

Adapting Performance Analytic Techniques in a Real-World Database-Centric System: An Industrial Experience Report

Lizhi Liao
University of Waterloo, Canada
lizhi.liao@uwaterloo.ca

Heng Li
Polytechnique Montréal, Canada
heng.li@polymtl.ca

Weiye Shang
University of Waterloo, Canada
wshang@uwaterloo.ca

Catalin Sporea
ERA Environmental, Canada

Andrei Toma
ERA Environmental, Canada

Sarah Sajedi
ERA Environmental, Canada

ABSTRACT

Database-centric architectures have been widely adopted in large-scale software systems in various domains to deal with the ever-increasing amount and complexity of data. Prior studies have proposed a wide range of performance analytic techniques aimed at assisting developers in pinpointing software performance inefficiencies and diagnosing performance issues. However, directly applying these existing techniques to large-scale database-centric systems can be challenging and may not perform well due to the unique nature of such systems. In particular, compared to typical database-based systems like online shopping systems, in database-centric systems, a majority of the business logic and calculations reside in the database instead of the application. As the calculations in the database typically use domain-specific languages such as SQL, the performance issues of such systems and their diagnosis may be significantly different from the systems dominated by traditional programming languages such as Java. In this paper, we share our experience of adapting performance analytic techniques in a large-scale database-centric system from our industrial collaborator. Our adapted performance analysis pays special attention to the database and the interactions between the database and the application with minimal reliance on expert knowledge and manual effort. Moreover, we document our encountered challenges and how they are addressed during the development and adoption of our solution in the industrial setting as well as the corresponding lessons learned. We also discuss the real-world performance issues detected by applying our analysis to the target database-centric system. We anticipate that our solution and the reported experience can be helpful for practitioners and researchers who would like to ensure and improve the performance of database-centric systems.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **General and reference** → *Performance*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12...\$15.00
<https://doi.org/10.1145/3611643.3613893>

KEYWORDS

Performance analysis, database-centric system, field testing, performance issue, performance regression, root cause analysis

ACM Reference Format:

Lizhi Liao, Heng Li, Weiye Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2023. Adapting Performance Analytic Techniques in a Real-World Database-Centric System: An Industrial Experience Report. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*, December 3–9, 2023, San Francisco, CA, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3611643.3613893>

1 INTRODUCTION

Performance is critical to the success of a modern software system since it directly impacts the system's speed, responsiveness, and reliability, which are crucial to user satisfaction and productivity [9]. Unsatisfactory performance of the system may potentially lead to significant adverse consequences to the stakeholders of the system, e.g., additional operation costs, damaged customer relations, or loss of profit [47, 50, 54]. This is also the case for our industrial collaborator. In particular, the system from our industrial collaborator is an enterprise application (EA) that provides important government regulation-related reporting services to clients across the globe, and due to the importance of the services, the system performance is considered a vital matter. EA is described in detail in Section 2.

Due to the practical advantages in the ease of development, scalability, maintainability, and reliability [26], our target industrial system (i.e., EA) adopts a database-centric architecture based on two tiers (i.e., application and database) to cope with the continuously increasing variety, quantity, and complexity of user data. Unlike typical database-based systems, like online shopping systems, where only simple CRUD (create, read, update, and delete) operations are applied to the database, EA extensively extends the database with pre-defined stored procedures to perform in-database operations to process business logic and calculations. By placing the database at the central position, the application written in regular source code (e.g., C#) is designed around the database and only serves as a connector between user requests and complex business logic and calculations in database stored procedures.

In recent years, many studies have proposed analytics-driven approaches to assisting in analyzing software system performance, such as monitoring and managing tremendous performance data [16, 22, 38, 51], identifying software performance issues [18, 31, 32, 35, 46, 55], and locating the root causes of performance issues [6, 12, 23, 29, 41]. Inspired by prior studies, we have deployed traditional

performance monitoring and analytics in *EA* to assist developers in ensuring the system performance. However, since the existing analytic techniques are not originally designed for database-centric systems, we face major challenges in the target industrial context. In particular, the results of traditional performance analytics consider only the application part of the system, while in our target system, the application source code is simply responsible for connecting user requests to the complex business logic and calculations residing in the database. Focusing solely on the application and ignoring the crucial impact of the database and the interactions between the database and the application often lead to failures in identifying real performance issues and their underlying root causes.

In this paper, we present our industrial experience of adapting performance analytic techniques in the large-scale database-centric system (i.e., *EA*) from our industrial collaborator. During the development and adoption of our solution, we encountered many engineering challenges. In particular, we first need to expose the runtime measurement of the database for performance analytics. We then need to expose the source code from the database for performance root cause analysis. For these challenges, we document our solution to address them and what we have learned during the process. We also share the implemented analysis in the target industrial setting and discuss the real-world cases detected by our analysis. We believe that our experience and learned lessons can provide software practitioners and researchers with helpful insights into ensuring and improving the performance of database-centric systems.

The main contributions of this study include:

- We provide an industrial experience report that discusses the encountered challenges, proposed solutions, and the learned lessons when adapting performance analytic techniques in a real-world database-centric system.
- We present the implemented analysis that can effectively assist developers in ensuring and improving the database-centric system performance from multiple aspects, including identifying performance anomalies within one release, detecting performance regressions between two releases, and locating the corresponding root causes.
- Our engineering process and experience can be helpful for software practitioners and researchers who are interested in ensuring and improving the performance of database-centric software systems.

Paper organization. Section 2 introduces our target system and explains the motivational background. Section 3 discusses the encountered challenges, our proposed solutions, and the learned lessons. Section 4 describes our implementation of the adapted performance analytic techniques. Section 5 discusses the detected real-world performance issues and improvements. Section 6 surveys prior research related to this paper. Section 7 discusses the threats to the validity of our study. Finally, Section 8 concludes the paper.

2 BACKGROUND

In this section, we briefly discuss the industrial system that we study and the backstory for adapting performance analytic techniques for the target system.

The industrial system under study. The industrial system (i.e., *EA*) studied in this work is a large-scale database-centric enterprise application from our industrial collaborator. *EA* is a commercial software system that provides reporting services related to government regulations (e.g., environmental regulations) to enterprises across the globe and it is also the market leader in its field. Due to the non-disclosure agreement (NDA), we cannot give the exact details about the system. *EA* has more than 20 years of history and is currently under active development for new features and maintenance activities. *EA* has more than two million lines of code based on the Microsoft .Net framework and adopts a database-centric architecture. The database plays a more vital role in the system and is extensively used by other software components (e.g., the web application) designed around it. *EA* utilizes the Microsoft SQL server as the database server to store all the user data and extensively extends the database with pre-defined stored procedures to perform in-database operations to process the business logic and calculations. The database is large in scale and high in complexity, consisting of more than 1,000 stored procedures and 5,000 tables.

The current practice in *EA* and its limitations. Due to the large client base and the importance of the service, the performance of the *EA* is a primary concern of our industrial collaborator. Like many other industrial systems, *EA* follows fast-paced agile software development and release practices. Therefore, instead of conducting resource- and time-consuming in-house performance testing before each new release, the existing performance assurance activities are conducted based on the system runtime data (e.g., web access logs or CPU usage) that is directly collected under real-world client operations in a canary releasing [1] manner. In particular, the new version of the system runs simultaneously with the old version, while the new version only serves a small portion of clients. After making sure there are no performance issues in the new version, it is then expanded to reach more clients until its full release.

The existing practice of performance analysis consists of the following two main parts. First, we put all the collected system performance metrics (e.g., CPU or memory usage) under the management of the ELK stack for data storing, processing, and visualization. Through the visualizations of the collected performance metrics, developers can have an overall comprehension of the system performance and then utilize the expertise of the system to further diagnose the performance issues. Furthermore, we also extract the response time for each type of web request from web application execution logs and adopt a pair-wise analysis to compare the response time of each web request type across versions of the system. We utilize statistical hypothesis testing (i.e., Wilcoxon rank-sum test (WRS) [53]) for the comparison and the null hypothesis assumes that the response times of a web request type are similar between the two versions while the alternate hypothesis is that there is a statistical difference (can be regression or improvement). The null hypothesis is rejected when p-values are smaller than 0.05. If there are web requests detected slow in the new release, we further adopt static code analysis techniques to identify the associated code changes in the commit history. Such slow web requests and code changes are provided to senior developers for further investigation.

However, there exist major limitations in the current practice of performance analysis. One limitation is that our existing practices focus solely on the application and do not have enough information

on the crucial impact of the database (e.g., database performance or runtime behaviors) and the interactions between the database and the application (e.g., how application code relates to database operations). Such a limitation often leads to failures in identifying the performance issues and their underlying root causes. Additionally, in our experience, we have observed that the pair-wise analysis often reports a large number of web requests as performance issues even though only a few or none of them were confirmed by senior developers (i.e., false positives), which leads to developers wasting a significant amount of time and effort in examining the results. Furthermore, unlike in-house performance testing where the workload is often fixed or constant, the real workload in the production environment is constantly changing, which makes it challenging for developers to determine whether the performance impact is caused by performance issues or workload variations solely by visualizing the collected performance metrics.

Current practices of addressing these challenges largely rely on developers' manual analysis in an ad-hoc manner, however, due to the increasing complexity and scale of the system, such a manual process can be extremely difficult and tedious, and even senior developers with expert knowledge may still need to spend a significant amount of manual effort and extensive time to diagnose the performance issues and fix them. Worse, the delay in the detection and fixing of performance issues may potentially cause a significant adverse impact in terms of both the reputational and financial aspects of the company.

Our proposed solution. In order to overcome these limitations, we propose to adapt performance analytic techniques in the context of the target database-centric system (i.e., *EA*). In particular, our adapted solution pays special consideration to the performance of the database and the interactions between the database and the application. Further, we aim to provide multi-aspect analysis, including detecting performance anomalies within one release, identifying performance regressions between two releases, and locating the root causes. Our approach has been adopted in the internal platform of the industrial collaborator to assist developers in ensuring and improving the system performance on a daily basis.

3 CHALLENGES AND LESSONS LEARNED

In this section, we discuss the challenges encountered during the process of adapting performance analytic techniques to our target real-world system (i.e., *EA*) that is database-centric and large in scale. For each challenge, we describe the corresponding solutions and the lessons that we learned from addressing the challenge.

Figure 1 outlines our engineering process. Overall, we first expose the runtime measurement of the database for performance analytics, then we expose the source code from the database for performance root-cause analysis. The details of our implemented analysis are described in Section 4 and the detected real-world performance issues are discussed in Section 5.

3.1 C1: Exposing the runtime measurement of the database for performance analytics

Challenge. Analyzing system performance is a critical task for a large-scale software system since it helps to guarantee and improve the system performance. In order to adapt the existing performance

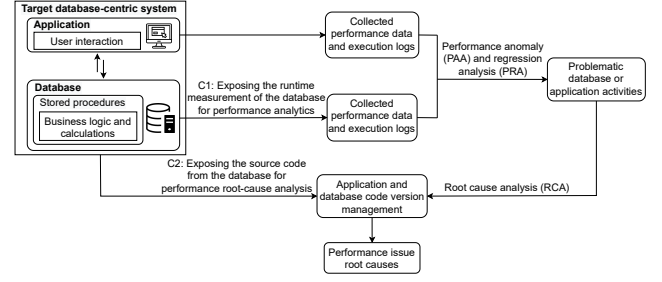


Figure 1: An overall engineering process of adapting performance analytic techniques in *EA*

analytic techniques in *EA*, we first need to understand how the system behaves under real-world workloads, especially when the system misbehaves or exceeds our expectations. However, unlike traditional applications, whose system performance or runtime behaviors are well captured by many available techniques, for example, application performance monitoring (APM) tools monitor and manage the performance of the application, and web servers (e.g., Apache Tomcat [4]) typically generate web access logs by default in order to support performance analysis, the performance and runtime behaviors of the database are often unexposed or not well understood. In database-centric systems without such essential information, when severe performance issues arise, what developers can only do is make decisions based on uninformed guesses or personal experiences. On the other hand, database-centric systems tend to have frequent database access operations that need to be captured. We could not add too much monitoring overhead to the system since the performance of the database is critical. Therefore, our first challenge is to comprehensively understand how the database in *EA* behaves under dynamic field workloads for performance analytics, without introducing significant monitoring overhead to the system.

Solution. To address the challenge, we propose to leverage two types of monitoring solutions to gain insight into how the database in *EA* performs under the dynamic field workloads and its corresponding runtime behavior from different perspectives and granularities. Resource utilization (e.g., CPU or memory usage) is an important aspect to consider when understanding the performance of a system, especially when some resources are overused or their utilization becomes unreasonable. Therefore, we first add the recording of system-level performance metrics for all the database servers (based on Windows platforms) by utilizing TypePerf [5]. TypePerf is a lightweight tool in Microsoft Windows operating systems used for performance monitoring. It provides a wide range of performance counters to collect the system performance, such as CPU usage, memory utilization, and disk activity, and it can output the data in various formats, like CSV or binary log files. These collected performance counters provide us with the window to observe how busy various resources of the system are when running under a specific workload.

Furthermore, to support a more comprehensive performance analysis of *EA*, we also leverage sp_WhoIsActive [2] to record a more fine-grained (i.e., at the query level) performance and runtime behavior of the database. sp_WhoIsActive is a powerful and lightweight stored procedure that can help gain important insights into

the internal workings of the SQL Server database. `sp_WhoIsActive` provides real-time information about running processes, such as query text, execution time, I/O activity, session information, and the outer batch or stored procedure information. By combining the abundant information from these sources, developers can obtain a clear and comprehensive understanding of the database performance in *EA*. This provides developers with a solid foundation to analyze which component may be malfunctioning and the corresponding contextual data, such as what else is running on the system, which helps identify and troubleshoot performance issues.

To avoid significant performance overhead to the system while keeping an adequate amount of the system runtime information, we have attempted to performance test the system with various data collecting intervals (e.g., 10 to 60 seconds) under different workloads. Through working closely with the senior developers, we determined that recording system-level performance metrics every 30 seconds and query-level database runtime behaviors every 15 seconds achieves an acceptable balance between the accuracy and the overhead of the measurement. It is worth noting that we also put these massive and valuable data that record system performance and runtime behaviors in our ELK Stack-based performance data platform (cf. Section 2) for the unified management and preliminary visualizations. Our solution enables further performance anomaly analysis (PAA) and performance regression analysis (PRA) described in Section 4.

Lessons learned. Leveraging monitoring techniques to understand how the database in the database-centric system behaves under dynamic field workloads with minimal performance overhead. Due to the high complexity and large scale of modern software systems, if there is a lack of adequate information about the system performance and runtime behavior, it can be extremely challenging to analyze and diagnose the system performance when the system misbehaves. This is even harder for large-scale database-centric systems since the database plays a vital role in the performance of such systems, and the performance and runtime behaviors of the database are usually unexposed or not well understood. Therefore, it is crucial to have a comprehensive monitoring solution from various perspectives to record the database performance and runtime behavior to gain more insights into how a database-centric system behaves under dynamic field workloads.

On the other hand, collecting more system runtime information (e.g., in a high recording frequency) often means more usable data and a more accurate representation of the system performance. However, it also brings more performance overhead, which is a major concern of developers, especially in the production environment. Therefore, it is critical to work closely with senior developers to run various experimental trials with different recording frequencies at various workloads to understand the introduced performance impact. While maintaining enough data and accuracy for the analysis, it is important to minimize additional performance costs to increase developers' confidence in adopting our solution in practice.

Providing centralized management and visualization support to developers. Based on our experience, we observe that large-scale database-centric systems tend to have a large number of performance metrics and runtime behavior logs since the system often involves frequent database access operations that need

to be monitored and recorded. Such data are valuable and contain rich information about the system's workload and running status for further performance analysis of the system. Therefore, putting these runtime monitoring data under the management of centralized processing and storage mechanisms (e.g., in the ELK stack-based platform) is important. In addition, putting these system runtime data into visualization also helps developers obtain a preliminary grasp and analysis of the performance of the large-scale database-centric system in a more intuitive manner.

We monitor both the system-level performance and query-level runtime behavior to comprehensively understand how the database in a database-centric system performs under dynamic field workloads. Minimizing monitoring overhead and providing centralized management and visualization of the collected data is helpful to increase developers' confidence in adopting our solution.

3.2 C2: Exposing the source code from the database for performance root-cause analysis

Challenge. In *EA*, the database is highly complex and large in scale, comprising over 5,000 tables and 1,000 stored procedures, with an average of over 300 lines of code (LOC) per stored procedure. Unlike application source code that is well managed by various modern version control systems, the code changes in the database are often treated as second-class citizens and are left out of the development process. This is because database code may be strongly integrated with the database (e.g., SQL code in stored procedures), which makes it more challenging and requires more specialized knowledge and skill sets to manage the changes. On the other hand, in a database-centric system, the application and the database are often tightly connected, for example, the SQL queries are often highly dynamically constructed through a series of string operations (e.g., concatenation and substitution) in the application code and the application code usually invokes stored procedures to perform in-database operations to process the business logic and calculations. Since the database plays a critical part in the performance of a large-scale database-centric system, if there is no proper management of database code changes and the link between the application code and the database code, when severe performance issues arise in the system, it becomes even more challenging to diagnose their root causes.

Solution. Exposing database code histories. In order to expose database code histories, we implemented a lightweight solution to automate the process of managing the database changes in version control. In particular, we first export a version of the database schema, including table structures, views, functions, triggers, and stored procedures as text files. After that, we link these files to a version control system (e.g., Git or TFVC). Whenever there are new database schema changes made in the new version of the system, it will trigger our automated tool to export the database schema again and developers can commit it to version control. By following such a process, it becomes straightforward for developers to track what the change to the database was and visualize the difference over time. More importantly, it also makes it possible to revert to

previous versions of the schema if needed and trace back to the source changes in the database when analyzing and diagnosing the performance issues in the system.

Linking database activity to application code. We developed an automated solution that helps developers link database activities to concrete application code. Our solution first extracts the accessed database tables in the application code that generates database queries and the accessed database tables in the database query. By matching the accessed database tables from the application code and database query, we can establish a coarse-grained mapping between the database activity and the application code. Based on this mapping, we then leverage static code and string analysis techniques to further capture a more fine-grained relationship between database activity and application code. In particular, starting from the locations in the application code that execute a database query, we extract a list of possible query strings by analyzing intra-procedural and inter-procedural control flow with the help of the abstract syntax tree (AST). We then calculate their similarities to the database query to link the database activity to the application code. We find that by conducting the analysis, the required time and effort for developers to link database activities to the application code can be significantly reduced.

Linking application activity to database code. In an effort to link application activity to database code, we first utilize static code analysis techniques to inspect the entire application source code and construct a call graph for the target application activities (e.g., web requests). Through traversing the methods in this call graph, we search for locations where the *SqlCommand* object is created with *CommandType.StoredProcedure* as its command type. We then extract the name of the stored procedure being executed in the method, which is passed as the command text parameter. By doing so, we can successfully link the application activity to a list of database stored procedures that the activity may invoke.

Our solution of managing database code changes and building the link between the application code and the database code enables further root cause analysis (RCA) presented in Section 4 when we identify any performance issues in the system.

Lessons learned. Managing the changes of the database is critical for the performance analysis of the database-centric system. Database version control brings many benefits in terms of improving management efficiency and performance in database-centric systems. In particular, by checking the database version control, it is convenient for developers to understand what changes have happened to the database, what has been done currently, and who is doing the changes. Having the database code changes under version control also help developers and database administrator to synchronize application and database better, which makes it easy to map the application changes to the database changes. Furthermore, in the event that any performance issues related to the database are identified, developers can promptly reference the database source changes and efficiently resolve them.

In large-scale database-centric systems, the performance issues are often due to the activities in the database rather than in the application. Analyzing and diagnosing the performance issues in database-centric systems are quite different from that in other systems. All too often, the developer cannot locate the root cause of performance issues in the application code, since

most of the business logic and calculations are processed by stored procedures that run in the database, rather than relying on the logic running in the application. With the help of static code analysis techniques, we can locate not only the application code but also the database code that is related to the problematic application activity, which effectively helps developers diagnose the real root cause of performance issues.

Leveraging static code and string analysis can help link the application code and dynamically generated SQL queries. Database activities, such as SQL queries, are often constructed by application code in a highly dynamic manner. Even the same code may generate significantly different SQL query commands depending on the corresponding parameter or conditions, which has remarkably increased the difficulty for developers to link a query to the corresponding application code. Through the utilization of the accessed table analysis combined with the static code and string analysis, we are able to assist developers in locating the application code that is associated with a problematic database activity.

Exposing database code history simplifies tracking database code modifications and enables tracing performance issues back to database code changes. By leveraging static code and string analysis, the database code and application code can be effectively linked to each other while requiring less expertise and effort from developers for performance analysis.

4 IMPLEMENTED ANALYSIS

In this section, we present the detailed implementation of our adapted performance analytic techniques in the context of the target large-scale database-centric system (i.e., *EA*). In total, our implemented performance analysis consists of three main parts: 1) performance anomaly analysis within one release, 2) performance regression analysis between two releases, and 3) root cause analysis. Part 1) and Part 2) of the analysis are supported by our solution to challenge C1 described in the previous section, while the analysis in Part 3) is enabled by our solution to challenge C2.

4.1 Performance anomaly analysis within one release (PAA)

In the first type of implemented analysis, our objective is to automatically identify the existence of performance issues that arise during a particular release. We identify the problematic application activity (i.e., web request) and database activity (i.e., query or stored procedure) through outlier analysis (PAA-1), and for the problematic database stored procedures, we also locate the corresponding performance bottleneck through proportion analysis (PAA-2).

PAA-1: Performance outlier analysis. Performance outliers refer to the performance data that deviates significantly from the expected or normal behavior of the system such as an extremely slow response of a SQL query or web request. It is a common yet important performance issue in large-scale database-centric systems dealing with massive amounts of data. To identify performance outliers, we first parse the recorded database and application runtime activity logs to extract the execution duration of each database activity (e.g., SQL query or stored procedures) and each application activity (e.g., web request) within the current version of the system.

Then, we utilize two statistical techniques that have been commonly utilized in prior work [7, 45, 52], namely Z-score and inter-quartile range (IQR), to effectively detect performance outliers in *EA*. In particular, the Z-score measures how far an observation deviates from the mean in the unit of standard deviation. If the Z-score of an observation is larger than three, it is an indicator that this observation is quite different from other ones and can be classified as an outlier [7]. By contrast, IQR is a measure of the dispersion of data, calculated by the difference between the third quartile and the first quartile. Typically, observations that fall below the value of the first quartile minus 1.5 times the IQR or exceed the value of the third quartile plus 1.5 times the IQR are identified as performance outliers. In the event of any performance outliers detected by both techniques, we leverage ElastAlert to provide a timely alert to the developers with emails. When an outlier is only detected by one technique, we do not produce an alert to avoid false alarms. Furthermore, we store and visualize the extracted execution duration and analysis results in our ELK stack, enabling developers to utilize our analysis in a more transparent and intuitive manner. Without this analysis, developers will not know in certain cases, a database query can take significantly longer than usual.

PAA-2: Performance proportion analysis. Database activities typically do not occur in isolation, but rather in the form of a sequence, for example, in the execution of database stored procedures, there may be a series of queries that are executed in a specific order. Identifying performance bottlenecks in the sequence of these activities is critical to help developers pinpoint the real cause of the identified performance issues. To this end, after we have identified the problematic database stored procedures from PAA-1, we further analyze the recorded database execution logs and recover the activities sequence by linking the related log lines into sequences. For instance, we group the log lines that have the same session ID and user ID as the problematic stored procedures, and these log lines can be used to represent the activity sequence of execution of the stored procedure. After that, we calculate the proportion of execution time of each individual activity (i.e., a query) compared to the performance of the entire sequence of activities (i.e., the stored procedure). Our intuition is that if a database stored procedures have performance issues, the activity that takes up the majority portion (e.g., more than two-thirds) of the execution time of that entire activity sequence is more likely to be the real cause of the problem. Without this analysis, developers would not identify the time-costly database query. We further prioritize the potentially problematic activities based on their contribution to overall performance issues. This allows developers to allocate their time and effort more effectively when conducting further investigations.

4.2 Performance regression analysis between two releases (PRA)

Different from the first type of analysis, in this subsection, we employ two complementary approaches, including statistical process control chart-based analysis and machine learning model-based analysis to point out any performance regressions introduced in the development of a new release compared to the previous release.

PRA-1: Statistical process control-based performance regression analysis. In an effort to identify the problematic system

activities that may cause the performance regression, we leverage a statistical process control technique called control charts (similar to prior research [35, 36]) to analyze the execution time of every type of system activity (including application activities and database activities) across the old and the new releases of the system. In particular, we extract the execution time for each system activity from the database and web application execution logs collected during the execution of both the old and new releases. The extracted performance from the old release serves as a baseline and is utilized to create control limits, including the center line (CL), upper control limit (UCL), and lower control limit (LCL). The LCL, CL, and UCL are commonly defined as the 10th, 50th, and 90th percentiles of the old release performance, respectively [35]. We create a separate control chart for each type of activity and once the control limits are established, we then compute the percentage of performance extracted from the new release that falls outside these limits, which is known as the violation ratio. This metric represents the likelihood of performance regression for a particular system activity type in the new release. If the violation ratio of an activity type surpasses a threshold set by experts (e.g., 20%) and its average execution time in the new version is higher than in the old version, then that system activity type is identified as a performance regression. Before our adaption, our analysis did not know the performance regressions occurring in the database activity.

PRA-2: Machine learning model-based performance regression analysis. To alleviate the impact of continuously varying workloads in the production environment, in this analysis, we rely on a machine learning model-based approach from prior work [27] to identify problematic system activities. In particular, when the system has a new release, we construct two random forest models to capture the relationship between the performance of the system and the runtime activities (i.e., system workload) by using the data from the old and new releases, respectively. We then apply these models to the new release data to evaluate their respective modeling errors. Afterward, we compare these two models by evaluating the deviation of their modeling errors through statistical analysis (i.e., performing statistical tests and measuring effect sizes). The idea behind this is that if a system runtime activity contributes to the deviation of the performance models built on two different software releases, then the system activity has a high chance of being the cause of performance regressions between the two releases. Finally, we utilize a linear regression model [19] to explain the relationship between the system activities and the deviation in modeling errors. If a system activity contributes significantly (with a p-value smaller than 0.05) to the deviation, then it is considered problematic. Prior to our adaption, our analysis was unaware of performance regressions related to the problematic database activity.

4.3 Root cause analysis (RCA)

Once performance issues have been identified through the above-mentioned analysis, we intend to locate their root causes from two perspectives, including problematic application activity and problematic database activity.

RCA-1: Root cause identification from problematic application activity. To locate the root causes (i.e., source code or code changes) of the identified problematic application activity (i.e., web

request), we first search through the entire application source code by keywords, function names, or URLs to find out one or more methods associated with the target application activity. Then, we utilize the static code analysis framework (i.e., .NET Compiler Platform SDK, also known as Roslyn) to parse the source code and construct a call graph for the method(s) associated with the web request. Moreover, for the methods in the call graph that invoke database stored procedures, we further extract the name of the stored procedures being executed. To achieve this, we traverse all the invoked stored procedures using static analysis and save their names in a list. Afterward, we enhance the previously constructed call graph by annotating the extracted stored procedure names onto the methods that invoke them. The source code, including both application code and database code (i.e., stored procedures), in these call graphs is identified as the potential root cause of the problematic application activity. In particular, when performance issues are detected between two releases of the system, we conduct a more in-depth investigation by analyzing the system version history. Specifically, we look for any changes between the two releases made to the application or database code that may have affected the methods or stored procedures presented in the enhanced call graph. We then consider these code changes, along with additional information such as code churn and committer data, as potential causes of the performance issues. Finally, we provide this information to the developers responsible for the affected code for further investigation and resolution.

RCA-2: Root cause identification from problematic database activity. The identified problematic database activities can be generally divided into two categories: problematic queries and problematic stored procedures. In cases where problematic stored procedures are identified from our performance anomaly analysis (PAA) or performance regression analysis (PRA), we can directly examine the corresponding database code and its version history, and consider this code or any associated code changes as the root cause of the issue (i.e., similar to RCA-1). However, for problematic queries, pinpointing the corresponding root cause becomes more challenging since these queries are often dynamically constructed through a series of string operations in the application code such as concatenation and substitution. Due to the complexity of the system, the same source code may generate remarkably different queries based on the different parameters or code paths. From a straightforward example shown in Listing 1, this application code constructs queries dynamically based on the user input, and the query may vary for different inputs.

```

1 public static string GenerateQuery(string userInput)
2 {
3     string sqlQuery = "SELECT * FROM Products";
4     if (!string.IsNullOrEmpty(userInput))
5     {
6         sqlQuery += " WHERE ProductName LIKE '%" +
7             userInput + "%'";
8     }
9     return sqlQuery;

```

Listing 1: Exmample of dynamically constructing a SQL query

To tackle this challenge, we first perform a coarse-grained matching between the identified problematic database activities and the

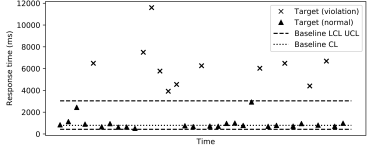
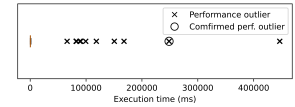
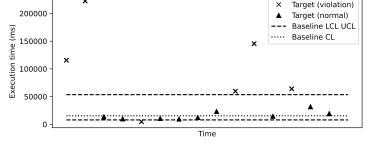
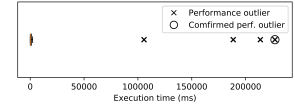
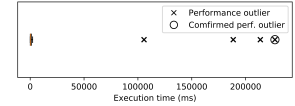
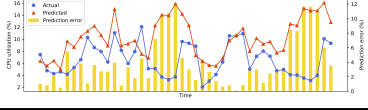
application code, based on the accessed database table. Firstly, we parse the application code to identify all the methods that involve database operations. Then, for each of these methods, we extract the database tables accessed by it (e.g., using regular expressions to extract the "FROM" clause). We also extract the accessed database tables of the SQL query under investigation. By matching these results, we can generate a list of methods in the application code that are potentially responsible for generating the target SQL query.

Once we have narrowed down the scope, we conduct a more fine-grained static code and string analysis (similar to prior research [33]) to further locate the root causes of the problematic database activities. In particular, for each of the candidate methods in the application code, we extract the abstract syntax tree (AST) from the corresponding source code and leverage the visitor pattern provided by the .NET compiler platform SDK (Roslyn) to identify the locations where SQL queries are sent to the database (e.g., *statement.executeQuery(query)*). Then we take each of these locations as the starting point and extract all the potential SQL string values passed to it. To achieve this, we extract the control flow that leads to the starting point with the help of Roslyn. Afterward, we trace back all the related expressions of the SQL query following the control flow previously extracted and try to determine the values of the query expression by resolving each variable involved in it. However, sometimes constructing a SQL query may involve variables passed as parameters, for example, "select" + *column* + "from table1", where *column* is passed to the method as a parameter. In order to determine the different possible values of the required parameters, we recursively resolve the query expression by following the call graph of the application in a backward direction. At the end of our analysis, a set of different possible SQL string values that can be produced at a specific location in the source code is generated. After that, we compare these SQL strings to the one that was identified as problematic. We leverage the Gestalt pattern matching algorithm [40] to calculate the similarities between the two SQL query strings and provide the ones with relatively high similarity (e.g., the top 5) as well as the corresponding locations (e.g., the names of the application methods) that could potentially generate these queries to developers for further investigation. In particular, if performance issues are detected between two releases of the system, we automatically scan the system version history to identify any changes made to these candidate methods and provide them to developers (i.e., similar to RCA-1).

5 DETECTED REAL-WORLD CASES

Our solutions and analysis have been adopted and used by the industrial collaborator to ensure the performance of *EA* on a daily basis. Since the performance analysis is based on the data collected in the production environment, the workload for *EA* is not predetermined (cf. Section 2). Table 1 summarizes a set of real-world issues that were detected by applying our adapted analysis in the target industrial setting. In particular, we have detected and confirmed a total of six cases, including five performance issues and one performance improvement that occurred either within one release or across two releases, and we also identified the corresponding root causes. In the following, we present more details about each detected case.

Table 1: The summary of real-world cases detected in the target industrial setting (i.e., EA)

ID	Type	Scope	Involved analysis	Description	Visualization of detection results																																
DI-1	Issue	BTR	PRA-1, RCA-1	This is a performance regression caused by newly added complex SQL queries in database stored procedures.																																	
DI-2	Issue	WOR	PAA-1, RCA-2	This is a performance anomaly due to an inefficient SQL query for a new feature.	 <table data-bbox="1094 596 1386 711"><tr><th colspan="4">Activity sequence: A1 → A2 → A3 → A4 → A5 → A6</th></tr><tr><th>Execution time (ms)</th><th>Proportion (%)</th><th colspan="2">Activity details</th></tr><tr><td>2113.56</td><td>39</td><td colspan="2">A6: detail info. of A6 (Confirmed)</td></tr><tr><td>1786.54</td><td>31</td><td colspan="2">A5: detail info. of A5</td></tr><tr><td>1302.84</td><td>23</td><td colspan="2">A3: detail info. of A3</td></tr><tr><td>161.95</td><td>3</td><td colspan="2">A4: detail info. of A4</td></tr><tr><td>146.52</td><td>3</td><td colspan="2">A1: detail info. of A1</td></tr><tr><td>39.67</td><td>1</td><td colspan="2">A2: detail info. of A2</td></tr></table>	Activity sequence: A1 → A2 → A3 → A4 → A5 → A6				Execution time (ms)	Proportion (%)	Activity details		2113.56	39	A6: detail info. of A6 (Confirmed)		1786.54	31	A5: detail info. of A5		1302.84	23	A3: detail info. of A3		161.95	3	A4: detail info. of A4		146.52	3	A1: detail info. of A1		39.67	1	A2: detail info. of A2	
Activity sequence: A1 → A2 → A3 → A4 → A5 → A6																																					
Execution time (ms)	Proportion (%)	Activity details																																			
2113.56	39	A6: detail info. of A6 (Confirmed)																																			
1786.54	31	A5: detail info. of A5																																			
1302.84	23	A3: detail info. of A3																																			
161.95	3	A4: detail info. of A4																																			
146.52	3	A1: detail info. of A1																																			
39.67	1	A2: detail info. of A2																																			
DI-3	Issue	WOR	PAA-1, PAA-2, RCA-2	This is a performance anomaly caused by a problematic SQL stored procedure related to business logic.																																	
DI-4	Issue	BTR	PRA-1, PRA-2, RCA-1	This is a performance regression caused by an inefficient SQL stored procedure when handling a large amount of data.	 <table data-bbox="1094 995 1386 1100"><tr><th>Execution time (ms)</th><th>Proportion (%)</th><th colspan="2">Activity details</th></tr><tr><td>2113.56</td><td>39</td><td colspan="2">A6: detail info. of A6 (Confirmed)</td></tr><tr><td>1786.54</td><td>31</td><td colspan="2">A5: detail info. of A5</td></tr><tr><td>1302.84</td><td>23</td><td colspan="2">A3: detail info. of A3</td></tr><tr><td>161.95</td><td>3</td><td colspan="2">A4: detail info. of A4</td></tr><tr><td>146.52</td><td>3</td><td colspan="2">A1: detail info. of A1</td></tr><tr><td>39.67</td><td>1</td><td colspan="2">A2: detail info. of A2</td></tr></table>	Execution time (ms)	Proportion (%)	Activity details		2113.56	39	A6: detail info. of A6 (Confirmed)		1786.54	31	A5: detail info. of A5		1302.84	23	A3: detail info. of A3		161.95	3	A4: detail info. of A4		146.52	3	A1: detail info. of A1		39.67	1	A2: detail info. of A2					
Execution time (ms)	Proportion (%)	Activity details																																			
2113.56	39	A6: detail info. of A6 (Confirmed)																																			
1786.54	31	A5: detail info. of A5																																			
1302.84	23	A3: detail info. of A3																																			
161.95	3	A4: detail info. of A4																																			
146.52	3	A1: detail info. of A1																																			
39.67	1	A2: detail info. of A2																																			
DI-5	Issue	BTR	PRA-2, PAA-1, RCA-1	This is a performance regression caused by an inefficient SQL query generated by the application code.																																	
DI-6	Improvement	BTR	PRA-2, RCA-1	This is a performance improvement due to the optimization of a SQL query in the new version.																																	

Note: In the "Scope" column, "WOR" stands for within one release and "BTR" stands for between two releases.

5.1 Detected Issue 1 (DI-1)

The first detected case (i.e., DI-1) is a performance regression between two consecutive releases of EA and it is caused by newly added complex SQL queries in the database stored procedures. In particular, the issue was detected by the control charts that track the response times of each web request type and examine the violation ratio of the same type of web request for a new release (i.e., PRA-1). From the results, we observed that there is one type of web request in the new release showing a significantly slower response time than that in the previous release with a violation ratio of 31.43% (the violation threshold is set at 20% by senior developers). The constructed control chart for the target web request is also shown in Table 1. Prior to exposing the source code from the database (as discussed in Section 3.2), we find that there seem to be no application code changes that are related to the performance regressions. After successfully addressing the challenge and performing the root cause analysis for problematic application activity (i.e., RCA-1), we managed to identify two suspicious updates of database stored procedures in the new release and then confirmed them with the developers from our collaborator. These updates introduced multiple complex SQL queries to support new requirements, resulting in the target web request that frequently invokes these stored procedures taking longer to respond to clients.

5.2 Detected Issue 2 (DI-2)

DI-2 is an identified performance anomaly due to an inefficient SQL query for a new feature. This issue was missed by our prior pair-wise analysis (cf. Section 2) and it caused a direct impact on end users. After investigation, we found that this issue is caused by a new feature (i.e., a new type of web request) released in the current version, however, our prior performance analytics focused only on the performance regressions of the same system feature during consecutive versions, which has limited support for this case. By utilizing our adapted analysis on the recorded database execution logs (i.e., PAA-1), we have successfully identified performance outliers caused by an inefficient SQL query. The detection results are also shown in Table 1. In particular, our root cause analysis (i.e., RCA-2) identified a specific section of the application source code that generates a complex query. Within this query, the subqueries implicitly create multiple temporary tables in the database. However, it was observed that these tables lack indexes, resulting in poor performance for both the query and the new web request that triggers this query. This adverse performance impact becomes more prominent when users interact with larger-than-usual amounts of data. After discussing the analysis results with developers, they confirmed the identified root cause and implemented optimizations to address the inefficiency. After applying the fix, we observed a

significant improvement in performance, with the execution time of the SQL query being approximately 10 times faster.

5.3 Detected Issue 3 (DI-3)

DI-3 is a performance anomaly caused by a problematic stored procedure related to business logic. Similar to the previous performance issue (i.e., DI-2), this case went undetected in the prior analysis, and the clients reported experiencing a noticeable slowdown when performing a specific operation. Our performance outlier analysis (i.e., PAA-1) has successfully identified that the issue stemmed from a problematic stored procedure associated with business logic. However, due to confidentiality concerns, we are unable to provide further details regarding its specifics. Our further performance proportion analysis (i.e., PAA-2) extracted the sequence of database queries related to the target stored procedure from execution logs and from the results (also shown in Table 1), we identified a particular query that consumed a significant portion of the overall processing time, approximately 39%. After presenting the corresponding root cause through our root cause analysis (i.e., RCA-2) to developers, they delightedly accepted our findings and promptly addressed the issue in the subsequent release.

5.4 Detected Issue 4 (DI-4)

DI-4 is a performance regression caused by an inefficient stored procedure when handling a larger amount of data than before. Our machine learning model-based performance regression analysis (i.e., PRA-2) has located a problematic web request. With the support of our solution of exposing the source code from the database, we successfully located the corresponding root cause through our root cause analysis (i.e., RCA-1). However, we also noticed that the target web request potentially calls a wide range of stored procedures (>10), making the manual inspection of these results time-consuming and requiring significant expertise. By combining the results from our statistical process control-based analysis (i.e., PRA-1) on the recorded database execution logs, we identified a stored procedure responsible for the performance regression. As shown in Table 1, the stored procedure in the new release shows a remarkably longer execution time than that in the previous release with a violation ratio as high as around 40%. Furthermore, we also found that this issue remained unnoticed in previous releases primarily due to the fact that clients are currently handling a larger volume of data than before.

5.5 Detected Issue 5 (DI-5)

DI-5 is a performance regression between two consecutive releases caused by a newly added inefficient SQL query in the application code. Our machine learning model-based performance regression analysis (i.e., PRA-2) has detected a problematic application activity, but from the results of the root cause analysis (i.e., RCA-1), we found that the identified problematic application activity seems to be responsible for a lot of work and is associated with many changed source code files (around 30 files), making it difficult to pinpoint the real root cause. By combining these findings with the results from our performance outlier analysis (i.e., PAA-1), we confirmed that one of the changed source code files added an inefficient SQL query that executes for a couple of minutes (the results are shown

in Table 1). We also provided the corresponding application code changes related to this query to developers for further optimization.

5.6 Detected Improvement 6 (DI-6)

The last detected case (i.e., DI-6) is a performance improvement due to the optimization of a SQL query in the new version. Our machine learning model-based performance regression analysis (i.e., PRA-2) successfully detected this case. The results shown in Table 1 reveal that the new version is more likely to perform better (i.e., consumes less CPU) than the old version when handling similar workloads since the system performance predicted by the model constructed from the old version (i.e., the red line) is often higher than the actual performance (i.e., the blue line). The results of our root cause analysis (i.e., RCA-1) indicate that this performance improvement was accomplished by several updates to an existing complex SQL query. After discussing the results with senior developers, we confirmed that the developers who made these updates opted to use a subquery with the EXISTS function in replace of the DISTINCT function in an SQL query to retrieve data from multiple tables. Such an update helps to avoid the additional performance overhead of scanning the entire table and sorting the retrieved rows before suppressing the duplicates. Furthermore, the updates also enable the query execution to stop early when specific conditions are met, thereby further improving performance, particularly when processing extensive amounts of user data. This detected case also indicates that our solution can derive good practices from detected performance improvement cases. We can utilize these cases as examples for developers to help them learn good coding practices and avoid sub-optimal practices in future development tasks.

6 RELATED WORK

In the following, we discuss prior work that is relevant to our study. The related work is categorized into two aspects: 1) diagnosing performance issues in large-scale systems; and 2) improving database-centric system quality.

Diagnosing performance issues in large-scale systems. Prior studies have proposed various approaches to assist developers and operators in diagnosing performance issues in large-scale software systems. These existing techniques collect and utilize different sources of data to analyze the performance of a software system. Analyzing and comparing the system performance metrics collected during runtime is the most straightforward approach to diagnosing system performance problems [17, 18, 31, 32, 34–37, 46, 55]. For instance, Mariani et al. [32] propose a lightweight approach that exploits the causality graph of multi-level performance metrics to pinpoint the performance anomalies in cloud systems. Nguyen et al. [37] combines machine learning techniques (e.g., Random-Forest [10] and SMO [39]) to mine the relationship between performance metrics and performance regression root causes from historical versions to predict the root causes in the new versions.

System execution logs are also an important source of data that are widely utilized to analyze system performance issues. Jiang et al. develop various algorithms to mine the execution logs of the runtime systems to uncover the functional [24] and performance [25] anomalies in load tests. Altman et al. propose WAIT [8] to diagnose the performance issues caused by idle time in applications from

execution logs. Lu et al. [28] develop an approach for performance abnormal analysis in Apache Spark system [3]. They extract the CPU-related features, execution path features, and garbage collection features from the raw execution logs, then build a statistical model based on these features to identify the performance anomaly and locate the corresponding root causes by using weighted factors.

Prior research also utilizes system source code to diagnose performance issues [6, 12, 23, 29, 41]. For example, Heger et al. [23] analyze both the source code of the system and the performance metrics collected during unit testing to identify the code commits that introduce the performance issues. Luo et al. [29] propose PerfImpact based on genetic algorithms to automatically provide developers with the system inputs and source code changes that potentially lead to performance regressions across different software versions.

The aforementioned studies focus mostly on traditional applications while lacking the consideration of the database. Distinct from them, we target a large-scale database-centric system from the industry (i.e., *EA*) with special attention to not only the runtime behavior and performance of the database but also its interactions with the surrounding applications.

Improving database-centric system quality. Due to the widespread use of databases in practice, a great number of studies have been conducted to improve database-centric system quality. Grechanik et al. [20, 21] develop approaches that utilize static and dynamic code analysis techniques to detect and prevent database deadlocks in database-centric systems. Rigger and Su utilize multiple analysis techniques, such as query partitioning [43], pivoted query synthesis [44], or non-optimizing reference engine construction [42] to detect database logic bugs that are critical in database-centric system quality but often go unnoticed by developers.

Prior studies also discuss and summarize many common performance issues (e.g., source code-level performance anti-patterns [48, 49] or ORM-level performance anti-patterns [56, 57]) in database-centric systems and suggest solutions to alleviate or eliminate the adverse performance impact. Based on these studies, extensive work has been proposed to detect performance anti-patterns, for example, Yan et al. [58] develop a RubyMine IDE plugin for Ruby on Rails systems by using static code analysis techniques to automatically detect ORM-level performance issues and provide the corresponding fixes. Chen et al. [11] provide an experience report on detecting and refactoring performance anti-patterns in an industrial system written in PHP with Laravel framework. They utilize both static and dynamic techniques in their approach. Lyu et al. [30] develop novel abstractions for the interactions between application and database, and detect 11 types of SQL anti-patterns on a set of 1,000 prevalent Android apps. Chen et al. [13–15] analyze various performance anti-patterns in the database-centric systems implemented in Java with Hibernate ORM framework.

The main focus of prior work is on optimizing the functional aspect or the performance of individual components of the database like optimizing SQL queries or database access code (e.g., ORM) under a fixed testing workload. Our work takes into consideration the performance of the entire running system (including both the database and the surrounding applications) and the continuously varying workload in the production scenario in order to effectively assist developers in diagnosing performance issues of a real-world database-centric system.

7 THREATS TO VALIDITY

This section discusses potential threats to the construct, external, and internal validity of our study.

Construct validity. In our study, we opt to leverage a built-in performance profiling tool in Windows (i.e., TypePerf) to monitor the system-level runtime performance and a community-maintained SQL Server stored procedure (i.e., `sp_whoisactive`) to capture the query-level database runtime behavior. Therefore, the quality of the recorded runtime performance metrics and behaviors may be a threat to the construct validity of our study. In addition, for the performance outlier analysis (i.e., PAA-1), we do not distinguish each type of database or application activity. Future work may separate each activity to examine the anomalies within each type of activity.

External validity. Our work is conducted on a large-scale enterprise system from our industrial collaborator, which is developed in the .NET framework with the Microsoft SQL server. Although our subject is a mature database-centric system with many years of development and maintenance history, which has a certain representativeness, it may still limit the generalization of our study results and findings to other systems.

Internal validity. Our solution depends on multiple statistical analysis techniques to detect performance anomalies or regressions within or across system releases. However, our approach may not perform well on small systems with an insufficient amount of data due to the nature of the statistical analysis. Besides, statistical analysis techniques often rely on the threshold to determine the statistical significance. However, in practice, determining the significance usually depends on the subjective judgment of the performance analysts.

8 CONCLUSION

In this paper, we report our experience in adapting performance analytic techniques to effectively assist developers in detecting performance issues and identifying the corresponding root causes for a large-scale database-centric system from our industrial collaborator. We share our challenges and solutions for exposing the runtime measurement of the database for performance analytics and exposing the source code from the database for performance root-cause analysis. We also share the implemented database-aware performance analysis, including detecting performance anomalies within one release, identifying performance regressions between two releases, and root cause analysis, as well as the discussions of the real-world performance issues detected by our analysis. We anticipate that our engineering solution and documented experience can provide helpful insights into ensuring and improving the performance of large-scale database-centric systems for both research and practice.

ACKNOWLEDGMENTS

We are grateful to ERA Environmental Management Solutions for providing access to the industrial system used in our study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of ERA Environmental Management Solutions and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of ERA Environmental Management Solutions' products.

REFERENCES

- [1] 2014. CanaryRelease. Retrieved Mar. 7, 2023 from <https://martinfowler.com/bliki/CanaryRelease.html>
- [2] 2017. sp_whoisactive SQL Server Monitoring Stored Procedure by Adam Machanic. Retrieved Jan. 23, 2023 from <http://whoisactive.com>
- [3] 2022. Apache Spark - Unified Engine for large-scale data analytics. Retrieved Mar. 15, 2023 from <https://spark.apache.org/>
- [4] 2022. Apache Tomcat® - Welcome! Retrieved Jan. 7, 2023 from <https://tomcat.apache.org/>
- [5] 2022. typeperf Microsoft. Retrieved Jan. 17, 2023 from <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/typeperf>
- [6] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. 2016. Learning from Source Code History to Identify Performance Failures. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016*. ACM, 37–48.
- [7] Hamzeh Alimohammadi and Shengnan Nancy Chen. 2022. Performance evaluation of outlier detection techniques in production timeseries: A systematic review and meta-analysis. *Expert Syst. Appl.* 191 (2022), 116371.
- [8] Erik R. Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. 2010. Performance analysis of idle programs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. ACM, 739–753.
- [9] André B Bondi. 2015. *Foundations of software and system performance engineering: process, performance modeling, requirements, testing, scalability, and practice*. Pearson Education.
- [10] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32.
- [11] Boyuan Chen, Zhen Ming Jiang, Paul Matos, and Michael Lacaria. 2019. An Industrial Experience Report on Performance-Aware Refactoring on a Database-Centric Web Application. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 653–664.
- [12] Jinfu Chen and Weiye Shang. 2017. An Exploratory Study of Performance Regression Introducing Code Changes. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 341–352.
- [13] Tse-Hsun Chen, Weiye Shang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2016. Detecting problems in the database access code of large scale systems: an industrial experience report. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. ACM, 71–80.
- [14] Tse-Hsun Chen, Weiye Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 1001–1012.
- [15] Tse-Hsun Chen, Weiye Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2016. Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks. *IEEE Trans. Software Eng.* 42, 12 (2016), 1148–1161.
- [16] Tse-Hsun Chen, Mark D. Syer, Weiye Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2017. Analytics-Driven Load Testing: An Industrial Experience Report on Load Testing of Large-Scale Systems. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 243–252.
- [17] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. 2010. Mining Performance Regression Testing Repositories for Automated Performance Analysis. In *Proceedings of the 10th International Conference on Quality Software, QSIQ 2010, Zhangjiajie, China, 14-15 July 2010*. IEEE Computer Society, 32–41.
- [18] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. 2015. An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*. IEEE Computer Society, 159–168.
- [19] David A Freedman. 2009. *Statistical models: theory and practice*. Cambridge university press.
- [20] Mark Grechanik, B. M. Mainul Hossain, and Ugo A. Buy. 2013. Testing Database-Centric Applications for Causes of Database Deadlocks. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 174–183.
- [21] Mark Grechanik, B. M. Mainul Hossain, Ugo A. Buy, and Haisheng Wang. 2013. Preventing database deadlocks in applications. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 356–366.
- [22] Wilhelm Hasselbring and André van Hoorn. 2020. Kieker: A monitoring framework for software engineering research. *Softw. Impacts* 5 (2020), 100019.
- [23] Christoph Heger, Jens Happe, and Roozbeh Farahbod. 2013. Automated root cause isolation of performance regressions during software development. In *ACM/SPEC International Conference on Performance Engineering, ICPE '13, Prague, Czech Republic - April 21 - 24, 2013*. ACM, 27–38.
- [24] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2008. Automatic identification of load testing problems. In *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*. IEEE Computer Society, 307–316.
- [25] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2009. Automated performance analysis of load tests. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*. IEEE Computer Society, 125–134.
- [26] Toon Koppelaars. 2004. A database-centric approach to j2ee application development. *Oracle Development Tools Users Group (ODTUG)* (2004).
- [27] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiye Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2021. Locating Performance Regression Root Causes in the Field Operations of Web-based Systems: An Experience Report. *IEEE Transactions on Software Engineering* (2021), 1–1.
- [28] Siyang Lu, Bingbing Rao, Xiang Wei, Byung-Chul Tak, Long Wang, and Liqiang Wang. 2017. Log-based Abnormal Task Detection and Root Cause Analysis for Spark. In *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*. IEEE, 389–396.
- [29] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2016. Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 25–36.
- [30] Yingjun Lyu, Sasha Volokh, William G. J. Halfond, and Omer Tripp. 2021. SAND: a static analysis approach for detecting SQL antipatterns. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*. ACM, 270–282.
- [31] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. 2013. Automatic detection of performance deviations in the load testing of large scale systems. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 1012–1021.
- [32] Leonardo Mariani, Cristina Monni, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. 2018. Localizing Faults in Cloud Systems. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 262–273.
- [33] Loup Meurice, Csaba Nagy, and Anthony Cleve. 2016. Static analysis of dynamic database usage in java systems. In *Advanced Information Systems Engineering: 28th International Conference, CAISE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings* 28. Springer, 491–506.
- [34] Thanh H. D. Nguyen. 2012. Using Control Charts for Detecting and Understanding Performance Regressions in Large Software. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*. IEEE Computer Society, 491–494.
- [35] Thanh H. D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2011. Automated Verification of Load Tests Using Control Charts. In *18th Asia Pacific Software Engineering Conference, APSEC 2011, Ho Chi Minh, Vietnam, December 5-8, 2011*. IEEE Computer Society, 282–289.
- [36] Thanh H. D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2012. Automated detection of performance regressions using statistical process control techniques. In *Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE '12, Boston, MA, USA - April 22 - 25, 2012*. ACM, 299–310.
- [37] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2014. An industrial case study of automatically identifying performance regression-causes. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. ACM, 232–241.
- [38] George Papadimitriou, Cong Wang, Karan Vahi, Rafael Ferreira da Silva, Anirban Mandal, Zhengchun Liu, Rajiv Mayani, Mats Rynge, Mariam Kiran, Vickie E. Lynch, Rajkumar Kettimuthu, Ewa Deelman, Jeffrey S. Vetter, and Ian T. Foster. 2021. End-to-end online performance data capture and analysis for scientific workflows. *Future Gener. Comput. Syst.* 117 (2021), 387–400.
- [39] John C. Platt. 1999. *Fast Training of Support Vector Machines Using Sequential Minimal Optimization*. MIT Press, Cambridge, MA, USA, 185–208.
- [40] John W Ratcliff, David Metzner, et al. 1988. Pattern matching: The gestalt approach. *Dr. Dobbs's Journal* 13, 7 (1988), 46.
- [41] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. 2019. PeASS: A Tool for Identifying Performance Changes at Code Level. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1146–1149.
- [42] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *ESEC/FSE '20: 28th*

- ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020. ACM, 1140–1152.
- [43] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 211:1–211:30.
 - [44] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 667–682.
 - [45] Peter J Rousseeuw and Mia Hubert. 2011. Robust statistics for outlier detection. *Wiley interdisciplinary reviews: Data mining and knowledge discovery* 1, 1 (2011), 73–79.
 - [46] Weiyi Shang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2015. Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*. ACM, 15–26.
 - [47] Connie U. Smith. 2007. Introduction to Software Performance Engineering: Origins and Outstanding Problems. In *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures (Lecture Notes in Computer Science, Vol. 4486)*. Springer, 395–428.
 - [48] Connie U. Smith and Lloyd G. Williams. 2002. New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In *28th International Computer Measurement Group Conference, December 8-13, 2002, Reno, Nevada, USA, Proceedings*. Computer Measurement Group, 667–674.
 - [49] Connie U. Smith and Lloyd G. Williams. 2003. More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *29th International Computer Measurement Group Conference, December 7-12, 2003, Dallas, Texas, USA, Proceedings*. Computer Measurement Group, 717–725.
 - [50] Connie U. Smith and Lloyd G. Williams. 2003. Software Performance Engineering. In *UML for Real - Design of Embedded Real-Time Systems*. Kluwer, 343–365.
 - [51] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: a framework for application performance monitoring and dynamic software analysis. In *Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12, Boston, MA, USA - April 22 - 25, 2012*. ACM, 247–248.
 - [52] HP Vinutha, B Poornima, and BM Sagar. 2018. Detection of outliers using interquartile range technique from intrusion dataset. In *Information and decision sciences*. Springer, 511–518.
 - [53] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.
 - [54] C. Murray Woodside, Greg Franks, and Dorina C. Petriu. 2007. The Future of Software Performance Engineering. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*. IEEE Computer Society, 171–187.
 - [55] PengCheng Xiong, Calton Pu, Xiaoyun Zhu, and Rean Griffith. 2013. vPerfGuard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *ACM/SPEC International Conference on Performance Engineering, ICPE'13, Prague, Czech Republic - April 21 - 24, 2013*. ACM, 271–282.
 - [56] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. ACM, 1299–1308.
 - [57] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How *not* to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 800–810.
 - [58] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. PowerStation: automatically detecting and fixing inefficiencies of database-backed web applications in IDE. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 884–887.

Received 2023-05-18; accepted 2023-07-31