Optimizing KV stores for workloads using Machine Learning

Aanandita Dhawan, Himanshu Chaudhary, Surendra Parla, Tanisha Hegde
UW–Madison

ABSTRACT

Enhancing the performance of a database system to meet specific workloads has been a constant challenge. LlamaTune utilizes machine learning approaches and assesses the performance of Database Management Systems (DBMS) across varied configurations. Existing frameworks use LLamaTune to optimize databases like PostgreSQL and MySQL. The increase of NoSQL databases due to a shift in storage methods and type of data generates the need to optimize these databases as well. Our goal is to extend this platform to NoSQL databases and test this setup on workloads using YCSB benchmark. We acheive this by using MLOS a Microsoft Machine learning platform that integrates LlamaTune with frameworks like Nautilus in conjunction. We compare our results based on default configurations of Cassandra database to the optimized configurations suggested by the model.

1 INTRODUCTION

Database tuning is a critical aspect of performance optimization, involving the adjustment of various parameters to achieve optimal operation. However, traditional methods such as Sequential Model-based Algorithm Configuration (SMAC) have complexities due to high-dimensional configuration spaces, leading to prolonged iterations and time-consuming processes. While machine learning (ML) models have shown promise in this domain, they too face challenges, particularly in efficiently navigating and pruning the search space.

LlamaTune [7] uses dimensionality reduction strategies such as randomized projections and biased sampling for specialized knob values. Its primary objective is to significantly enhance sample efficiency, surpassing the capabilities of existing optimizers by pinpointing optimal configurations with fewer workload runs and increased throughput. LlamaTune is currently integrated as an adaptor in Microsoft MLOS an open-source project to enable autotuning for systems.

The project aims to use this framework for KV stores, namely Cassandra and Redis and observe the effect of tuning for workloads generated by YCSB. To achieve this we first work on setting up an existing framework named Nautilus to streamline the evaluation of DBMS configurations for Redis and Cassandra. In other words, we use the platform to enable measuring the performance of the DBMS configuration under different workloads and hardware configurations. Following this we then work on Tuning the database using results and a Configuration space for each of these databases for multiple iterations. We evaluate our model based on Throughput as a result of benchmarks.

The main work of our project can be divided into the following steps:

(1) Extend the Nautilus framework to support Redis and Cassandra.
(2) Create an MLOS pipeline to read results generated by Nautilus for tuning.
(3) Run multiple experiments by changing the workload for each of Cassandra and Redis.
(4) Derive conclusions from the experiments.

2 RELATED WORK

Achieving optimal performance in Database Management Systems (DBMS) is a challenging task due to their inherent complexity and the multitude of configurable options that govern nearly every aspect of their runtime behavior. These configuration parameters empower database administrators (DBAs) to fine-tune various aspects of the DBMS's operation, such as memory allocation for data caching versus transaction log buffering. Modern DBMS are known for offering a plethora of configuration options [4, 9], and their performance and scalability heavily rely on these configurations. Complicating matters further is the fact that the default settings for these parameters are often suboptimal. Given this, tuning the configuration of a database management system (DBMS) is necessary to achieve high performance.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In light of these challenges, many organizations find themselves resorting to hiring costly experts to fine-tune the system's parameters according to anticipated workloads. However, as databases and applications continue to grow in size and complexity, the task of optimizing a Database Management System (DBMS) to align with application requirements has surpassed human capabilities. The correct configuration of a DBMS relies on numerous factors that often exceed human reasoning abilities[1]. Moreover, only a handful of experienced database administrators (DBAs) possess the expertise to set appropriate knob configurations. Even among these skilled professionals, cloud database environments pose unique challenges. In cloud databases (CDB), even the most seasoned DBAs may struggle to resolve many tuning issues. Consequently, cloud database service providers face the dilemma of tuning systems for numerous users with limited access to skilled and expensive DBA expertise. In response, developing effective systems for automatic parameter configuration and optimization emerges as an essential strategy to address this challenge.

As a response to these challenges, several automated tuning methods [2, 5, 10–12] have been proposed, with recent advancements leveraging machine learning (ML) techniques. ML-based approaches have demonstrated the ability to generalize across various workloads and have shown remarkable success in identifying configurations that yield significantly higher throughput or lower latency compared to manually-tuned configurations, often achieving improvements of up to 3-6×.

These ML-based approaches can be broadly categorized into two main classes. The first class involves prior training on selected benchmarks, with subsequent transfer or fine-tuning of this knowledge to accommodate new customer workloads. Examples of such approaches include OtterTune and CDBTune. The second class directly tunes new customer workloads by iteratively selecting configurations using an optimizer and then running workloads with them. In these approaches, algorithms within the configuration optimizers aim to strike a balance between exploring unseen regions of the configuration search space and exploiting the knowledge gathered thus far. Commonly utilized techniques include search-based methods, reinforcement learning (RL)-based methods, and Bayesian Optimization (BO) methods. Notably, a recent study found that SMAC, a BO-based method, outperformed others in the context of DBMS tuning.

3 BACKGROUND

3.1 Architecture

The complete architecture of our system includes the conjunction of Nautilus and MLOS. The Figure 1 shows the system flow where the green box indicates Nautilus and

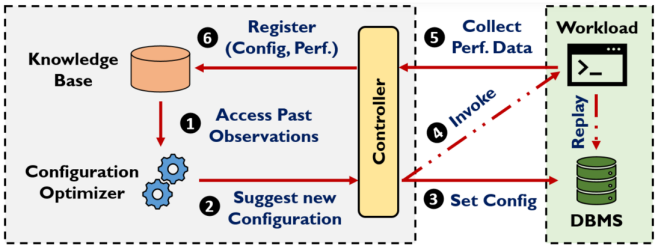

Figure 1: System Flow

the Grey box indicates MLOS. The whole processes can be broken down into the below steps:

(1) Run a workload against a database.
(2) Retrieve results of the benchmark along with other metrics.
(3) Feed the data to an Optimizer (SMAC).
(4) Get a new suggested configuration from the optimizer.
(5) Apply these new configurations to your database.
(6) Repeat number of times.

3.1.1 Nautilus. Nautilus aims to simplify the assessment of DBMS setups. Essentially, it offers a platform for gauging the efficiency of a DBMS setup across various workloads and hardware setups. At present, Nautilus is compatible with two DBMSs (specifically, PostgreSQL and MySQL) and two benchmark suites (namely, YCSB and BenchBase). However, its architecture is designed to facilitate the seamless integration of additional DBMSs or benchmark suites with minimal effort. Nautilus submits a job to Ray [8] to invoke a container with the given database and runs a specified workload.

3.1.2 MLOS. MLOS is an open-source project that introduces two Python modules, `mlos-bench` and `mlos-viz`, enabling standalone or combined use for automated benchmarking and visualization. MLOS currently focuses on an offline tuning approach. `mlos-bench` streamlines benchmark execution, while `mlos-viz` simplifies result visualization via straightforward APIs. `mlos-core` abstracts optimization frameworks like FLAML and SMAC, offering a user-friendly, low-dependency interface for describing parameter spaces, constraints, and objectives. It includes an "optimizer" service for seamless swapping of search methods and utilities for automating experiment loops and data collection. With the help of existing OSS libraries, the design prioritizes adaptability, with policies tailored for autotuning systems. Through wrappers, users can experiment with replacing optimizer components to accommodate evolving techniques.

3.2 Tuning Cassandra

Apache Cassandra is a distributed NoSQL database known for its scalability and fault tolerance. It uses a peer-to-peer

Table 1: Cassandra knobs and corresponding configuration space

knob	config space
key_cache_size_in_mb	0-100
row_cache_size_in_mb	0-100
row_cache_save_period	0-14400
counter_cache_size_in_mb	0-50
counter_cache_save_period	7200-14400
commitlog_segment_size_in_mb	8-64
concurrent_reads	16-64
trickle_fsync_interval_in_kb	5120-20480
column_index_cache_size_in_kb	1-4
compaction_throughput_mb_per_sec	16-32

gossip protocol for communication between nodes and follows a column-family data model. Data is partitioned and replicated across nodes, with tunable consistency levels for balancing consistency, availability, and performance. Write operations are first logged and then stored in memory before being flushed to disk, while read operations are performed based on the partition key. This architecture makes Cassandra suitable for applications requiring high throughput and resilience to failures.

Apache Cassandra boasts approximately 155 tunable parameters. Building on this research [6], we found that optimizing a select few critical knobs is ample for attaining top-tier database management system (DBMS) performance. Additionally, narrowing down the configuration space can notably reduce the number of samples needed for effective tuning. Table 1 shows the important knobs for cassandra and the search space for each knobs.

3.3 Tuning Redis

Redis is an open-source, in-memory data structure store utilized as a database, cache, and message broker. Renowned for its speed, simplicity, and adaptability, it finds application in real-time analytics, caching, messaging, and session management. Redis primarily stores data in memory, enabling rapid read and write operations, with options for disk persistence. It employs a key-value data model, offering efficient data retrieval and manipulation. Supporting various data structures like strings, hashes, lists, sets, and sorted sets, Redis ensures flexibility in handling diverse data types. The platform provides persistence options such as snapshotting and append-only file (AOF) logging for data durability. It supports replication for fault tolerance and clustering for scalability and high availability. Additionally, Redis features built-in publish/subscribe messaging for real-time communication. Its attributes of speed, simplicity, and versatility render it a favored choice across a broad spectrum of applications and use cases.

Table 2: Redis knobs and corresponding configuration space

knob	config space
maxmemory-policy	['volatile-random', 'noeviction', 'volatile-lru', 'allkeys-random', 'allkeys-lru']
tcp-backlog	100-600
appendfsync	['everysec', 'always', 'no']
timeout	0-60
save 60	1000-10000
auto-aof-rewrite-percentage	0-100

Similar to cassandra, we also used few critical knobs for redis. Table 2 shows the important knobs for redis and the search space for each knobs.

4 METHODOLOGY

This section provides a brief on steps involved in executing the system as a whole, going into the nuances of code implementation, dependencies management, and setup intricacies.

4.1 System Setup

The deployment environment selected for the system is CloudLab, specifically `c220g5` node within the Wisconsin Cluster. Within this environment, the Nautilus framework is used to run the workload. Our code is written in Python and uses Docker for invocation of database instances.

The hardware specifications of the c220g5 node are outlined in Table 3.

4.2 Code Flow

A main aspect of the methodology is understanding the code flow involved in seamlessly integrating and fine-tuning the Cassandra database for the Yahoo! Cloud Serving Benchmark (YCSB).

4.2.1 MLOS. The MLOS framework plays an important role in optimizing the database configurations. The process involves the below steps:

Step 1 Definition of a Python task function tailored to accept thread count as input and deliver throughput as output, ensuring efficient workload processing and performance assessment.

Step 2 Defining a DataFrame to store configuration parameters' values, including throughput and latency metrics for each iteration, allowing performance analysis and optimization tracking.

Step 3 Initialization of the chosen optimization engine, such as Sequential Model-Based Algorithm Configuration (SMAC), to iteratively explore the configuration space and identify optimal parameter settings.

Table 3: Hardware Specifications of c220g5

Component	Specification
CPU	Two Intel Xeon Silver 4114 10-core CPUs operating at 2.20 GHz, providing substantial computational horsepower for intricate database operations.
RAM	192GB ECC DDR4-2666 Memory, ensuring ample memory capacity for handling large datasets and intensive computational tasks.
Disk 1	One 1 TB 7200 RPM 6G SAS HDs, offering robust storage capacity and efficient data retrieval capabilities for optimized database performance.
Disk 2	One Intel DC S3500 480 GB 6G SATA SSD, enhancing storage performance with faster data access and lower latency for critical database operations.

Step 4 Execution of the optimization loop, where the chosen optimizer iteratively suggests new configurations based on performance feedback and refines existing configurations to enhance database efficiency.

Step 5 Preservation of the resultant DataFrame to a CSV file, enabling easy data storage and accessibility for further analysis and visualization.

4.2.2 Nautilus. The Nautilus framework, functioning in tandem with MLOS, orchestrates the execution and evaluation of database configurations and ensures easy integration and performance enhancement. The workflow within Nautilus includes the following key steps:

Step 1 Initialization of the Ray framework, facilitating an containerized execution of the a version specific database.

Step 2 Provisioning of a Cassandra container within the CloudLab environment, enabling isolated and controlled database instances for rigorous testing and evaluation.

Step 3 Loading of default database configurations for the first iteration, serving as the baseline for performance comparison and optimization.

Step 4 Execution of an initial script tailored to the specific database requirements, facilitating the benchmarking process and generating essential performance metrics.

Step 5 Execution of the designated workload, simulating real-world database operations and generating comprehensive performance data for analysis and optimization.

Step 6 Aggregation of the generated output metrics and storage in a structured format, ensuring seamless data management and accessibility for further analysis and visualization.

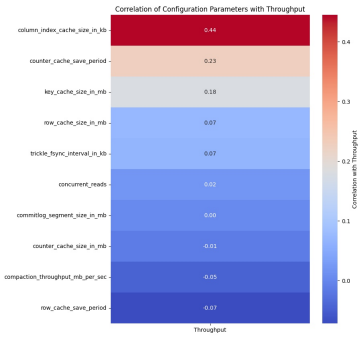

Figure 2: Correlation of Cassandra knobs (Workload C)

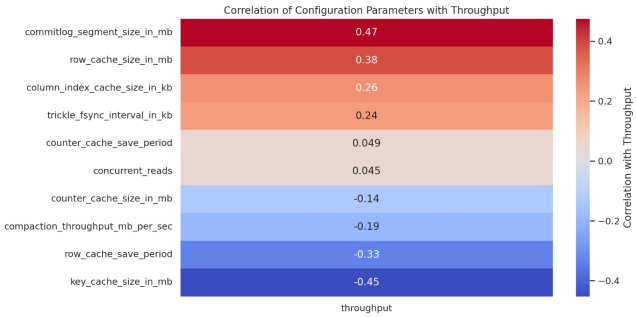

Figure 3: Correlation of Cassandra knobs (Workload A)

The iterative collaboration between MLOS and Nautilus forms the cornerstone of the optimization process, enabling continuous refinement of database configurations to achieve optimal performance under diverse workloads and scenarios.

4.2.3 Visualization. Upon completion of the optimization iterations, a CSV file is generated, with the tuned parameter settings and their corresponding performance metrics. This dataset helps visualization and analysis.

5 EVALUATION

In this section we detail about our experiments performed on Cassandra version 3.11 and Redis Version 7.2.4 on YCSB workload C. We performed 21 iterations for tuning our databases. As mentioned earlier all our experiments are performed on `c220g5` of CloudLab.

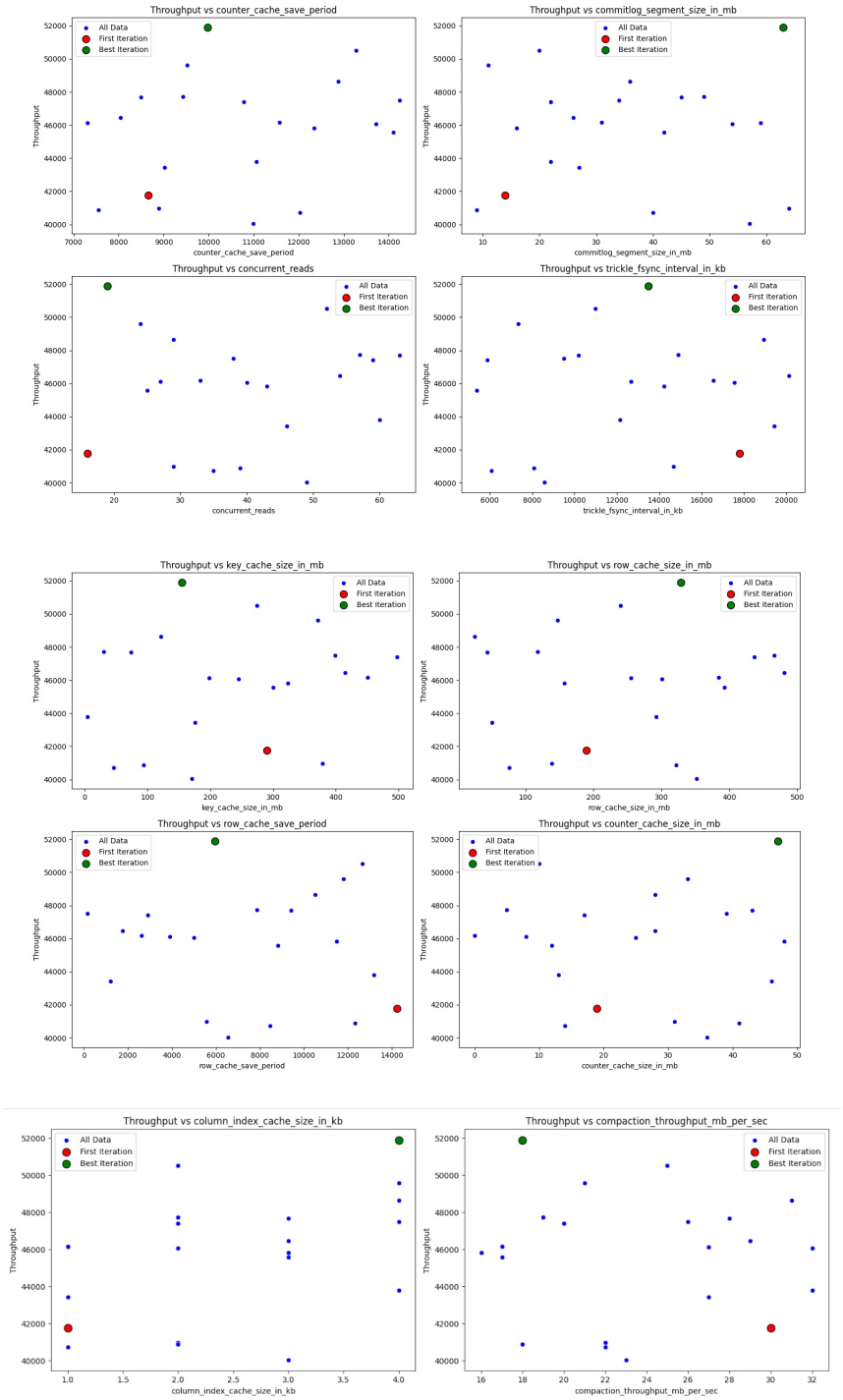

Figure 4: Scatter plot for Cassandra knobs in each iteration (Workload C)

5.1 YCSB Workload C

The Yahoo! Cloud Serving Benchmark (YCSB) is a popular open-source framework designed to evaluate the performance of cloud-based database systems under various workloads. The "C" workload in YCSB is characterized by a mixture of read and write operations with a predominantly read-heavy pattern. This workload is representative of scenarios where the application requires frequent access to existing data.

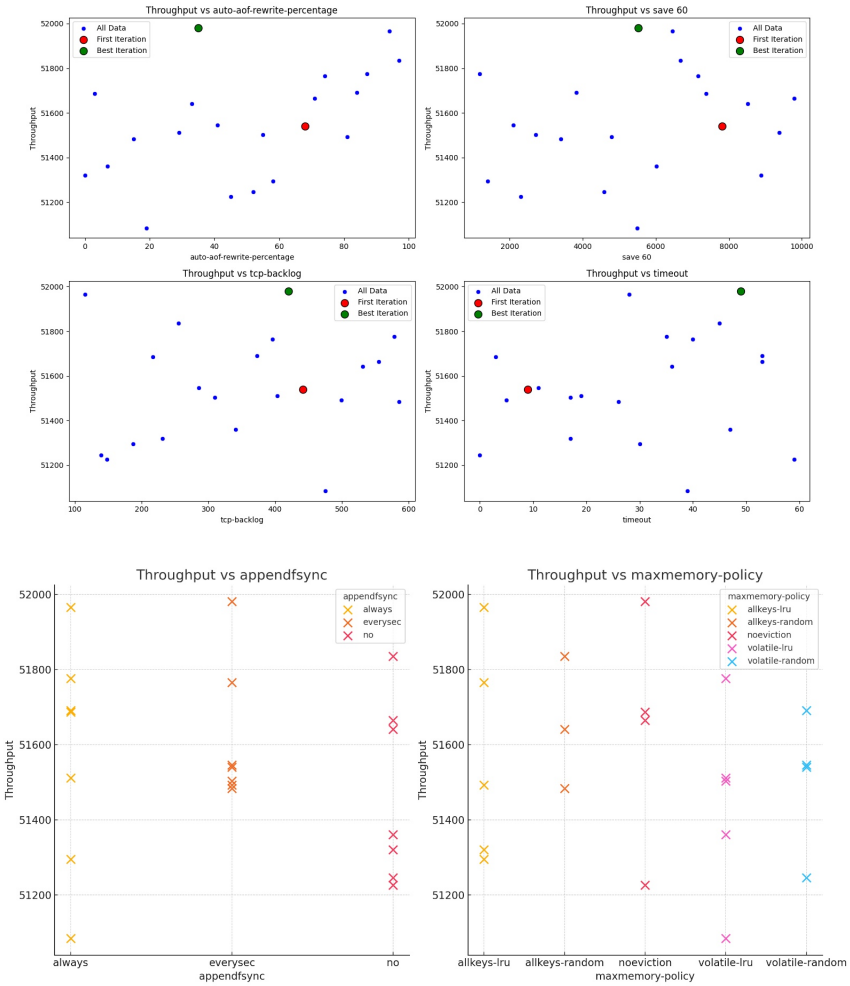

Figure 5: Scatter plot for Redis knobs in each iteration (Workload C)

Workload C is characterized by a high query intensity, indicating a large volume of read requests compared to write requests [3]. This workload is suitable for testing the read performance of database systems under high-concurrency scenarios. The access patterns in workload C may involve a mix of sequential and random data access, simulating real-world application scenarios where data retrieval occurs based on specific access patterns.

Additionally, workload C may involve a diverse distribution of data across the database, including both hot and cold data. This distribution pattern helps assess the database's ability to efficiently handle data retrieval requests across different data segments.

Workload Properties :
The experiments involved with Cassandra and redis with YCSB workload C have the below Properties:
Record Count: 10M
Operation Count: 10M
Thread Count: 20
Warmup duration: 20s
Benchmark duration: 5 mins

5.2 Expriments on Cassandra

We observed that, tuning Cassandra on the Configuration space of 10 knobs helped increase the throughput by 25%. Figure 2 indicates the correlation of each knob to the throughput over all the iterations. We see that `column_index_cache_size` is largely and positively correlated followed by the `counter_cache_save_period`. Though the workload is read heavy, we observe tuning the concurrent reads does not have an effect on the throughput. Plotting values of each knob through each iteration showed that the search is random. This helped us conclude that experiment would need to run for a larger duration for it to converge to a value. Figure4 show these observations.

As a comparison we ran Cassandra with the same workload properties but with Workload A. While the results of

the tuning were not promising we looked at the correlation of the knobs and the results. We observed a change in correlation of knobs and throughput. Figure 3 shows the correlation for Workload A. We see that the selected knobs are more effective than for Workload C. While Workload A is update heavy and Workload C is read heavy, we observe that the number of knobs that affect the throughput is more. This would mean a higher number of iterations to get get promising results.

5.3 Experiments on Redis

We observed only 0.8% increase in the throughput while tuning Redis on the Configuration space of 6 knobs. The performance of Redis has notably declined. We have extensively discussed on the causes in the discussion section. Figure 5 shows the plot of each knob values and corresponding performance through each iterations. Again the search seems to be random and higher number of iterations would be required to get better results.

6 DISCUSSION

6.1 Redis Vs Cassandra

The same set of default parameters for Cassandra gave throughput of 47800 ops/sec and 41000 ops/sec for ycsb workload a and workload c respectively. Which shows the importance of tuning parameters with respect to the workload.

We observed that the average throughput of Redis is 25% higher than Cassandra, attributed to Redis being an in-memory datastore. However, our experimentation with Redis was limited to smaller record counts due to its RAM limitations; Redis exhausts available RAM at a record count of 1 million. In contrast, Cassandra can operate at record counts of 10M or even 100M since it operates on disk. The selection of knobs and their respective ranges significantly influences convergence, particularly when training for fewer iterations and with fewer configuration parameters.

For workload C, the best throughput result of 51,800 ops/sec was achieved in less than 10 iterations, as we identified knobs with a better correlation with throughput. Conversely, identifying tunable parameters for Redis proves challenging because it is an in-memory database, and its performance directly correlates with available RAM, which is limited. In such situations, including all tunable knobs and running the experiment for 50 iterations may yield substantially better results.

6.2 Ease of Use: Nautilus

While Nautilus offers great flexibility for integrating standalone databases, integrating them in cluster mode can be more involved. The main effort lies in creating a configuration file. This file requires a good understanding of the specific database and its documentation, as it details all the necessary parameters.

The integration process itself is relatively straightforward. It involves writing a short Python script (around 150 lines) with two functions: one to retrieve the current database size and another to merge a generated configuration with a default configuration file. This merged configuration is then used for the actual run.

6.3 Ease of Use: MLOS

The MLOS framework offers extensive customization for configuring hyperparameters. It supports defining search spaces for categorical options, continuous values within specific ranges (integers and floats), and discrete parameters with weights. However, managing a large number of parameters can be cumbersome due to the need for manual variable definition. To address this, we have added feature where users can specify tunable parameters and their ranges in a simple CSV format. These are then automatically converted and fed into the framework. This approach can be further extended to support other parameter types using YAML configuration files, simplifying hyper-parameter management for complex models.

7 FUTURE WORK

We had plans to integrate join order benchmark into the Nautilus framework but couldn't finish it in time because we had to integrate Nautilus with MLOS and the training, identifying the knobs to tune took most of our project time. So, the future work would be to compare ease of integrating a benchmark to Nautilus.

We have identified that integrating a database in cluster mode would require some work in terms of modifying the interface or implementing a new interface for cluster mode. The number of nodes can directly be configured in the docker compose file which makes the code cleaner and simpler. Comparing the experiment results with standalone and cluster mode would be an interesting aspect to explore in future.

To further enhance the Nautilus framework, consider enabling the integration of custom application workloads. This would allow users to fine-tune database configuration parameters specifically for their applications. Integration testing and regression testing frameworks could then be leveraged to validate these configurations, ensuring optimal performance. Finally, the seamlessly generated configuration files could be seamlessly deployed to production environments, streamlining the database setup process.

8 CONCLUSION

We observed that with though Bayesian optimization techniques we can reduce the training time significantly and can achieve better results selecting the tunable knobs plays an important role in the optimization process. It leads to some form of manual/human intervention identifying the knobs and corresponding ranges.

There are some researchers who use machine learning and LLM techniques to create a knowledge base from the official documentation and websites like stack-overflow and stack-exchange. This knowledge base is then used in the reinforcement learning algorithm to suggest the next batch parameters. We could integrate the knowledge to select the parameters to tune and intial config and integrate into MLOS.

9 CONTRIBUTIONS

Instance and code setup: Surendra and Tanisha
Integration of Cassandra and Redis with Nautilus: Surendra, Tanisha
Code understanding and integration with MLOS: Aanandita and Himanshu
Running experiments: Surendra
Code for generating plots: Aanandita
Selection of knobs for Redis and Cassandra: Himanshu

The four of us coordinated our work and distributed the load while working on the Poster, Checkins and Final Report. We discussed on the experiments to run and coordinated with our TA through Slack and E-mail for any issues we encountered. We had weekly calls with him to stay on track and inform him about our progress.

REFERENCES

[1] Philip Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Michael Franklin, Hector Garcia-Molina, Jim Gray, Gerald Held, Joseph Hellerstein, H. Jagadish, Michael Lesk, David Maier, Jeffrey Naughton, Hamid Pirahesh, Michael Stonebraker, and Jeffrey Ullman. 1998. The Asilomar Report on Database Research. *SIGMOD Record* 27 (12 1998), 74–80. https://doi.org/10.1145/306101.306137

[2] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. 2021. CGPTuner: a contextual gaussian process bandit approach for the automatic tuning of IT configurations under varying workload conditions. (2021).

[3] Brian F. Cooper and contributors. Accessed: 2024. YCSB Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. (Accessed: 2024).

[4] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:2916569

[5] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (aug 2009), 1246–1257. https://doi.org/10.14778/1687627.1687767

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

[7] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: sample-efficient DBMS configuration tuning. *Proceedings of the VLDB Endowment* 15, 11 (July 2022), 2953–2965. https://doi.org/10.14778/3551793.3551844

[8] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[9] Adam Storm, Christian Garcia-Arellano, Sam Lightstone, Yixin Diao, and Maheswaran Surendra. 2006. Adaptive Self-tuning Memory in DB2. 1081–1092.

[10] 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 (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[11] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. 2102–2114. https://doi.org/10.1145/3448016.3457291

[12] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 338–350. https://doi.org/10.1145/3127479.3128605