Modified DB-BERT
A Database Tuning Tool that 'Reads the Manual'

Mathew George, Samiha Kuncham, Joel J Stephen, Peter V Vu, Punya P Modi

Erik Jonsson School of Engineering and Computer Science, The University of Texas at Dallas

Abstract

Database performance tuning helps optimize system performance like latency and response time, but traditional approaches rely on manual configuration, trial and error, and rule-based heuristics. Modern machine learning techniques have introduced ways to automate database performance tuning. One such example is DB-BERT, which automates the tuning process by leveraging reinforcement learning and NLP techniques to identify and optimize database parameters, reduce manual overhead, and help achieve optimal performance settings more efficiently. However, the hardware requirements to run DB-BERT alongside an existing instance are strict. We attempt to modify and run DB-BERT locally through Docker and state our problems overcoming the tightly coupled nature of its old NLP implementation. We review the general workings and algorithms of DB-BERT along with our test results.

Paper Chosen:

- Title: DB-BERT: A Database Tuning Tool that 'Reads the Manual'
- Publication Year: 2022
- Conference / Publisher: SIGMOD '22: Proceedings of the 2022 International Conference on Management of Data, Cornell University
- URL: https://dl.acm.org/doi/10.1145/3514221.3517843
- Author: Immanuel Trummer

Introduction

Databases have hundreds of tuning parameters (knobs) that database administrators (DBAs) adjust to optimize the performance of a database instance. Since manually adjusting these knobs is a cumbersome task, there is a concerted effort in the scientific community to automate the process of database tuning.

Automated tuning systems like those of DB-BERT use predefined performance metrics on a given workload to find parameter settings that optimize database performance. Our selected model, DB-BERT, goes by the principle of "When all else fails, read the instructions," [1]. Based on the BERT natural language model, DB-BERT analyzes database manuals to extract tuning hints for optimizing database parameters [2]. From these tuning hints, DB-BERT aims to reduce the manual effort and time to tune a database by automatically extracting recommended domain ranges for database knobs from documents and evaluating performance metrics subject to the extracted bounds.

One area we identified for improving DB-BERT was to reduce its hardware requirements and to try running it locally. To scale DB-BERT for smaller systems, we developed a Docker solution that encapsulates dependencies, automates setup, and ensures seamless deployment on local machines. The Docker file configures Python, PostgreSQL, MySQL, and benchmark initialization, enabling operation on weaker hardware.

Although we explored reimplementing DB-BERT using alternative NLP models like DistilBERT and RoBERTa for improved performance - the tightly coupled architecture of the original DB-BERT made further improvements infeasible within the project timeline. Our Docker solution still significantly enhances DB-BERT's accessibility and scalability for diverse environments.

Background and Framework

In traditional database tuning, the aim is to identify bottlenecks, such as from monitoring or performance analysis, then through query optimization, hardware tuning, knobs like shared buffers, connection limits, and max parallel workers, or the working RAM[1]. Automated Tuning methods like DB-BERT differ from this approach by using an LLM and learning to automatically adjust these parameters based on the benchmarks provided.

The model used in DB-BERT is a pre trained large language model. LLMs are trained on massive amounts of data and use a type of neural network, like a transformer model, to extract meanings from text and understand the relationships between words and phrases from data rich to sparse problems, and with no manual pre-processing required on input text[1]. Using LLMs addresses the lack of task-specific training data that automated tuning would otherwise lack.

DB-BERT uses Deep Reinforcement Learning (DRL) to balance exploration and exploitation when trying out different parameter / knob settings. It tries to balance immediate and future rewards based on its estimates, such as the Double Deep Q-Networks algorithm[3], alongside the pretrained BERT model.

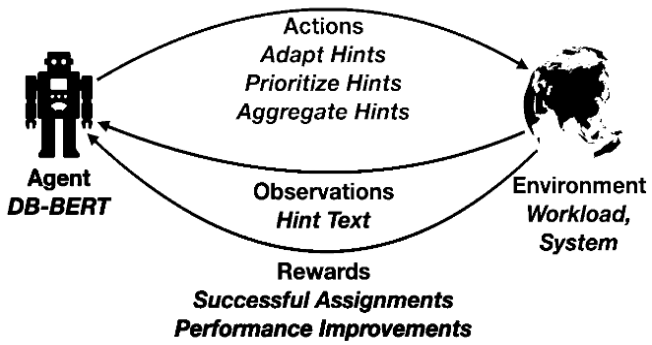

Figure 1: DB-BERT Training[1]

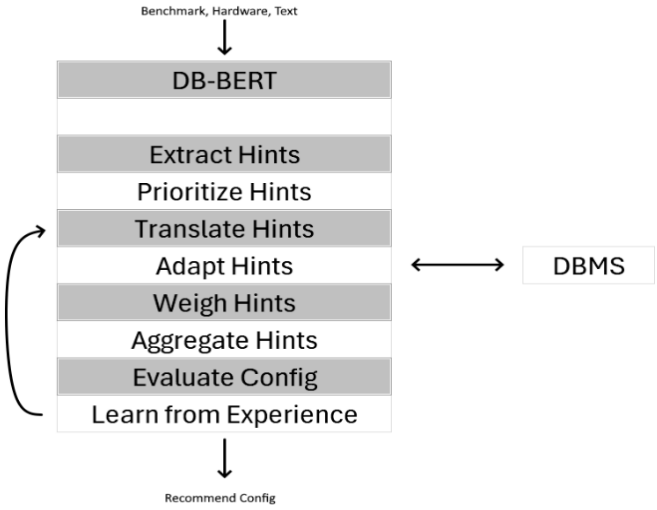

Figure 2: DB-BERT System[1]

Training data used by DB-BERT included information from public sites like Percona, StackExchange, or Google, providing input during training phase [1]. First, DB-BERT divides input text and tries to extract tuning hints from each snippet, where the set of candidate hints for a given text snippet is the Cartesian product between parameter/knob references and values. Then DB-BERT determines the order in which hints will be considered, translates the hint, and assigns values as a literal equation [1]. DB-BERT will use its past iterations to decide whether to deviate from the recommended value, assigning a weight to the hint. The system then compiles these equations as a set of parameters, evaluates from the given benchmark, then repeats. In the case of DB-BERT, the authors wish in future work to consider more diverse tuning objectives, as it does not optimize latency or throughput [1].

Survey Related Work with DB-BERT

OtterTune

Ottertune is an automated database tuning system that uses a combination of supervised learning models to arrive at an optimal knob configuration[4]. It uses techniques like lasso regression and k means clustering to identify dominant knobs for a set of queries. The final tuning is done using either Gaussian Process Regression (GPR), Deep Neural Networks (DNN), or Deep Deterministic Policy Gradient (DDPG) [4]. OtterTune maintains a repository of prior tuning activities and uses that as a base to fine tune itself to the given environment and query set.

When compared with DB-BERT, the following differences are apparent-

- Since OtterTune does not have any predefined rules for each database knob, it requires many iterations (and time) to set itself on the right path to reach an optimum.
- OtterTune often tries parameter changes that decrease performance significantly while attempting to vary its cost. DB-BERT is less prone to this phenomenon since ranges for each database knob are bounded based on expert knowledge.
- OtterTune has, however, been streamlined for many cloud databases, making it a more attractive option on the cloud.

CDBTune+

CDBTune+ [5] is an automated database tuning system that leverages deep reinforcement learning to optimize database performance. Similar to OtterTune, it internally uses a deep deterministic policy gradient (DDPG) algorithm to explore a high dimensional space of configuration knobs to arrive at the optimal value.

CDBTune+ is initially trained on a set of training quadruples <q,a,s,r>, where:

q - set of SQL queries for training

a - set of knob values

s - set of metrics that represent the database state at the time of query

r - the resulting performance metrics of the query (latency and throughput).

The Reinforcement Learning model uses a trial-and-error process to run over these quadruples and arrive at an optimal solution. Since there is no historical data present during the training phase, different benchmarks like sysbench and TPC-MySQL are used to get training queries. Once tuned, a user finetunes the model based on their query space. Because the DBMS configuration tuning problem that aims to find the optimal solution in continuous space is NP-hard [4], a trial-and-error algorithm is the best that can be achieved in this case.

On the other hand, DB-BERT parses database manuals to extract intuition from domain experts. Due to this, DB-BERT has several advantages over CDBTune+, such as-

- **Automatically assume best practices:** By "reading" the manual, DB-BERT can quickly identify optimal configurations based on best practices and recommendations found in documentation. CDBTune+ cannot do this since it solely relies on the query set presented to it.
- **Faster Convergence to Optimal Configurations:** As DB-BERT uses intuition gained from manuals along with fine tuning in the training phase, this hybrid approach allows it to converge on optimal configurations faster by starting with informed guesses from the manual, reducing the number of iterations needed to find good settings. This also allows the model to avoid settling at a local maxima. CDBTune+ may require more iterations and training overhead to reach a similar optimal setting.
- **Reduced Tuning Overhead:** DB-BERT's ability to extract hints from manuals helps reduce the overhead typically associated with reinforcement learning-based tuning systems. It can avoid unnecessary exploration by applying known good practices directly from the manual, which reduces the overall tuning time and computational cost. CDBTune+ may experience higher tuning overhead due to its reliance on DRL alone, which requires extensive exploration of the configuration space before finding optimal settings.

WATuning

A modification of CDBTune, WATuning is an attention-based deep reinforcement learning technique that can adapt to the changes in workload characteristics and optimize the system performance efficiently and effectively. WATuning[7] introduces a workload-aware multi-instance mechanism tuning system, which enhances the DDPG algorithm to adapt to varying workload characteristics.

While this multi-instance mechanism employs a shared neural network framework, its significant drawback is that it greatly increases the system's training time, which may hinder its practical applicability. [8] It can generate multiple instance models according to the workload change to complete targeted recommendation services for different

workloads. [7] WATuning works better in dynamic and write-intensive transactions. [6] This is because WATuning can dynamically fine-tune itself according to the constantly changing workload in practical applications so that it can better fit the actual environment to make recommendations [7]. It also improves database management and operational efficiency. [6]

Method	*Model/Workload*	*Throughput Improvement (%)*	*Latency Reduction (%)*
WATuning	Model0 (Read-Only)	383.73	59.52
	Model5 (40%-50% Write)	160.39	-
	Model11 (Write-Only)	984.66	-
CDBTune+	Model0 (Read-Only)	42.15	-43.95
	Model5 (40%-50% Write)	68.28	-51.65
	Model11 (Write-Only)	128.66	-61.35
OtterTune	Model0 (Read-Only)	44.46	-23.63
	Model5 (40%-50% Write)	29.8	-35.51
	Model11 (Write-Only)	91.25	-59.27

Figure 3: Survey of DB Knob Tuning [9]

As we can see in the above table [6], WATuning stands out, especially for write operations.

Most of the other previously discussed solutions only provide coarse-grained tuning since they evaluate the state of the database as a whole [7]. WATuning enhances this by incorporating workload-aware or query-aware technologies, allowing better performance on specific tasks. The core algorithm of WATuning, called ATT-Tune, is based on the Deep Deterministic Policy Gradient (DDPG) model. ATT-Tune introduces an attention mechanism that helps the system focus on relevant workload metrics when making configuration decisions.

When compared with DB-Bert, the following differences are apparent-

- DB-BERT surpasses WATuning through its innovative use of natural language processing (NLP), specifically with a BERT-based model, allowing it to interpret and act on unstructured text from tuning manuals—something WATuning struggles to achieve with its reliance on structured data and predefined metrics. This NLP-driven, context-aware approach enables DB-BERT to capture nuanced guidance and make adaptive tuning decisions that align closely with specific workload requirements.
- DB-BERT reduces the need for human intervention by autonomously processing complex instructions, while WATuning typically requires manual adjustments for scenarios outside its predefined configurations. The use of transfer learning further enhances DB-BERT's adaptability, allowing it to quickly incorporate new tuning guidelines without retraining from scratch. Hence, DB-BERT is more scalable across various database management tasks, whereas WATuning's focus on metric-based workload adaptation limits its broader applicability.

In this way, despite WATuning being the faster and more optimized solution to database tuning, DB-BERT's capability to interpret human language and adjust autonomously makes it a more flexible, powerful, and self-sustaining solution than WATuning.

QTune

Q-tune [10] attempts to solve the problem of automated knob-tuning by using a query aware database tuning system using a DRL model. The architecture optimizes SQL query performance through a structured workflow. Clients submit tuning requests to the **Controller**, which analyzes SQL queries and generates feature vectors based on query plans and estimated costs.

The **Tuner**, powered by a DS-DDPG (Double State Deep Deterministic Policy Gradient) deep reinforcement learning model, recommends continuous knob values for execution. It has the following components:

1. The **Environment** contains database information, including the inner state (tunable knob configurations) and outer metrics (key performance indicators that reflect database status but cannot be tuned). For example, in PostgreSQL, the inner state includes configurations like working memory, while outer metrics include the number of committed transactions and deadlocks.
2. The **Query2Vector** component generates feature vectors for queries or workloads.
3. The **Predictor** is a deep neural network that forecasts changes in outer metrics (ΔS) before and after query processing. The Environment combines the original metrics (S) with the predicted changes (ΔS) to simulate the new metrics (S') after executing the queries.
4. The **Agent** tunes the inner state based on the observation S' and consists of two modules: Actor and

Critic. The Actor outputs a vector of tuned knob configurations based on S′, while the Critic evaluates these actions by providing a Q-value that reflects their effectiveness. Both the Actor and Critic update their neural network weights based on the rewards received from the Environment, which indicates performance changes.

This collaborative process enables the model to recommend optimal configurations. The DS-DDPG model is particularly effective for continuous action spaces, making it well-suited for database tuning, where many knobs need to be adjusted continuously.

When compared to DB-Bert, QTune has the following advantages and disadvantages:

- QTune focuses more on real-time query-aware optimization. This gives fine-grained control over query-level, workload-level, and cluster-level tuning. DB-Bert utilizes external textual knowledge for tuning.
- QTune is more computationally intensive due to the need for real-time analysis of query features due to high complexity.
- QTune ignores historical data that can be easily mined, which DB-Bert takes advantage of reading the database manual. Due to this, it often adopts values for parameters that worsen performance before reaching an optimal state. It performs like WATuning and QTune, despite being the faster and more optimized solution to database tuning, lacks the flexibility that DB-BERT's capability to interpret human language and adjust autonomously offers.

Technical Section

Algorithms and Analysis

As discussed in the previous sections, (see figures 1 and 2), DB-BERT tries to obtain and adapt and aggregate hint values through multiple training episodes. The following algorithms are listed and documented in the original project by Trummer [1]. Details of the main algorithms used will be discussed here. After, we discuss implementation of DB-BERT and our attempts to modify it.

The main DB-BERT loop works by taking a collection of text snippets, T, which might contain optimization hints. Using the algorithms: *ExtractHints*, *OrderHints*, and *RunEpisode*, it orders these hints based on priority, limiting the number of hints to l per parameter. DB-BERT repeats these until a set limit:

a) Batch the ordered hints, up to e

b) Use hints to evaluate n configs each episode

DB-BERT then returns the best DBMS configuration found during the optimization process.

- *ExtractHints*

Searches for parameters P that are explicitly mentioned in the text t. If not specifically mentioned, by calculating the minimum distance δ between the text embedding of t (generated using the pretrained BERT model) and the embeddings of the parameters P - then iterates over said params. The result is a set of triplets <t, p, v>, where each triplet associates the text with a parameter and value.

- *OrderHints*

Prioritizes a collection of candidate hints 'H' based on their associated parameters and a stride length 'l' (determining how many hints per parameter are processed in each range). It guesses based on important parameters being mentioned more often, and to expect diminishing returns for similar hints. It extracts the params P from H, groups by param, sort based on the number of associated hints, iterates over hint ranges, and returns R as an ordered list of hints.

- *TStep*

Transition function used by DB-BERT to translate single hints, where the original author notes its similarity to the step function in the corresponding OpenAI Gym Env[1]. If the initial decision given by 'd' is 0, it ignores comparing it to the DBMS Set and just returns the hint usage. Otherwise, TStep further refines its function (Also in *RunEpisode*), 'f,' using a multiplier, 'M,' to update the formula based on given parameters from the database. If successful, the algorithm evaluates the performance and returns a reward for further training.

- *RunEpisode*

Related to TStep and the AI part of DB-BERT, and processes the set of optimization hints by translating,

weighing, and evaluating them. From a batch of candidate hints as input, it iterates over those hints and uses the **TStep** function to translate single hints and to determine that a candidate hint is erroneous and should not be considered). If successful at translating the current hint into an arithmetic formula, it assigns a weight based on the ran learning agent, ChooseAction, then returns all weighted tuning hints.

Implementation

Trying to run DB-BERT posed significant challenges due to its reliance on outdated packages, an older Python 3.8 environment, and high resource demands, making it impractical to run on standard EC2 instances. Running it locally further compounded these issues, with dependency conflicts and compatibility problems arising from outdated libraries clashing with newer versions in the system.

Several resulting build issues came from running DB-BERT and resulted in multiple versions of the same dependencies and disrupting the dev environment. Testing and resolving these conflicts in a local environment was error prone and tedious, often impacting other projects. Moreover, running the application on an EC2 instance introduced storage and resource limitations, making it unsuitable for large datasets and extended operations.

- *Transition to Docker*

Recognizing the limitations of the initial approach, we adopted a Docker-based strategy. We decided to adopt the Docker approach as it can encapsulate all dependencies and configurations - ensuring consistency, avoiding conflicts, enabling seamless execution without impacting the local system, while also providing an easily replicable and scalable environment for future use.

We crafted a Docker file to automate the build process, ensuring the installation of all necessary dependencies and the setup of PostgreSQL and MySQL databases. This approach also facilitated the integration of TPC-H and JOB benchmarks. We transitioned to the Docker approach because running DB-BERT locally led to dependency clashes caused by multiple versions of required packages.

Using Docker, we created an isolated environment where dependency issues could be resolved without affecting the local system or other projects. Docker also provided the flexibility of a full Ubuntu-based system and eliminated resource constraints, ensuring sufficient storage and computational capacity. Additionally, Docker simplifies the installation process for future users. Local isolation, scalability, and ease of use made Docker our choice for modernizing DB-BERT.

- *Environment Info*

The provided Docker file automates the setup of a consistent, containerized environment for DB-BERT. It uses Ubuntu 22.04 as the base image, installs necessary dependencies like Python 3.10, PostgreSQL, and MySQL, and configures database users and permissions. The Docker file copies the application source code and data files into the container and sets up a Python virtual environment to isolate dependencies. Initialization scripts populate benchmark datasets, while environment variables are configured to ensure smooth application execution. The container exposes port 8501 for the Streamlit interface and sets the entry point to run the application. This approach simplifies setup, ensures compatibility, and isolates the application environment for reliable deployment.

- *Addressing Dependency Conflicts*

The transition to Docker presented numerous package conflicts as well, primarily due to outdated dependencies. The Original DB-BERT implementation is not very modular. We addressed these concerns by upgrading the Python version to Python 3.10 and updating most dependencies to versions that were compatible with each other. This process was time-consuming, as each change required rebuilding the Docker image, taking an hour to an hour and a half per iteration due to the project's scale. Despite these challenges, we eventually resolved all dependency-based and transformer-based conflicts.

- *Reimplementation Efforts*

The 'NlpTuningEnv' class in zero_shot.py is a sophisticated reinforcement learning environment that combines natural language processing with database tuning. It allows an agent to learn optimal tuning strategies by interpreting natural language hints, making it a powerful tool for automating and improving database performance. The design emphasizes flexibility in hint

processing, a structured reward system, and detailed logging for performance analysis.

We attempted to replace BART with DistilBERT for efficiency, RoBERTa for improved performance, T5 for versatility in handling various NLP tasks and XLNet for enhanced contextual understanding and performance on classification tasks. This endeavor was complicated by the tightly coupled nature of the project's modules. Changing the base model led to cascading errors, complicating the reimplementation process. Given the constraints of our timeline, we found a complete reimplementation to be infeasible within the project's timeframe. Then, evaluation and benchmarking were done on the dockerized, lightweight version of DB-Bert.

Evaluation

Experimental Setup

Since the experiments of the original paper were done with pricey p3.2xlarge EC2 instances with 61GB RAM and extra GPU power, our experiments were done on an Ubuntu:22.04 docker-container running with 4 CPUs, 14GB RAM, 2GB Swap space and 248 GB Virtual Disk space instead. Postgresql-15 and MySQL server 8 were used with Python 3.10.

Evaluation Process

A total of 232 system parameters for Postgres and 266 system parameters for MySQL were evaluated, attempting to replicate the configurations attained in the original paper.

DB-BERT employs reinforcement learning to choose multiplier values and weights for each hint from a predefined set of options. In all experiments, the multipliers are selected from the set {1/4, 1/2, 1, 2, 4}, while the weights are chosen from {0, 2, 4, 8, 16}, as done in the original paper. The environment configuration is also provided since certain parameters are extracted in a ratio form (For example, RAM must be set to 25%).

Extracted Tuning Hints

	Document	Text	Parameter	Frequency	Value
18	15	checkpoint_flush_after | …	max_wal_size	2	16384
19	15	checkpoint_flush_after | …	checkpoint_warning	2	300
20	15	checkpoint_flush_after | …	checkpoint_warning	2	300
21	15	checkpoint_flush_after | …	checkpoint_timeout	3	32
22	15	checkpoint_flush_after | …	checkpoint_timeout	3	4096
23	57	shared_buffers = 1GB # u…	checkpoint_timeout	3	20min
24	15	checkpoint_flush_after | …	min_wal_size	2	1024
25	15	checkpoint_flush_after | …	min_wal_size	2	16384
26	15	max_wal_size sets the m…	max_slot_wal_keep_size	2	1GB

Evaluated Parameter Settings

	Elapsed (ms)	Evaluations	Configuration	Performance	Best Configuration
0	205,375.3552	1	{}	0.3206	{}
1	231,781.8740	2	{"max_connections":"400…	0.0552	{}
2	234,819.3315	3	{"max_connections":"50",…	0.0437	{}
3	259,755.1340	4	{"checkpoint_completion…	0.0603	{}
4	262,795.1008	5	{"checkpoint_completion…	0.0547	{}
5	284,877.9580	6	{"cpu_index_tuple_cost":…	0.0549	{}
6	287,953.8511	7	{"cpu_index_tuple_cost":…	0.0715	{}
7	309,303.8330	8	{"checkpoint_warning":"…	0.0574	{}
8	312,346.7627	9	{"checkpoint_warning":"…	0.0388	{}

DBMS Tuning Decisions

	Parameter	Recommendation	Inferred Type	Rec. Value
0	shared_buffers	25% of available RAM	Relative (RAM)	2000000.0
1	shared_buffers	25%	Relative (RAM)	2000000.0
2	shared_buffers	256MB	Absolute Value	256.0 MB
3	shared_buffers	1024MB	Absolute Value	1024.0 MB
4	shared_buffers	1GB # up to 8GB	Absolute Value	1.0 GB
5	shared_buffers	1GB # up to 8GB	Absolute Value	8.0 GB
6	max_connections	100 concurrent connections	Absolute Value	100.0
7	max_connections	100	Absolute Value	100.0
8	max_connections	275	Absolute Value	275.0
9	max_connections	275	Absolute Value	275.0

Running for up to 1500 seconds, 1000000 frames

Step 0 - tuned for 5.555432081222534 seconds

Step 500 - tuned for 745.1984438896179 seconds

Step 1000 - tuned for 1375.2303886413574 seconds

DB-BERT: finished tuning session.

Fig: Running DB-Bert on Postgres against TPC-H Queries

At most ten hints per episode per parameter were considered. We consider at most 50 hints per episode in total and evaluate two configurations per episode DB-BERT splits text documents into segments of length at most 128 tokens. Focusing on latency evaluation, we compare DB-BERT against baselines on TPC-H. We tune Postgres and MySQL for 25 minutes per run. Documentation for PostgreSQL (postgres100) and MySQL (MySQL 100) are used to tune the database before running the TPC-H evaluation queries.

Extracted Tuning Hints

	Document	Text	Parameter	Frequency	Value
0	4	innodb_buffer_pool_size …	innodb_buffer_pool_size	22	50%
1	4	innodb_buffer_pool_size …	innodb_buffer_pool_size	22	70%
2	4	Hi Jie Zhou. The MySQL d…	innodb_buffer_pool_size	22	1GB
3	15	query_cache_size = 0 que…	innodb_buffer_pool_size	22	8
4	15	query_cache_type = 0 inn…	innodb_buffer_pool_size	22	5G
5	21	The number of regions th…	innodb_buffer_pool_size	22	1
6	21	you can increase innodb_…	innodb_buffer_pool_size	22	2
7	21	you can increase innodb_…	innodb_buffer_pool_size	22	3
8	21	you can increase innodb_…	innodb_buffer_pool_size	22	4
9	21	you can increase innodb_…	innodb_buffer_pool_size	22	8

DBMS Tuning Decisions

	Parameter	Recommendation	Inferred Type	Rec. Value
0	innodb_buffer_pool_size	50% – 70% of your total R…	Relative (RAM)	4000000.0
1	innodb_buffer_pool_size	50% – 70% of your total R…	Relative (RAM)	5600000.0
2	innodb_buffer_pool_size	1GB or more	Absolute Value	1.0 GB
3	innodb_buffer_pool_size	8	Absolute Value	8.0
4	innodb_buffer_pool_size	5G	Absolute Value	5.0 G
5	innodb_buffer_pool_size	1 gigabyte or more	Absolute Value	1.0
6	innodb_buffer_pool_size	2, 3, 4 or 8	Absolute Value	2.0
7	innodb_buffer_pool_size	2, 3, 4 or 8	Absolute Value	3.0
8	innodb_buffer_pool_size	2, 3, 4 or 8	Absolute Value	4.0
9	innodb_buffer_pool_size	2, 3, 4 or 8	Absolute Value	8.0

Fig: Running DB-Bert on MySQL against TPC-H Queries

Key Findings

While training time does increase based on the size of the system, the dockerized version of DB-Bert managed to arrive at a solution that demonstrated performance improvement and was consistent with the results of the original paper.

Following are the parameters extracted by DB-BERT for PostgreSQL and MySQL on TPC-H:

Parameter	**Rec Value**
Postgres	
Max_connections	100
max_parallel_workers_per_gather	19
Max_wal_size	1GB
Shared_buffers	256MB
MySQL	
innodb_buffer_pool_size	1GB
join_buffer_size	2GB
max_connections	88
innodb_log_file_size	2GB

- The difference in value is expected since the system used is much smaller than the system used in the original paper.
- DB-BERT does not explore negatively correlated configurations before achieving a positive one.

Conclusion

We covered how DB-BERT works and explored scaling the architecture to function effectively on smaller, resource-constrained systems by leveraging a Docker approach. With Docker, we encapsulated all dependencies and configurations - along with overcoming dependency conflicts and outdated libraries. Despite limited hardware compared to the high-performance EC2 instances used in the original study, our Docker-based setup achieved performance improvements consistent with the results reported in the original paper. This validates Docker's capability to make DB-BERT accessible for diverse environments, enhancing its scalability and usability.

While we also attempted reimplementing DB-BERT to replace its NLP model with alternatives like DistilBERT, RoBERTa, and XLNet for better efficiency and versatility, the project's tightly coupled architecture posed significant challenges. The cascading errors resulting from changes to the base model and time constraints made a complete reimplementation infeasible. However, the integration of DB-BERT within the Docker environment allowed us to demonstrate its effectiveness in database tuning. Our evaluation on TPC-H benchmarks highlighted its capability to optimize configurations with minimal overhead, even on constrained systems.

Despite the challenges encountered, our efforts culminated in the successful modernization of DB-BERT. We developed a Docker file that automates the build process, allowing users to run DB-BERT by simply building the Docker image. This solution mitigates the need for powerful EC2 instances, making the tool accessible on standard systems. Although we could not fully reimplement DB-BERT's algorithms due to time constraints and the intricacies of its architecture, our work ensures that DB-BERT can be reliably run in a modern environment, paving the way for future enhancements and optimizations.

References

(1) Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool that "Reads the Manual". In Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22). Association for Computing Machinery, New York, NY, USA, 190–203. DOI: https://doi.org/10.1145/3514221.3517843.

(2) Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), 4171–4186. Association for Computational Linguistics, Minneapolis, Minnesota.

(3) Hado Van Hasselt, Arthur Guez, and David Silver. 2015. Deep Reinforcement Learning with Double Q-Learning. In Proceedings of the AAAI Conference on Artificial Intelligence, vol. 30. DOI: https://doi.org/10.1609/aaai.v30i1.10295.

(4) 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), 1009–1024. Association for Computing Machinery, New York, NY, USA. DOI: https://doi.org/10.1145/3035918.3064029.

(5) Jiahui Zhang, Kai Zhou, Guoliang Li et al. 2021. An efficient deep reinforcement learning-based automatic cloud database tuning system. The VLDB Journal, vol. 30, 959–987. DOI: https://doi.org/10.1007/s00778-021-00670-9.

(6) Sandeep Sheoran and Deepak Kumar Singh. 2024. Analysis of Machine Learning Techniques for Knob Tuning. In 2024 IEEE International Conference on Computing, Power and Communication Technologies (IC2PCT), Greater Noida, India, 1824-1830. IEEE Press. DOI: https://doi.org/10.1109/IC2PCT60090.2024.10486503.

(7) Jian-Kang Ge, Yi-Fan Chai, and Yan-Ping Chai. 2021. WATuning: A Workload-Aware Tuning System with Attention-Based Deep Reinforcement Learning.Journal of Computer Science and Technology, vol. 36(4), 741–761.DOI: https://doi.org/10.1007/s11390-021-1350-8.

(8) Jiahui Wang, Liang Chen, Lei Yang and Min Wang .2023 HWTune: A Hardware-Aware Database Tuning System with Deep Reinforcement Learning.In Proceedings of the International Conference on Data-driven Optimization of Complex Systems (DOCS) Tianjin China pp1-8.DOI: https://doi.org/10.

(9) Jiahui Wang, Liang Liu, and Cheng Li. 2023. A Survey of Database Knob Tuning with Machine Learning. In Proceedings of the 2023 IEEE 2nd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA), 1545–1550. DOI: https://doi.org/10.1109/EEBDA56825.2023.10090597.

(10) Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. Proceedings of the VLDB Endowment, 12, 12 (August 2019), 2118–2130. DOI: https://doi.org/10.14778/3352063.3352129.

URL - https://github.com/JoelJacobStephen/DB-Bert-Mod

The above URL contains the Source Code, program, and dataset in order to replicate the experiment's findings. The ReadMe file consists of everything needed to run DB-BERT smoothly using step-by-step instructions.

Contributions of Each Group Member

Steps	Actions	Division of Work
Setup and Understanding	Goal: Setup DB-BERT environments for MySQL and Postgres databases. Action: • Install dependencies, set up cloud environments • Setup MySQL and Postgres. • Discover data and fill databases.	Joel Mathew Peter
Running of DB-BERT and Dockerization	Goal: Modernize and optimize the deployment of DB-BERT to resolve dependency conflicts, improve resource efficiency, and ensure scalability and usability. Action: • Transitioned from local and EC2-based setups to a Docker-based solution, encapsulating dependencies / configurations for consistency and scalability. • Designed a Docker file to automate the build process, integrate PostgreSQL and MySQL databases, and facilitate TPC-H / JOB benchmark initialization. • Addressed dependency conflicts by upgrading to Python 3.10, updating outdated libraries, and iteratively resolving compatibility issues through extensive testing and rebuilds. • Explored model reimplementation with alternative NLP models like DistilBERT and RoBERTa *(Deferred due to tight coupling and project time constraints).*	Punya Mathew Samiha Joel
Reimplementation, Experimentation and Benchmarking	Goal: Replicate the experiments on TPC benchmarks for MySQL and Postgres, as detailed in the paper. Action: • Attempt to reimplement DB-BERT with various models.	Punya Mathew
Performance Analysis and Final Reporting	Goal: Analyze the results and document the findings. • Compile and update our technical report. • Present key findings along with the software deliverables.	Joel Mathew Peter Punya Samiha