parameters are designated as "key parameters" and used for subsequent stages of the workflow.
3. **Data Collection**: Now targeted training runs are carried out by varying the values of only the key parameters so that their interdependent impact on the output metric can be collected for further analysis.
4. **Surrogate Modeling**: The effect of the configuration parameters on the output metrics of interest for any given workload characteristic, is modeled using a Deep Neural Network (DNN).
5. **Configuration Optimization** (Online stage): This occurs while the NoSQL system is operational. In this stage, the optimal configuration parameters for the actual observed workload is identified via a Genetic Algorithm (GA). The GA, to explore a particular point in the search space, does not need to execute the application but queries the surrogate model, which is much faster.

Using this entire workflow, Rafiki generates optimal configuration for a specific server architecture (CPU speed, IO bandwidth, IO capacity, memory capacity, etc.). Rafiki can be used by a NoSQL datastore engine and it is agile enough to converge to new optimal settings in the face of changing workloads.

3.2 Notation

Here, we introduce some notation that we will use for the rest of the section. Each database has a set of parameters $P = \{p_1, p_2, \cdots, p_J\}$, where J is the number of "key parameters", *i.e.*, those that impact performance to a given statistically significant level. Each parameter p has constraints on its values, either programmatically via software-defined limits, or pragmatically, via hardware or application feasibility. It also has a *default value* that is specified by the database software at distribution. The configuration is a set of parameter values $C = \{v_1, v_2, ...v_J\}$, where value v_i corresponds to parameter p_i. In shorthand, we define a configuration by the set of values that have changed from the default settings, *e.g.*, $C = \{v_1 = 5, v_3 = 9\}$ implies that v_2 has its default value. Each parameter p_i has a number of possible distinct values n_i, which may be infinite if the domain of the parameter is in the set of real numbers $\mathcal{R}$. In these cases, the value may be quantized into a finite space. The total number of possible configurations is then $\prod_{i=1}^{J} n_i$.

3.3 Workload Characterization

In the first step of Rafiki, we characterize the application workload in order to apply synthetic benchmarking utilities to drive our system. We use two key parameters to characterize the workload:

- *Read Ratio (RR):* the ratio between the number of read queries and the total number of queries.
- *Key Reuse Distance (KRD):* the number of queries that pass before the same key is re-accessed in a query.

The RR is crucial because the read and write workflows inside of the database take different paths with different tuning parameters that influence performance. The KRD is crucial when measuring the importance of various levels of cache since if keys are rarely reused, then caching is of limited value.

The granularity of time window over which the RR will be measured is dependent on the application. Conceptually, this time interval should be such that the RR statistic is stationary, in an information-theoretic sense [15]. For the MG-RAST workload, we find that a 15 minute interval satisfies this property. A visual representation can be seen in Figure 3. For the KRD, we define a very long window of time over which the key reuse is studied. We then fit an exponential distribution to summarize this metric and use that for driving the benchmarking for the subsequent stages of the workflow. For the MG-RAST case, we use the entire 4-day period over which we have the detailed information about the queries to calculate the KRD statistic. Operationally, it is challenging to get the very detailed information about queries that is required for calculating KRD. This is because of two factors. First, the logging of the exact queries puts a strain on the operational infrastructure, especially in a production environment (MG-RAST in our case), and second, there are privacy concerns with this information, especially in the genomics domain. Hence, for practical purposes, the window of time over which the KRD is calculated, is bounded.

3.4 Important Parameter Identification

Cassandra[4] has over 25 performance-related configuration parameters, and in this piece of our solution approach, we identify which of these are the "key parameters". We will then include these as

[4] http://docs.datastax.com/en/cassandra/3.0/cassandra/configuration/configCassandra_yaml.html

features in Rafiki's surrogate model of the performance of the NoSQL datastore.

3.4.1 Key Parameters

Both Cassandra and ScyllaDB have over 50 configuration parameters, around half of which are related to performance tuning [21]. It is not feasible to search the space of all these parameters in order to identify the optimal parameter settings and instead some pruning is needed. For this, we seek to identify the "key parameters" *i.e.*, the ones that affect performance of the application in a statistically significant manner. For this, we apply an Analysis of Variance test (Anova [28]) to each parameter individually, while fixing the rest of the parameters to their default values. For example, $C_1 = \{v_1 = 5, v_2 = \text{def}, v_3 = \text{def}\}$, $C_2 = \{v_1 = 10, v_2 = \text{def}, v_3 = \text{def}\}$, and $C_3 = \{v_1 = 15, v_2 = \text{def}, v_3 = \text{def}\}$ will be used to collect three sample points S_1, S_2, S_3. This parameter p_1 will be scored with the $var(S_1, S_2, S_3)$, and the other two parameters will also be scored in a similar way. Afterward, we select the top parameters with respect to variance in average throughput. We find empirically that there is a distinct drop in the variance when going from top-k to top-$(k+1)$ and we use this as a signal to select the top-k parameters as the key parameters. For Cassandra, Rafiki identifies the following list of parameters as the key parameters:

1. *Compaction Method (CM)*: This takes a categorical value between the available compaction strategies: Size-Tiered or Leveled[5]. Size-Tiered is recommended for write-heavy workloads, while Leveled is recommended for read-heavy ones.
2. *Concurrent Writes (CW)*: This parameter gives the number of independent threads that will perform writes concurrently. The recommended value is 8 × number of CPU cores.
3. *file_cache_size_in_mb (FCZ)*: This parameter controls how much memory is allocated for the buffer in memory that will hold the data read in from SSTables on disk. The recommended value for this parameter is the minimum between 1/4 of the heap size and 512 MB.
4. *Memory table cleanup threshold (MT)*: This parameter is used to calculate the threshold upon reaching which the MEMTable will be flushed to form an SSTable on secondary storage. Thus, this controls the flushing frequency, and consequently the SSTables creation frequency. The recommended setting for this is based on a somewhat complex criteria of memory capacity and IO bandwidth and even then, it depends on the intensity of writes in the application.
5. *Concurrent Compactors (CC)*: This determines the number of concurrent compaction processes allowed to run simultaneously on a server. The recommended setting is for the smaller of number of disks or number of cores, with a minimum of 2 and a maximum of 8 per CPU core. Simultaneous compactions help preserve read performance in a mixed read-write workload by limiting the number of small SSTables that accumulate during a single long-running compaction.

[5]The third supported option "Time Window Compaction Strategy" is not relevant for our workload and is thus not explored. That is only relevant for time series and expiring time-to-live (TTL) workloads.

These five parameters, combined, control most of the trades between read and write performance, but there is to date no indisputable recommendation for manually setting these configuration parameters nor is there any automated tool for doing this. Poring through discussion forums and advice columns for the two relevant software packages [4, 20, 34], we come away convinced that there is significant domain expertise that is needed to tune a server for a specific workload characteristic and a specific server architecture. Furthermore, these tunings change, and sometime quite drastically, when the workload characteristics change. These lead us to investigate an automatic and generalizable method for achieving the optimal configuration parameter values.

3.5 Data Collection for Training Models

In our goal to optimize the server performance, we encounter a very practical problem: there are too many search points to cover them all in building a statistical model. The five key configuration parameters, as described in the previous section, even if discretized at broad levels, represent $2 \cdot 4 \cdot 8 \cdot 10 \cdot 4 = 2,560$ configurations. If this is combined with 10 potential workloads, then, there are over 25,000 combinations for sampling. With each benchmark run executing for 5 minutes, this would require almost 3 months of compute time to calculate for a single hardware architecture. The benchmark execution time cannot be reduced much below 5 minutes so as to capture the temporal variations in the application queries and remove the startup costs.

The surrogate model in the next step is trained on only a small subset of the potential search space, from which it predicts the performance in any region of the exploration space. This eliminates the need for exhaustively collecting the performance data and allows for a much faster process through the large exploration space of configuration parameters than a typical exhaustive, grid-based search. Empirically we find that training with approximately 5% of the training space (Figure 7) produces accurate enough model, which corresponds to the model results presented in the evaluation results.

We capture the dynamic elements of a NoSQL system with the input feature set: workloads and server configurations. For workloads, we represent the workload feature as read ratio (RR) (write ratio is 1-RR). The KRD is used to configure the data collection, but it is not provided as an input to our model as it is found to be stationary in our target metagenomics domain. Second, the configuration file is represented by the five key parameters, described in Section 3.4. We select the configuration set $\mathbb{C}$ so that, for each parameter p_i, the minimum $\underline{v}_i$ and maximum $\overline{v}_i$ value occurs at least once in the set. The default value is also included in at least one experiment, and additional experiments are added by randomly selecting other values for the parameters, but not in a fully combinatorial way.

Performance data is collected by applying the particular workload W_i and configuration to a specific server and database engine and observing the resultant performance P_i. The space of all workloads exists as $[0, 1]$, representing RR, and it is quantized into n_w discrete values, $\{W_1, W_2, ..., W_{n_w}\}$. The particular sample is defined as $S_i = \{W_i, C_i, P_i\}$. A set of such samples defines the training data for our model.

3.6 Performance Prediction Model

3.6.1 Surrogate Performance Model

Server performance is a function of workloads, configuration parameters, and the hardware specifications on which the software is executed. The relationship between these pieces is sufficiently complex so that no simple guideline can optimize them. This is well known in the performance prediction domain [40, 44] and we show this empirically for MG-RAST in Section 4.6. Although we qualitatively understand the impact of most of these configuration parameters on the output metric, the quantitative relation is of interest in guiding the search toward the optimal settings. It is to be reasonably expected that such a quantitative relation is not analytically tractable. We therefore turn to a statistical model of the relationship of workloads and configurations on the end-performance, for a given hardware specification. We call this the *surrogate performance model*—surrogate because its prediction acts as an efficient stand-in for the actual performance that will be observed.

3.6.2 Creation of the Performance Model

We assume that the relationship between performance, workload, and the database configuration is non-linear and potentially very complex. For this reason, we rely on a sophisticated deep neural network (DNN)-based machine learning model, with multiple hidden layers, to capture the mapping. Our choice of a DNN was driven by the fact that it does not have any assumptions of linearity for either single configuration parameters, or jointly, for multiple parameters, while being able to handle complex interdependencies in the input feature space. The control parameters include the number of layers and the degree of connectivity between the layers. In fact, our experiments validate our hypothesis that the input parameter space displays significant interdependence (Figure 6).

One potential drawback of DNN is the peril of overfitting to the training data. However, in our case, the amount of training data can be made large, "simply" by running the benchmarks with more combinations of input features (a very large space) and workloads (a large space). Therefore, by using different, and large-sized, training datasets and the technique of Bayesian regularization, we can minimize the likelihood of overfitting. Bayesian regularization helps with reducing the chance of overfitting and providing generalization of the learned model. On the flip side, we have to make sure that the network is trained till completion, *i.e.*, till the convergence criterion is met, and we cannot relax the criterion or take recourse to early stopping. Additionally, to improve generalizability, we initialize the same neural network using different edge weights and utilize the average across multiple (20) networks. Further, we utilize simple ensemble pruning by removing the top 30% of the networks that produce the highest reported training error. The final performance value would be an average of 14 networks in this case.

The output of the model is a function that maps {W, C} (workloads and configurations) to the average database operations per second:

$$AOPS_{general} = f_{\text{net}}(W, C) \tag{1}$$

$$AOPS_{Cassandra} = f_{\text{net}}(RR, CM, CW, FCZ, MT, CC) \tag{2}$$

Where: AOPS is the average operations per second that the server achieves, RR is the read ratio describing the workload, and the configuration parameters are CM/CW/FCZ/MT/CC.

In this paper, we utilize Bayesian regularization with back-propagation for training a feed forward neural network to serve as f_{net}. The network size itself can be selected based upon the complexity of the underlying database engine and server and increased with the size of the input dataset to better characterize the configuration-workload space. Section 4.3 further covers our approach for the MG-RAST/Cassandra case.

3.7 Configuration Optimization

3.7.1 Optimal Configuration Search

An exhaustive search could take three months or more of compute time to complete for a single architecture. To reduce this search time, we collect only hundreds of samples (compute hours vs. months) to train f_{net} and use this to predict performance. In doing so, we lose some precision in the configuration space, but we gain the ability to use less efficient but more robust search techniques (*e.g.* genetic algorithms) without an explosion in compute time. For example, in our experiments Rafiki utilizes 3,500 calls to the surrogate model for a single workload data point, so that a few seconds of searching represents 12 days of sampling. Using this approach, we find optimal configurations in Rafiki.

The goal of the database administrator is to maximize server throughput for a given workload:

$$C_{opt} = \arg\max_{C} \text{AOPS}(W, C) \tag{3}$$

Where C represents the configuration parameters (CM, CW, FCZ, MT, CC) and AOPS is the performance. Using the surrogate model, AOPS can be replaced by f_{net} to create an approximate optimization problem:

$$C_{opt} = \arg\max_{\text{C}} f_{\text{net}}(\text{W}, \text{C}) \tag{4}$$

Where f_{net} is from Equation (2). This substitution allows for rapid objective function evaluation.

3.7.2 Optimization using Genetic Algorithms

Since we hypothesize the relationship between the configuration parameters is non-linear and non-continuous (integer parameters CM, CW, and CC), finding optima in this space will require at least some searching. For a general and powerful approach, we use a Genetic Algorithm (GA) to search this space.

We formulate the GA as follows. The fitness function values is the result of f_{net} with the workload parameter W fixed and the C parameters as the input, with higher value denoting a better configuration. The parameters are constrained by the practical limits of the configuration values and as integers where necessary. The initial population is selected to be uniformly random within these bounds. The crossover function calculates intermediate configurations within the bounds of the existing population (to enforce interpolation rather than extrapolation) by taking a random-weighted average between two points in the population. For example, if $C_1 = 3, 5, 7$,$C_2 = 2, 4, 6$, then the crossover output $C_3 = \{\frac{r_1 \cdot 3 + (1-r_1)\cdot 2}{2}, \frac{r_2 \cdot 5 + (1-r_2)\cdot 7}{2}, \frac{r_3 \cdot 7 + (1-r_3)\cdot 6}{2}\}$, where r_1, r_2, r_3 are chosen uniformly random between 0 and 1. The fitness function is modified to ensure constraints are met, as described in [16, 17], where infeasible configuration files are scored with a penalty, and feasible ones are scored as the original fitness function (performance). For example, from above, if $r_1 = 0.3$ and p_1 is an integer parameter, then C_3's $v_1 = 1.15$. The performance of this point will

be penalized by adding a penalty factor because v_1 is not an integer for C_3.

Solving our system using a GA creates a blackbox model, rather than the more desired interpretable model. It may be argued that an interpretable model is particularly important to a system administrator whereby she can estimate what some parameter change will likely do to the performance of the NoSQL datastore. With this goal we experimented with an interpretable model, the decision tree, with the node at each level having a single decision variable, one configuration parameter. We found that this was woefully inadequate in modeling the search space, giving poor performance. When each node was allowed to have a linear combination of the parameters, the performance improved and this is beginning to move toward a less interpretable model. So it appears that for this particular problem, we have to sacrifice interpretability to gain higher expressivity and therefore higher prediction accuracy from the model.

3.8 DBA level of intervention

Although our proposed system provides automatic tuning for the underlying data store, some limited DBA intervention is still needed to support Rafiki with the following items:

1. Performance metrics: What application-specific performance metric should be considered for tuning (throughput, latency, etc.).
2. Performance parameters: List of performance influencing parameters with valid ranges. Excluding security, networking, and consistency related parameters.
3. Representative application trace: A workload for Rafiki to use for characterization as described in Section 3.3.

4 Experiments and Results

In this experimental section we seek to answer the following questions:

1. How effective are the selected parameters in capturing the performance (Average Throughput), and what are the interdependencies among them?
2. What is the sensitivity of the datastore to workload pattern, with the default configurations?
3. Can we accurately predict the performance of the NoSQL datastore for unknown workloads and configurations, *i.e.*, how good is our surrogate model?
4. Finally, the overall goal of Rafiki, can we efficiently search for optimal configuration parameters and improve the performance of the NoSQL datastore? Does this improvement scale to multiple servers?

4.1 Workload Driver and Hardware

We used a Dell PowerEdge R430 as the server machine in the single-server and single-client setup. This server has 2x Intel Xeon E5-2623 v3 3.0 GHz 4-core, 32GB RAM (2x 16GB), and 2x 1TB magnetic disk drives (PERC H330 Mini) mirrored, each drive supporting 6 Gbps. The client machine is Opteron 4386, 8 cores at 3.1 GHz each, 16 GB DDR3 ECC memory. The client and the server are directly connected through a 1 Gbps switch, ensuring that the network is never a bottleneck for our experimental setup. We use a single-server single-client setup for all of the following experiments, except for the multi-server experiment where multiple clients were used to benchmark the multiple servers that were connected in a peer-to-peer cluster.

For all the following experiments, we modified the Yahoo Cloud Serving Benchmark (**YCSB** [14]) tool for emulating MG-RAST simulated workloads to Cassandra. We use YCSB only as a harness to drive the experiments and collect metrics, while all the workload-specific details (query patterns, payload size, key reuse distance, etc.) are derived from actual MG-RAST queries. The simulated queries are based on the most frequent queries submitted to MG-RAST, selected from the MG-RAST query logs, with keys selected from used data-shards. The output is the average throughput measured in operations per second (Section 2.3).

4.2 Data Collection

We orchestrated Cassandra and ScyllaDB inside of Docker containers so that the statefulness of the database is easily maintained. Between data collection events, the server is reset to prevent any caching or persistent information from influencing subsequent benchmarks. For each data point, a fresh Docker container with Cassandra is started and a corresponding YCSB "shooter" then loads the Cassandra server. Multiple shooters are used for a particular server to ensure that it is adequately loaded.

In our experiments, we use 11 different workloads spanning 10% increments between 0% and 100% reads. The number of configurations $|\mathbb{C}| = 20$, resulting in $11 \cdot 20 = 220$ total data points. Each performance point was measured as the average throughput over a 5-minute long benchmarking period. In some cases, if an experiment fails prematurely or other activities add noise to the data collection, then the impacted samples can be removed—20 noisy/faulted samples were removed in our dataset, due to faults in the load-generating clients, thus leaving 200 total samples.

4.3 Surrogate Model and Training

As described in section 3.6.2, the surrogate model (DNN) was trained and tuned with a hidden layer size of [14, 4] based on trial and error, and the ensemble size for the model was selected at 20 networks during all our experiments. For the final experiment where Rafiki selects the optimal parameter settings using GA, we use 100 neural nets since it is still fast enough and gives a moderate improvement. The final DNN architecture has 6 input parameters into the first hidden layer with 14 neurons, a second hidden layer with 4 neurons, and a single output layer and 1 output value. The neurons are connected in a feed-forward setup so that the output of one layer is the input to the next. All experiments utilize 75% training and 25% test unless otherwise mentioned. When we evaluate our model with unseen configurations, we group the data points by configuration, i.e., for each configuration C_i, there exists 11 W workloads, and similarly for unseen workloads. This division is along the configuration or workload dimension—unseen configuration means that no entries for C_i seen in the test set exists in the training set.

We rely on MATLAB's Neural Network Toolbox [1] to construct and train such networks. To train the model, we utilize Bayesian Regularization (`trainbr` in MATLAB). We prevent overfitting by training until convergence or 200 epochs, whichever comes first, and averaging the output of the ensemble of networks, each with different initial conditions, for each validation set. This regularization technique automatically reduces the effective number of parameters in the model at the expense of increasing the training time.

4.4 Cassandra's Sensitivity to Dynamic Workloads

In this experiment, we wish to see how Cassandra's baseline performance is affected by changing nature of the workloads. Figure 4 shows how Cassandra's default performance is highly sensitive to its workload, demonstrating that the average throughput decreases with respect to the increase in workload's read proportion—the swing going from R=0/W=100 to R=100/W=0 is above 40%. This observed behavior is consistent with Cassandra's write path and compaction technique described in Sections 2.2.1 and 2.2.2. Specifically, Cassandra uses a size-tiered compaction strategy, which is triggered whenever a number of similar sized SSTables are created (4 by default). These are not consolidated to ensure that keys are non-overlapping among them. Hence search for a specific key through a read query causes Cassandra to search in all the SSTables causing the lowered read performance. Therefore, tuning Cassandra's performance under read-heavy workloads is essential, especially because we frequently observe read-heavy metagenomics workloads as shown in Figure 3.

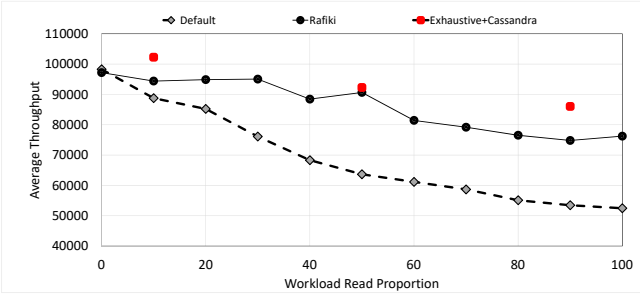

Figure 4. Performance of Cassandra with optimal configuration selected by Rafiki vs. Default configuration. Also three points are shown for the theoretically optimal performance using exhaustive searching.

4.5 Key Parameters Selection

As described in Section 3.4, tuning all configuration parameters for Cassandra is impractical because of the combinatorial explosion of the number of possible configuration sets. Therefore, we apply the ANOVA statistical technique to identify the most significant set of configuration parameters that affect Cassandra's performance, which we call the "key parameters". We vary the value of each parameter individually, while fixing the values of the rest of the parameters to their default. For categorical parameters (*e.g.*, Compaction Strategy) all possible values are tested. Whereas for numerical parameters (*e.g.*, memtable_cleanup_threshold, and concurrent_reads), a number of values (4) are tested as described in Section 3.4. Figure 5 shows the standard deviation in throughput for the top 20 configuration parameters. The most significant parameter, Compaction Strategy has standard deviation 11X that of concurrent writes, and thus removed from the figure for better visualization. From Cassandra's configuration description, we observed that the fifth configuration parameter –memtable_cleanup_threshold– controls the flushing frequency. Cassandra uses this parameter to calculate the amount of space in MB allowed for all MEMtables to grow. This amount is computed using the values of memtable_flush_writers, memtable_offheap_space_in_mb, and memtable_cleanup_threshold. Thus we skip the second and third configuration parameters and only include memtable_cleanup_threshold to control the frequency of MEMtables flushing, having 5 different parameters in total as shown in Equation 1.

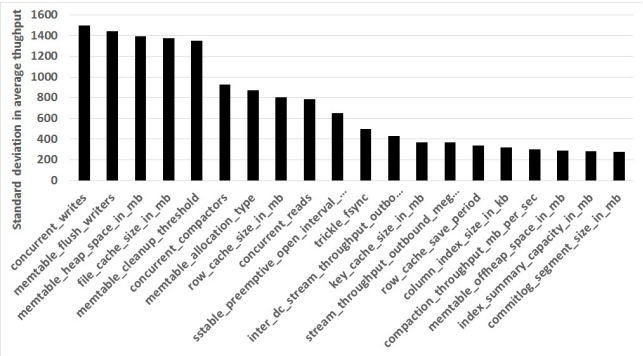

Figure 5. ANOVA analysis for Cassandra to identify the key parameters, *i.e.*, the ones that most significantly control its performance.

4.6 Effectiveness of Selected Configuration Parameters

In this experiment, we assess the effectiveness of the key configuration parameters selected by our statistical analysis. We start by collecting the average throughput for the 220 configuration sets. From the collected data, we examine how impactful the change in performance is compared to the default configuration setting. Table 1 shows the maximum, minimum, and default performance for three different workloads. Which shows that the selected parameters have a significant effect on performance.

One obvious way to tune such parameters is by sweeping greedily through each of them individually while fixing the values of the rest of the parameters. However, we infer that this obvious technique is suboptimal, because it ignores the interdependencies among the selected parameters. Figure 6 shows the effect of changing two parameters from this set: Compaction Method (CM) and Concurrent Writes (CW). Changing one parameter's value results in changing the optimal values for the other parameter. For example, doubling the value of CW from 16 to 32 has a positive effect on performance when CM is set to Size-Tiered Compaction, *i.e.*, throughput increases by 30%, but the same change has very little effect when CM is set to Leveled Compaction. However, doubling the value of CW from 32 to 64 has a negative effect on performance when CM is set to Leveled Compaction, *i.e.*, throughput decreases by 12.7%, but this time it has very little effect when CM is set to Size-Tiered Compaction. This suggests that using a greedy optimization technique, *i.e.*, tuning each configuration parameter individually, cannot find the optimal solution, and motivates us to look for a more sophisticated search strategy for finding the optimal configuration.

Table 1. Cassandra maximum, minimum, and default throughputs as the key-configuration parameters are varied

	Maximum	Default	Minimum
Average Throughput (read=90%)	78,556	53,461	38,785
% over Minimum	102.5%	27.4%	-
Average Throughput (read=50%)	89,981	63,662	53,372
% over Minimum	68.5%	16.16%	-
Average Throughput (read=10%)	102,259	88,771	78,221
% over Minimum	30.7%	11.8%	-

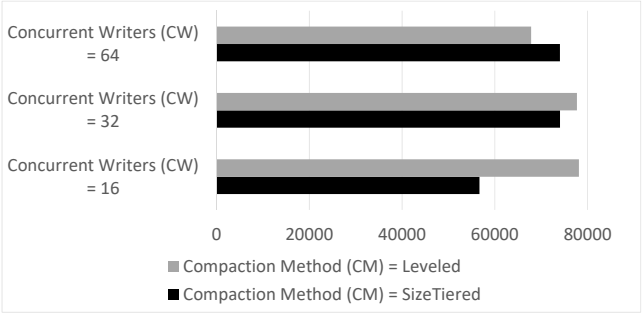

Figure 6. Interdependency among selected parameters necessitating use of non-greedy search strategy.

4.7 Performance Prediction

In this section, we create a surrogate model that can predict the performance of the NoSQL datastore for hitherto unseen workloads and unseen combinations of configuration parameters.

4.7.1 Training the Prediction Model

Figure 7 shows the change in error for predicting performance under unseen workloads and configurations with respect to the number of training samples. We notice that training the model with more data samples enhances the performance but the performance improvement begins to level off at 180 collected training samples. The collected samples were sufficient for getting a prediction error of 7.5% for unseen configurations and 5.6% for unseen workloads using an ensemble of 20 neural nets, compared to 10.1% and 5.95% using a single net (Table 2). Going beyond 20 neural nets again gives diminishing improvements.

4.7.2 Validating Model Performance

We validate the prediction model separately for two dimensions: workloads and server configurations. To validate server configurations, we randomly withhold 25% of the configurations and predict on these. The network is then trained with the remaining 75% of the data, until convergence, and performance statistics are taken for the network (shown in Table 2). We perform 10 randomized trials where we vary which 25% items we withhold and present the average results.

Table 2. Prediction Model Performance for Cassandra

	20 Nets		1 Net	
	Config.	Workload	Config.	Workload
Prediction Error	7.5%	5.6%	10.1%	5.95%
R^2 Value	0.74	0.75	0.51	0.73
Avg. RMSE	6859 op/s	6157 op/s	9338 op/s	6378 op/s

The same procedure was used to validate workload prediction performance, and Figures 8 and 9 show the histogram for the validation cases. The model maintains slightly better prediction accuracy across workloads, suggesting that the single feature that represents the workload's Read-proportion can capture the system dynamics well. The histogram also shows little bias, since the mean is close to zero, and this indicates that our DNN model is expressive enough for the task.

4.8 Performance Improvement through Optimal Configuration

For this experiment, Rafiki searches for better configuration settings for improving the performance of Cassandra. Since we make no assumptions about the relationships between configuration parameters (*e.g.*, their linearity), we utilize a Genetic Algorithm to optimize the set of configuration parameters. We train the network using all available 200 samples, unlike in the previous validation experiments (where we held out some samples). Thus, here while predicting, Rafiki has already seen that particular workload, but in all likelihood has not seen the optimal configuration parameter for that workload considering the large search space. We compare the performance of optimal configurations selected by Rafiki against the default performance, and and exhaustive search for three workloads (90% reads, 50% reads, and 10% reads). This exhaustive search is performed by testing 80 configuration sets for each workload. Figure 4 shows the performance gain when applying Rafiki's selected configuration compared to the default configuration for various workloads. On average, Rafiki shows 30% improvement over the default configuration across the range of workloads. We notice that higher gains are achieved for read-heavy workloads, with 41% improvement on average (range = 39-45%) for read-heavy workloads, *i.e.*, having read-proportion $\geq$ 70%. Lower gains are achieved with respect to write-heavy workloads, *i.e.*, with read-proportion $\leq$ 30%, average of 14% with a range of 6-24%. This reveals that Cassandra's default configuration is more suited for write-heavy workloads, while our metagenomics workload is read-heavy most of the time.

It should be noted that the Rafiki selected configuration comes within 15% with respect to the exhaustive grid search. However, the surrogate model is able to generate a new performance sample in 45 μs, thus allowing the GA to investigate approximately 3000 samples in the search space every 0.17 seconds, whereas around 14 days are needed to collect the equivalent number of samples with grid searching. The combined GA+surrogate takes 1.8 seconds to find the optimal configuration utilizing 3,350 surrogate evaluations on average. A single sample with grid searching takes around 2 minutes for loading data and another 5 minutes for collecting stable performance metrics experimentally. Thus, training the surrogate models and searching using GA is four orders of magnitude faster than exhaustive grid search, suggesting that Rafiki could be used to

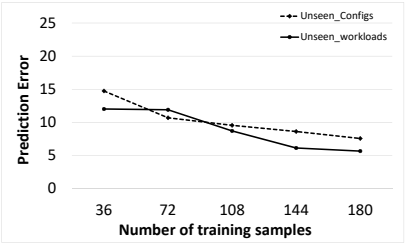

Figure 7. Prediction error for Cassandra using the surrogate performance model with Neural Network, as a function of the number of training samples

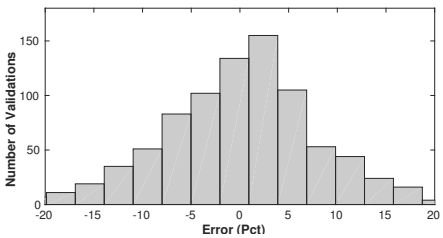

Figure 8. Cassandra: Distribution of performance prediction errors for unseen configurations. The average absolute error is 7.5% with most projections lying in the | 5 |% range.

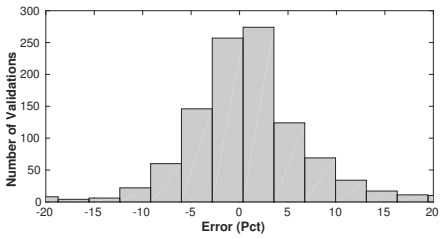

Figure 9. Cassandra: Distribution of performance prediction errors for unseen workloads. The average absolute error is 5.6% with most projections lying in the | 5 |% range

Table 3. Cassandra: Performance improvement of optimal configuration selected by Rafiki vs. Default configuration performance for single-server and two-server setups.

workload	RR=10%	RR=50%	RR=100%
Single Server Improve	15.2%	41.34%	48.35%
Two Servers Improve	3.2 %	67.37%	51.4 %

Table 4. ScyllaDB: Performance of Rafiki selected configurations vs. Grid search

	WL1(R=70%)		WL2(R=100%)	
Opt. Technique	Rafiki	Grid	Rafiki	Grid
Avg Throughput	69,411	75,351	66,503	63,595
Gain over Default	12.29%	21.8%	9%	4.57%

tune the compaction-related and concurrency-related parameters, and possibly others, at runtime as workload characteristics change.

4.9 Performance Improvement for Multiple Cassandra Instances

In this experiment, we measure the improvement in performance for multiple Cassandra instances using Rafiki's selected configurations over the default configurations performance. For the two-servers experiment, we add one more shooter to utilize the increased performance of the created cluster. We also increase the replication factor by one, so that each instance stores an equivalent number of keys as the single-server case. From Table 3, we see similar improvements, on average, using the configuration set given by Rafiki over the default configuration (34% for single-server case, and 40% for two-servers case). Again, the improvement due to Rafiki increases with increasing RR, but it reaches an inflection point at RR=70% and then decreases a little.

4.10 ScyllaDB performance tuning

In this section, we investigate the effect of applying our performance tuning for ScyllaDB[6]. First, we wish to select the key performance tuning parameters. ScyllaDB provides a user-transparent auto-tuning system internal to its operation, and this system makes parameter selection especially difficult because user settings for many configuration parameters are ignored by ScyllaDB, giving preference to its internal auto-tuning [38]. Consequently, even in an otherwise stationary system, without any change to the workload or to the configuration parameters, the throughput of ScyllaDB varies significantly. Figure 10 demonstrates this tuning-induced variance.

We find through experiments that the tuner acts like a hidden parameter—changing a parameter causes variance due to interdependence with the auto-tuning system that we cannot control. Due to these internal interactions, the ANOVA analysis yields significance to configuration settings that have high interaction with the auto-tuning system rather than sustained net-positive impacts on performance. For this reason, despite the ANOVA analysis, poor prediction performance will be obtained. For ScyllaDB, we rectify this by taking the ANOVA analysis for Cassandra, stripping out any parameters that are ignored by ScyllaDB, and adding in new parameters (sorted by variance) until 5 parameters are in the set, matching Cassandra in count. As shown in Table 4, using the selected parameters, Rafiki was able to achieve a performance gain of 12% for workload with 70% reads, and a gain of 9% for workload with 100% reads.

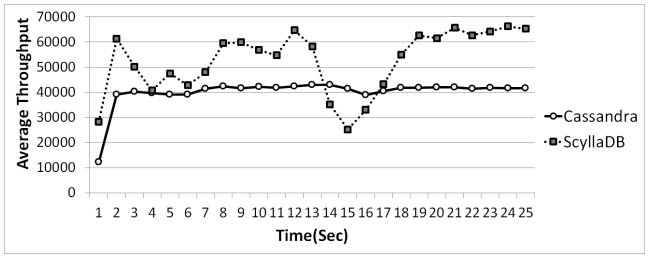

Figure 10. Average throughput for Cassandra and ScyllaDB under a 70% reads workload (collected every 10 seconds). Since Cassandra has a more stable performance compared to ScyllaDB, throughput prediction for Cassandra is more accurate.

5 Related Work

NoSQL Benchmarking: Benchmarking and comparing the performance of NoSQL datastores has been studied recently from several different angles. Authors in [31] show a performance comparison between Cassandra and ScyllaDB. Also Work in [2, 12, 27, 36, 41] has compared and benchmarked Cassandra and various other SQL systems for the purpose of engine selection. These approaches,

[6] https://github.com/scylladb/scylla/blob/master/conf/scylla.yaml

however, do not account for the dynamic workloads seen in MG-RAST and do not predict the performance for unseen workloads.

DBMS Auto-tunning: *iTuned* [19] and *Ottertune* [42], recent DBMS tuning systems, rely on nearest-neighbor interpolation for optimizing configurations for unseen workloads. Rafiki's surrogate model provides algorithm-independent predictive capabilities in contrast to interpolation in *Ottertune*, enabling quick, accurate responses to new workloads. In comparison with *Ottertune's* data collection time,*Ottertune* reports collecting data over 30k trial runs and storing them in its knowledge base ([40] sec 7.2) to be compared with the target workload at run time. In our setup, after required DBA intervention (Section 3.8), a single data-point is collected in 5 min with Rafiki requiring only 200 data points to train a surrogate model which is able to predict unseen workloads and configurations very efficiently, removing the need for collecting any extra data points. In run time, *Ottertune* starts by comparing the target workload to all previously seen workloads. Because of the lack of a surrogate model, this step takes 30-45 min to identify the nearest workload previously seen and to start suggesting a better configuration (in [42] sec 7.4). On the other hand, Rafiki's online training takes few seconds (10-20s) to apply the GA searching algorithm to the surrogate model (trained offline). This allows the GA to test thousands of configurations datapoints per second and suggest the optimal configurations much faster (sec 4.8), which is essential for rapidly changing workloads as seen in MG-RAST.

Distributed Application Parameter Tuning: A subset of the authors have been involved in building scalable cyberinfrastructure previously, for the domain of earthquake and structural engineering [26]. That also involved dealing with large volumes of data with a large set of users on shared computational infrastructure. We are deriving some lessons from this prior work in building the metagenomics cyberinfrastructure. Several middleware tunning systems have been proposed for web-servers such as in [32, 33]. Moreover, A plethora of prior works [6, 7, 29, 30, 40] have attempted to solve parameter tuning problems for MapReduce and other distributed applications [3, 5, 25]. Rafiki differs from and improves upon these works in three key ways. First, the workloads for the database server are inputs to the optimizer. Second, the parameters of interest are mathematically selected using variant analysis. Third, in contrast with some searching methods [5, 25, 30], Rafiki is able to use a less efficient but more generic searching algorithm that avoids local maxima.

Surrogate-Based Tuning: Algorithm complexity has reached a point where searching for optimal parameters requires very large, high dimensional space characterizations, and running software-in-the-loop based tuning is impractical. This has led many tuning papers to rely on *surrogate models* [6, 22, 37] trained from observed system behavior. Such models are used in the optimization phase in place of the actual software to reduce the time complexity for finding optimal tuning parameters. Rafiki also utilizes *surrogate models*, but it also includes characteristics for the inputs to the system directly in the model. We hypothesize that different inputs or workloads require different tuning parameters to be optimal, this allows the surrogate to quickly find new parameters for new workloads.

Performance Prediction: Work in [18] closely resembles the performance prediction piece of our solution. In this work, the authors targeted combined analytical modeling with machine learning techniques to predict the performance of two casestudies: Total Order Broadcast service and a NoSQL datastore (Red Hat's Innispan v.5.2). Their case of NoSQL datastore (performed on synthetic data) shows a good performance of prediction with respect to the number of nodes in the system and the proportion of write transactions. Rafiki provides similar accuracy, but it is based on configuration and workload parameters rather than hardware data (such as CPU, network, etc.). We also utilize real-world workloads (MG-RAST) rather than purely synthetic ones for prediction. In this paper, we combine the prediction element with several supporting elements to improve configuration rather than only predicting performance.

6 Conclusion

In this paper, we have highlighted the problem of tuning NoSQL datastores for dynamic workloads, as is typical for metagenomics workloads, seen in platforms such as MG-RAST. The application performance is particularly sensitive to changes to the workloads and datastore configuration parameters. We design and develop a middleware called Rafiki for automatically configuring the datastore parameters, using traces from the multi-tenant metagenomics system, MG-RAST. We demonstrate the power of Rafiki to achieve tuned performance in throughput for a leading NoSQL datastore, Cassandra, and a latest generation reimplementation of it, ScyllaDB, with the additional distinguishing feature that ScyllaDB has an auto-tuning feature. We apply ANOVA-based analysis to identify the key parameters that are the most impactful to performance. We find, unsurprisingly, that the relation between the identified configuration parameters and performance is non-monotonic and non-linear, while the parameter space is infinite with both continuous and integer control variables. We therefore create a DNN framework to predict the performance for unseen configurations and workloads. It achieves a performance prediction with an error in the range of 5-7% for Cassandra and 7-8% for ScyllaDB. Using a root-cause investigation indicates that ScyllaDB's native performance tends to fluctuate, sometime to a large degree (60% for 40 seconds), making prediction more challenging. We then create a Genetic Algorithm-based search process through the configuration parameter space, which improves the throughput for Cassandra by 41.4% for read-heavy workloads, and 30% on average. Further, to get an estimate of the upper bound of improvement, we compare Rafiki to an exhaustive search process and see that Rafiki, using 4 orders of magnitude lower searching time than exhaustive grid search, reaches within 15% and 9.5% of the theoretically best achievable performances for Cassandra and ScyllaDB, respectively. In future work, we are developing algorithms for the actual online reconfiguration process keeping the downtime to a minimum. We are also developing a prediction model for the workloads, which will allow us to develop an algorithm to cluster reads and writes for higher throughput.

7 Acknowledgements

We thank all the reviewers for their insightful comments, which improved the quality of this paper. This work is supported in part by NSF grant 1527262, NIH Grant 1R01AI123037, and a gift from Adobe Research. Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science under contract DE-AC02-06CH11357. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] 2017. MATLAB Neural Network Toolbox. https://www.mathworks.com/products/neural-network.html. (2017). [Online; accessed 19-May-2017].

[2] Veronika Abramova and Jorge Bernardino. 2013. NoSQL databases: MongoDB vs Cassandra. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*. ACM, 14–22.

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association.

[4] Alteroot. 2017. How to change Cassandra compaction strategy on a production cluster. https://blog.alteroot.org/articles/2015-04-20/change-cassandra-compaction-strategy-on-production-cluster.html. (2017). [Online; accessed 19-May-2017].

[5] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 303–316.

[6] Zhendong Bei, Zhibin Yu, Qixiao Liu, Chengzhong Xu, Shengzhong Feng, and Shuang Song. 2017. MEST: A Model-Driven Efficient Searching Approach for MapReduce Self-Tuning. *IEEE Access* 5 (2017), 3580–3593.

[7] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. 2016. Rfhoc: A random-forest approach to auto-tuning hadoop's configuration. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1470–1483.

[8] Nathan Blow. 2008. Metagenomics: exploring unseen communities. *Nature* 453, 7195 (2008), 687–690.

[9] Rick Cattell. 2011. Scalable SQL and NoSQL data stores. *ACM Sigmod Record* 39, 4 (2011), 12–27.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[11] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 3–14.

[12] Max Chevalier, Mohammed El Malki, Arlind Kopliku, Olivier Teste, and Ronan Tournier. 2015. Benchmark for OLAP on nosql technologies comparing nosql multidimensional data warehousing solutions. In *IEEE International Conference on Research Challenges in Information Science (RCIS)*. 480–485.

[13] Charles E Cook, Mary Todd Bergman, Robert D Finn, Guy Cochrane, Ewan Birney, and Rolf Apweiler. 2016. The European Bioinformatics Institute in 2016: data growth and integration. *Nucleic acids research* 44, D1 (2016), D20–D26.

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

[15] Imre Csiszar and János Körner. 2011. *Information theory: coding theorems for discrete memoryless systems*. Cambridge University Press.

[16] Kalyanmoy Deb. 2000. An efficient constraint handling method for genetic algorithms. *Computer methods in applied mechanics and engineering* 186, 2 (2000), 311–338.

[17] Kusum Deep, Krishna Pratap Singh, Mitthan Lal Kansal, and C Mohan. 2009. A real coded genetic algorithm for solving integer and mixed integer optimization problems. *Appl. Math. Comput.* 212, 2 (2009), 505–518.

[18] Diego Didona, Francesco Quaglia, Paolo Romano, and Ennio Torre. 2015. Enhancing performance prediction robustness by combining analytical modeling and machine learning. In *Proceedings of the 6th ACM/SPEC international conference on performance engineering*. ACM, 145–156.

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

[20] DataStax Enterprise. 2017. Apache Cassandra: Configuring compaction. http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsConfigureCompaction.html. (2017). [Online; accessed 19-May-2017].

[21] DataStax Enterprise. 2017. The cassandra.yaml configuration file. http://docs.datastax.com/en/cassandra/3.0/cassandra/configuration/configCassandra_yaml.html. (2017). [Online; accessed 19-May-2017].

[22] Adem Efe Gencer, David Bindel, Emin Gün Sirer, and Robbert van Renesse. 2015. Configuring Distributed Computations Using Response Surfaces. In *Proceedings of the 16th Annual Middleware Conference*. ACM, 235–246.

[23] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2 (2002), 51–59.

[24] Margaret E Glasner. 2017. Finding enzymes in the gut metagenome. *Science* 355, 6325 (2017), 577–578.

[25] Yanfei Guo, Palden Lama, Changjun Jiang, and Xiaobo Zhou. 2014. Automated and agile server parametertuning by coordinated learning and control. *IEEE Transactions on Parallel and Distributed Systems* 25, 4 (2014), 876–886.

[26] Thomas J Hacker, Rudi Eigenmann, Saurabh Bagchi, Ayhan Irfanoglu, Santiago Pujol, Ann Catlin, and Ellen Rathje. 2011. The NEEShub cyberinfrastructure for earthquake engineering. *Computing in Science & Engineering* 13, 4 (2011), 67–78.

[27] Jing Han, E Haihong, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE, 363–366.

[28] Gudmund R Iversen and Helmut Norpoth. 1987. *Analysis of variance*. Number 1. Sage.

[29] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. 2014. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 165–176.

[30] Guangdeng Liao, Kushal Datta, and Theodore L Willke. 2013. Gunther: Search-based auto-tuning of mapreduce. In *European Conference on Parallel Processing*. Springer, 406–419.

[31] Ashraf Mahgoub, Sachandhan Ganesh, Folker Meyer, Ananth Grama, and Somali Chaterji. 2017. Suitability of NoSQL systems–Cassandra and ScyllaDB–for IoT workloads. (2017).

[32] Amiya K Maji, Subrata Mitra, and Saurabh Bagchi. 2015. Ice: An integrated configuration engine for interference mitigation in cloud services. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on*. IEEE, 91–100.

[33] Amiya K Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. 2014. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference*. ACM, 277–288.

[34] Stack Overflow. 2017. Can Cassandra compaction strategy be changed dynamically? http://stackoverflow.com/questions/26640385/can/-cassandra/-compaction/-strategy/-be/-changed/-dynamically. (2017). [Online; accessed 19-May-2017].

[35] End Point. 2017. The Write Path to Compaction. http://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_write_path_c.html. (2017).

[36] Jaroslav Pokorny. 2013. NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems* 9, 1 (2013), 69–82.

[37] Mike Preuss, Günter Rudolph, and Simon Wessing. 2010. Tuning optimization algorithms for real-world problems by means of surrogate modeling. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, 401–408.

[38] ScyllaDB. 2017. Scylla release: version 1.6. http://www.scylladb.com/2017/02/06/scylla-release-version-1-6/. (February 2017).

[39] Dennis Shasha and Philippe Bonnet. 2002. *Database tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann.

[40] Lizhen Shi, Zhong Wang, Weikuan Yu, and Xiandong Meng. 2017. A case study of tuning MapReduce for efficient Bioinformatics in the cloud. *Parallel Comput.* 61 (2017), 83–95.

[41] Bogdan George Tudorica and Cristian Bucur. 2011. A comparison between several NoSQL databases with comments and notes. In *10th International Conference on Networking in Education and Research*. IEEE, 1–5.

[42] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1009–1024.

[43] Andreas Wilke, Jared Bischof, Wolfgang Gerlach, Elizabeth Glass, Travis Harrison, Kevin P Keegan, Tobias Paczian, William L Trimble, Saurabh Bagchi, Ananth Grama, and others. 2016. The MG-RAST metagenomics database and portal in 2015. *Nucleic acids research* 44, D1 (2016), D590–D594.

[44] Yunquan Zhang, Ting Cao, Shigang Li, Xinhui Tian, Liang Yuan, Haipeng Jia, and Athanasios V Vasilakos. 2016. Parallel processing systems for big data: a survey. *Proc. IEEE* 104, 11 (2016), 2114–2136.