# An Offline Profile-Guided Optimization Strategy for Function Reordering on Relational Databases

Weibin Chen[1] and Yeh-Ching Chung[2]

*Abstract*— **Profile-guided optimization (PGO) is an advanced technique used to improve the performance of Relational Databases (RDBs). However, the most common strategy is to perform profiling on the production environment, which can lead to instability and performance loss for the Database System. To address this issue, we propose an offline profiling strategy that uses a query reduction strategy to obtain a reduced query set from the production environment's log file. We then run this sample on an offline test environment to collect profile data and use it to conduct function reorder optimization.**

**To evaluate our approach, we compared the performance improvements achieved by running a reduced query set and a full query set on a MYSQL database. We generated function call graphs for both query sets and observed that the performance improvement achieved by the reduced query set was slightly lower than that of the full query set. These results demonstrate the effectiveness of our approach and highlight the potential benefits of offline profiling strategies for improving the performance of RDBs in production environments, while also avoiding performance losses and increasing stability.**

*Index Terms*— **Profile-guided Optimization, Query, Relational Databases, Function Reorder, Function Call Graph**

## I. INTRODUCTION

In recent years, Relational Databases (RDBs) have become increasingly popular for a wide range of applications across various industries, including Enterprise Information Systems [1], Transportation Systems [2], and others. As the volume of data processed by RDBs continues to grow, the performance of low-performing databases can significantly impact the quality of life. Therefore, there is an increasing demand to optimize the performance of database operations to address these challenges.

One promising approach to improving RDBs performance is Profile-guided Optimization (PGO), which involves using profiling data collected during database operation to guide optimization of the database system. PGO is a technique used to optimize the performance of software applications by collecting and analyzing performance data through profiling. This strategy aims to improve the efficiency of database operations by identifying and addressing performance bottlenecks. Numerous PGO tools have been developed with the aim of enhancing code locality, including AutoFDO [3], HFSort [4], PLTO [5] and BOLT [6]. These tools have demonstrated the ability to improve software performance by a range of 3% to 30%.

[1]Weibin Chen is with School of Science and Engineering, The Chinese University of Hong Kong (Shenzhen), Shenzhen, China `weibinchen1@link.cuhk.edu.cn`

[2]Yeh-Ching Chung is with School of Data Science, The Chinese University of Hong Kong (Shenzhen), Shenzhen, China `ychung@cuhk.edu.cn`

In the field of database systems, previous studies have extensively explored the concept of online PGO [7] [8]. One such approach involved collecting profile data from a full execution query set using a tool such as perf [9] and extracting a function call graph from this profile data. This function call graph was then used to generate a new function order for program recompilation.

However, an alternative approach is offline PGO, which involves performing profiling in a non-production environment. There are two key reasons why an offline profiling strategy for function reordering is necessary despite the potential optimization loss compared to online method. Firstly, the use of profiling for collecting data in a real production environment may result in program instability, as perf adds additional trace code in the kernel that can lead to unexpected issues. Secondly, profiling requires system resources for data collection, which can potentially impact the performance of the production environment.

In this paper, we investigate the advantages of an offline profiling optimization approach for function reordering in the context of RDBs. We present the concept and methodology of our approach, which involves obtaining the query execution log file from the original production environment and identifying the number of queries for each template. The template queries are those with the same predicate. We select a subset of the template queries as the running queries to collect the profiling data and perform the profiling with function reordering. We evaluate the performance impact of PH [10], $C^3$ [4], and RL [11] on this offline profiling strategy.

The contributions of this paper are:
- Implementation of an offline framework that generates template SQL queries for PGO.
- Evaluation of the impact of offline and online PGO strategies for function reordering using three different existing algorithms on MYSQL [12].

The paper is organized as follows. We first introduce the optimization procedures of both traditional PGO and offline PGO for databases in Section II. Next, in Section III, we present the idea of graph similarity to support our assumption that executing similar queries will result in a similar function call graph. Section IV describes the experimental results of full-data PGO and offline PGO on MYSQL. Finally, Section V concludes the paper, and Section VI provides an overview of previous research on RDBs performance optimization.

## II. OPTIMIZATION PROCEDURE

In this section, we present a detailed description of the implementation process for the optimization procedure, aimed

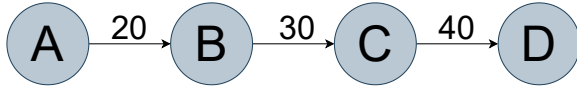Fig. 1. Procedure of profile-guided optimizaiton for function reordering



Fig. 2. Example of function call graph

at improving the performance of RDBs by recompiling them with a new function order. We first introduce the PGO approach, which serves as the basic method for optimizing the databases. Given the inherent features of RDBs, all entries in the log comprise SQL queries. As most of these queries are highly probable and may be repeated with slight variations in certain values, we can take advantage of this characteristic to minimize the profiling scale and perform offline PGO, as proposed next.

### A. Profile guided optimization for function reordering

Profile-guided optimization (PGO) is a software development technique that aims to improve program performance by analyzing the execution profile of the code. Function reordering is a specific optimization approach that involves changing the order of function execution to optimize branching and cache locality. In the context of PGO, function reordering is based on profiling data collected during program execution, which identifies frequently executed functions and their execution order. This information is then used to generate a new function order that can be applied to the code to enhance its performance. The procedure for performing function reordering using PGO typically involves several steps, including profiling the code, analyzing the profile data, generating a new function order, and applying the new order to the code.

In our study, we utilized the Perf [9] profiling tool to gather system and hardware performance metrics such as CPU usage, memory usage, and disk I/O while the database was running. The procedure for function reordering is depicted in Fig. 1. After analyzing the profile data, we extracted the function call graph, as shown in Fig. 2, which comprises a set of functions (V) and arcs (A) with weights that indicate the frequency of function calls. We applied state-of-the-art function reordering algorithms to generate a new function order. Finally, we utilized the LLVM [13] feature to relink the program and generate a new binary.

This technique can improve the efficiency of the instruction translation lookaside buffer (I-TLB) by reducing the number of I-TLB misses. The I-TLB caches page table entries for recently accessed pages. If an entry is not present in the I-TLB, it results in an I-TLB miss, and the processor must access the page table from memory, which is a time-consuming operation. By rearranging functions to maximize the reuse of pages already loaded in the I-cache, the probability of I-TLB misses is reduced, resulting in improved

I-TLB efficiency. Enhancing the efficiency of the I-TLB can lead to better program performance.

### B. Offline Procedure

Based on the observation that executing similar queries results in similar function calls, we have developed an offline profiling strategy to guide the optimization of database binaries. This strategy eliminates the need for profiling on the production environment, thereby minimizing the impact on system performance. An overview of our strategy is shown in Fig. 3. Our strategy consists of three main parts. The first step, "generate templates," involves aggregating a large number of queries into a smaller set of template queries and counting the number of queries that belong to each template. The second part, "collect call graph", involves running traditional PGO steps to collect function graphs and generate a new function order. The third part, "optimize binary", involves using the LLVM characteristic to relink the database binary with the new function order.

In general workloads, SQL queries play a vital role in controlling database operations. Users often interact with dashboards or reporting tools that provide an interface to construct queries with various predicates and input parameters. These queries often share similarities in terms of execution frequency and resource utilization. By aggregating a large number of queries with identical templates, we can approximate the workload's characteristics. This reduces the number of queries needed to generate templates and only requires maintaining a small set of templates, thereby minimizing the impact on system performance.

The reduction process for each query involves two steps. Firstly, we extract all constants from the query string and replace them with value symbols. This step transforms all queries into a template format. There are three types of constants: those following WHERE clause predicates, those in SET fields of UPDATE statements, and those in INSERT statements. As illustrated in Fig. 3, for example, `UPDATE admin SET value = 1` is recognized as `UPDATE admin SET value = x`, where we use the symbol 'x' to replace the values. This way, `UPDATE admin SET value = x` is recognized as a template.

Secondly, we count the number of occurrences of each query under each template to determine the actual number of queries associated with each template. In order to provide users with more flexibility, our approach allows adjusting the reduction percentage according to the characteristics of different query sets. This feature enables users to fine-tune the trade-off between accuracy and reduction ratio based on their specific needs and requirements. In this paper, we select the top 10 templates based on this count, and each query is then attributed to one of the selected templates. We then execute 10% of the queries belonging to the top template queries to collect data.

### III. GRAPH SIMILARITY

Because our optimization is based on the assumption that the function call graph of similar query executions is similar,
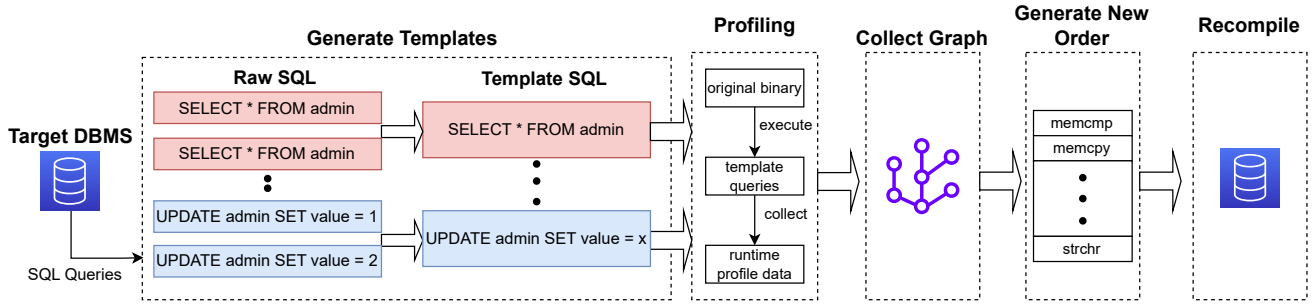
Fig. 3. procedure of offline profile-guided optimization for reordering on database

|                                | Similarity Percentage |
|--------------------------------|-----------------------|
| Jaccard similarity (nodes)     | 92.7%                 |
| Jaccard similarity (out-edges) | 91.9%                 |
| Jaccard similarity (in-edges)  | 91.9%                 |

in this section, we use Jaccard Similarity [14] to preliminarily verify the similarity between a graph profile by running a reduction and full sample.

Jaccard similarity is a measure of similarity between two sets, defined as the size of the intersection of the sets divided by the size of the union of the sets. In other words, it measures the proportion of shared elements between two sets out of the total elements in both sets. To calculate the Jaccard similarity between two graphs, Algorithm 1 shows the calculation process of Jaccard similarity of nodes, out-edges, and in-edges. We first calculate the intersection and union of the sets for each element type separately. Then, we compute the Jaccard similarity coefficient as the size of the intersection divided by the size of the union for each element type. The output is Jaccard similarity of nodes ($J_N$), out-edge ($J_{out}$) and in-edge ($J_{in}$).

We evaluated the two function call graphs generated by the queries presented in Table II, which were created using SYSBENCH [15], a MySQL test benchmark. The large dataset contains three times as many queries as the small dataset, and "x" denotes queries with different values. Queries in the small dataset were randomly selected from the large dataset. By running the queries in both datasets, we collected over 4000 edges and 1500 functions. The results of the computation are shown in Table II, where the Jaccard similarity was calculated for three different evaluation indexes. The Jaccard similarity was found to be greater than 90%, indicating that executing a small sample of the query dataset can gather a function call graph that is similar to the one generated by the large dataset.

## IV. EXPERIMENT

### A. Experiment Platform

Our experiments were executed on a Linux Ubuntu 18.4 server powered by an Intel Xeon E5-2640 CPU running at

---

**Algorithm 1:** Calculate Jaccard Similarity between Graphs

---

**Input:** Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where $V_1$, $V_2$ are sets of nodes, $E_1$, $E_2$ are sets of edges

**Output:** Jaccard similarity between the two graphs for nodes, out-edges, and in-edges

1   $N_1$ = set of nodes in $G_1$;
2   $N_2$ = set of nodes in $G_2$;
3   $E_{out,1}$ = set of out-edges in $G_1$;
4   $E_{out,2}$ = set of out-edges in $G_2$;
5   $E_{in,1}$ = set of in-edges in $G_1$;
6   $E_{in,2}$ = set of in-edges in $G_2$;

7   $J_N = |N_1 \cap N_2| / |N_1 \cup N_2|$;
8   $J_{out} = |E_{out,1} \cap E_{out,2}| / |E_{out,1} \cup E_{out,2}|$;
9   $J_{in} = |E_{in,1} \cap E_{in,2}| / |E_{in,1} \cup E_{in,2}|$;

10   **return** $J_N$, $J_{out}$, $J_{in}$;

---

2.4GHz, with 192 GB of random access memory. We used a self-implemented tool to perform profile-guided optimization (PGO) [11]. Profile data was collected using Linux's perf, version 5.3.18.

### B. Benchmark

For validation, we used the popular open-source database program MYSQL, version 8.0, compiled with GCC 7.5 [16] at the -O2 optimization level. We generated the experiment queries and data using SYSBENCH. To test the performance of MYSQL, we used two sets of queries: a small set consisting of 780,000 queries, and a large set with 1,800,000 queries. For the small query set, we used three client threads to execute queries on five tables with 500,000 records each, generated by SYSBENCH. For the large query set, we used 20 tables with 500,000 records each. We evaluated the performance of MYSQL using SYSBENCH's transaction per second (TPS) metric.

### C. Algorithm

We evaluated the performance of three algorithms, which are introduced as follows:

- **Pettis-Hansen (PH)**: The Pettis-Hansen algorithm [10] is a classic "bottom-up" function reorder algorithm

TABLE II

NUMBERS OF QUERIES IN.THREE TEMPLATE

| | Full Queries Set | Reduced Queries Set |
|---|---|---|
| SELECT c FROM sbtest2 WHERE id=x; | 281916 | 93972 |
| SELECT SUM(K) FROM sbtest2 WHERE id BETWEEN x AND x; | 28188 | 9396 |
| SELECT c FROM sbtest4 WHERE id BETWEEN x AND x; | 28080 | 9360 |

that works by iteratively swapping adjacent pairs of functions in a sequence until a desirable ordering is achieved. The algorithm evaluates the effectiveness of each swap by comparing the sum of the distances between adjacent functions before and after the swap. The process continues until no further improvements can be made.

- **Call-Chain Cluster (C3)**: The Call-chain cluster ($C^3$) [4] algorithm is the latest "bottom-up" method used for function reordering in software optimization. This algorithm is designed to store functions in clusters with a limit on the page size. If both clusters are larger than the page size, they are not merged. Before placing each function in a cluster, $C^3$ sorts all functions in the call graph in descending order of hotness. The hotness of clusters is judged based on the sum of the weights of the clusters' incoming arcs, as determined by $C^3$. Subsequently, $C^3$ starts appending the function to the end of the cluster of its most common caller. $C^3$ stops merging clusters when the size of any two clusters that can be merged is greater than the page size. Finally, $C^3$ sorts the final cluster based on its "density".

- **Reinforcement Learning (RL)**: The reinforcement learning algorithm [11] is the first "top-down" approach for function reordering. It uses Q-learning, which is a popular reinforcement learning algorithm. In Q-learning, the agent maintains a Q-table, which stores the estimated values (Q-values) for each possible action (i.e., selecting the next function to visit) in each possible state (i.e., the current function visited). The agent selects actions based on the highest Q-value for the current state and updates the Q-values based on the rewards received and the maximum Q-value of the next state. Through repeated iterations of exploration and exploitation, the agent learns an optimal policy for selecting actions that result in new function order with the maximum call frequency.

### D. Result

We evaluated two different types of query sets, and all experiments were conducted five times. The middle three values were taken as the result, and we set the profiling frequency as 9999 samples per second for perf in order to collect full data for better performance improvement. Due to the I/O limitations typically imposed on queries involving inserts and updates, performance variance in testing is often significant. Therefore, all queries tested in this study were select statements.

Fig. 4 and Fig. 5 show the results of the two different sizes of query sets. Both experiment results were compared using
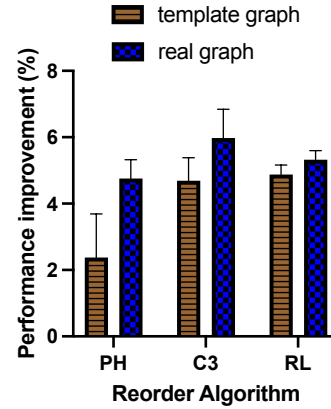


Fig. 4. Performance improvement in the execution of 780000 queries relative to the baseline.

two function call graphs: one was the real graph profiling the original size of the query set, and the other was the template graph profiling the reduced query set, as described in Section II. For the reduced query set, we only selected the top 10 template queries and used only 10% of the queries that belonged to these queries. Table III shows the number of queries for each experiment. The results show that using our optimization strategy on the smaller query set with 780,000 queries can result in a performance improvement of about 2% to 6%. However, running our optimization strategy on a larger query set resulted in a better performance improvement of about 6% to 9%. The performance improvement achieved using the template graph is mostly less than 2% compared to the improvement achieved using the real graph. It is interesting to note that the $C^3$ algorithm was slightly better than RL, while PH always had the worst improvement.

If the program conducts online profile-guided optimization, the program may suffer a performance reduction. Therefore, we also considered the performance loss when profiling. We conducted two performance profiling frequency settings on Perf: one was 9999 samples per second, and the other was 99 samples per second. Setting a higher profiling frequency can collect more profile data. The results show that profiling with a frequency of 9999 samples per second resulted in an 8.8% performance reduction, which is greater than the performance improvement on PGO. Therefore, using a frequency of 9999 samples per second greatly affects the program running in a production environment. The other setting, with a profiling frequency of 99 samples per second, is a common profiling frequency in production environments and resulted in about a 0.8% performance reduction. While this is more reasonable, it still results in performance reduction.
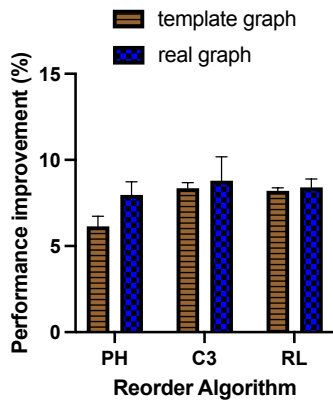
**970**

Fig. 5. Performance improvement in the execution of 1800000 queries relative to the baseline.

| | Small Query Set | Large Query Set |
|---|---|---|
| Select Queries | 780000 | 1800000 |
| Templates | 25 | 55 |
| Template Select Queries | 66056 | 117484 |

## V. CONCLUSION

In this paper, an offline profiling strategy is proposed for optimizing the performance of relational database systems. The technique is evaluated on the open-source MYSQL database using two different query sets, and compared with the commonly used online profiling strategy. Experimental results show that both methods can improve the performance of the database system by 3% to 9%, with the offline method achieving slightly lower improvement compared to the online method. Especially in the case of $C^3$ and RL, which have a better optimization effect, the performance improvement between the template graph and real graph is less than 2%. Furthermore, it is shown that the high sampling frequency used in the online method can lead to up to 8.8% performance loss on the production environment, while the low sampling frequency can cause a loss of up to 0.8%. Therefore, the offline profiling strategy offers a promising alternative that can not only achieve similar performance gains without impacting the production environment but also profile with a high profiling frequency.

Future work could explore the use of prediction techniques to forecast the future query set and improve profile-guided optimization, potentially leading to better adaptation to the production environment.

## VI. RELATED WORK

In recent years, optimizing the performance of relational databases has become a critical research topic in the field of database management. As data complexity and the need for faster query processing continue to grow, it has become essential to improve the performance of relational databases. In this section, we present a summary of previous work on relational database optimization, focusing on both tuning and index optimization. These methods, including the PGO optimization method employed in this article, are specifically designed to improve query performance.

Tuning techniques are methods that aim to enhance the throughput or decrease the latency of a particular workload by adjusting a set of configurable parameters known as tuning knobs. Database configuration tuning can be performed through hard-coded rules or heuristics, which can be recommended by database vendors or provided by dedicated tuning tools. These tools aim to determine the optimal allocation of resources [17] [18] or identify bottlenecks caused by misconfigurations [19]. For instance, BestConfig [20] adopts several heuristics to find a suitable configuration.

Another important development is the use of indexing techniques. Indexing is a way of organizing data in a database to improve query performance. By creating indexes on specific columns, the database can quickly locate the data that matches the query criteria. Choenni et al. created a set of candidate columns that includes all possible single columns to gain pure benefit [21]. During each iteration, one candidate is dropped, and the cost of processing the workload is evaluated until a specific index number is reached. To obtain per storage benefit, Valentin implemented a DB2 advisor. The proposed method utilizes the optimizer and hypothetical indexes to generate the set of candidate indexes. This set is sorted in decreasing order of benefit-per-space, and a greedy selection process is applied until the storage budget is reached. Further, random substitutions of the selected indexes are performed to identify lower costs due to index interactions. The approach aims to improve the performance of relational databases by efficiently using indexes while staying within the given storage budget [22].

Machine learning has emerged as the predominant approach for query optimization in recent years. It also represents a future direction for our work, as we plan to explore the potential of applying machine learning techniques to predict SQL queries and subsequently execute these queries to gather predictive profile data for conducting PGO in relational databases. Ding et al. proposed a novel approach to optimizing index recommendation using a machine learning model for cost comparison of two execution plans. To vectorize the plans, they utilized a few feature channels that measured the amount of work and encoded the structure. The machine learning model was trained to take the difference between two plans as input and output, which plan had a lower cost. This approach allowed for more efficient index recommendation by reducing the need for costly evaluation of all possible indexes. The effectiveness of the method was demonstrated through experiments conducted on real-world datasets [23].

## REFERENCES

[1] N. N. Annisa, D. I. Sensuse, and H. Noprisson, "A systematic literature review of enterprise information systems implementation," in *2017 International Conference on Information Technology Systems and Innovation (ICITSI).* IEEE, 2017, pp. 291–296.

[2] Q. Ren, K. H. Kang, and I. T. Paulsen, "Transportdb: a relational database of cellular membrane transport systems," *Nucleic acids research*, vol. 32, no. suppl_1, pp. D284–D288, 2004.

[3] D. Chen, T. Moseley, and D. X. Li, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2016, pp. 12–23.

[4] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 233–244.

[5] B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "Plto: A link-time optimizer for the intel ia-32 architecture," in *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[6] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: a practical binary optimizer for data centers and beyond," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 2–14.

[7] R. Lavaee, J. Criswell, and C. Ding, "Codestitcher: inter-procedural basic block layout optimization," in *Proceedings of the 28th International Conference on Compiler Construction*, 2019, pp. 65–75.

[8] M. Annavaram, J. M. Patel, and E. S. Davidson, "Call graph prefetching for database applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 4, pp. 412–444, 2003.

[9] "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/.

[10] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 16–27.

[11] W. Chen and Y.-C. Chung, "Profile-guided optimization for function reordering: A reinforcement learning approach," in *2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2022, pp. 2326–2333.

[12] "MySQL," https://www.mysql.com/.

[13] L. Project, "The llvm compiler infrastructure," http://llvm.org/.

[14] S. Bag, S. K. Kumar, and M. K. Tiwari, "An efficient recommendation generation using relevant jaccard similarity," *Information Sciences*, vol. 483, pp. 53–64, 2019.

[15] A. Kopytov, "Sysbench: Cross-platform benchmarking tool," https://github.com/akopytov/sysbench.

[16] "GCC, the GNU Compiler Collection." [Online]. Available: https://gcc.gnu.org/

[17] K. Kukich, "Techniques for automatically correcting words in text," *Acm Computing Surveys (CSUR)*, vol. 24, no. 4, pp. 377–439, 1992.

[18] D. Narayanan, E. Thereska, and A. Ailamaki, "Continuous resource monitoring for self-predicting dbms," in *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2005, pp. 239–248.

[19] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood, "Automatic performance diagnosis and tuning in oracle." in *CIDR*, 2005, pp. 84–94.

[20] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 338–350.

[21] S. Choenni, H. Blanken, and T. Chang, "Index selection in relational databases," in *Proceedings of ICCI'93: 5th International Conference on Computing and Information*. IEEE, 1993, pp. 491–496.

[22] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley, "Db2 advisor: An optimizer smart enough to recommend its own indexes," in *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 2000, pp. 101–110.

[23] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya, "Ai meets ai: Leveraging query executions to improve index recommendations," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1241–1258.