Database Query Analysis and Optimization in a Large Scale Information System

Case Study on Large Scale DHIS2 Implementations

Mohamed Ameen

Thesis submitted for the degree of
Master in Informatics:Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2021

Database Query Analysis and Optimization in a Large Scale Information System

Case Study on Large Scale DHIS2 Implementations

Mohamed Ameen

Database Query Analysis and Optimization in a Large Scale Information System

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Information Systems play an integral role in several aspects of businesses and society. They provide valuable insights by collecting and analyzing data and help optimal decision makings. With the advancement in technologies, the scale of information systems also increases. Modern requirements of Information systems demand high scalability to support very large-scale needs. Large-scale essentially means the data volume is large and data access frequency is very high. Resolving bottlenecks and avoiding common pitfalls in Information systems is the key to achieving higher scalability.

DHIS2 is a web application originally designed for collecting, aggregating, and analyzing statistical health data. DHIS2 is used in more than 73 different countries, each with its implementation and use cases. Due to the covid pandemic, the demand for a scalable DHIS2 system increased and Covid contact tracing and Covid vaccination tracking. Even though DHIS2 is used mainly in the Health domain, there are also implementations of DHIS2 in other sectors like Education. Some of these implementations need to support a country-wide scale. Such large-scale DHIS2 implementations frequently suffer from performance issues and bottlenecks.

This thesis aims to study the types of performance issues faced by large-scale Information Systems. I focus on various large-scale DHIS2 implementations and investigate the bottlenecks both on the application side and database side of DHIS2. The thesis also aims at finding out optimization techniques and changes to improve performance and clear these bottlenecks. The results of this research are generalized in such a way that they can be applied to any Information system and not just DHIS2. The results show successful optimization changes and how much of an impact these changes have had on the performance of real-world large-scale DHIS2 implementations. Qualitative analysis of the performance improvement is done to understand the impact of each optimization.

Acknowledgements

First, I would like to give a big thanks to my supervisor Sundeep Sahay for his guidance throughout the research. I was very new to Academic writing. He guided me in the right direction and gave me valuable insights into how best to express my work.

I would also like to thank the DHIS2 Core Development team for their collaboration in this research. Thanks to Bob Joliffe for his valuable inputs to my research. His experience helped me learn a lot and contribute to assisting several country implementations. A big thanks to Gintare Vilkelyte and Stian Sandvold of the DHIS2 Core team. Collaborating with them was one of the best experiences throughout this research work. Special thanks to all the HISP Nodes and System administrators of the DHIS2 implementations part of my research. Pamod Amarakoon from HISP Sri Lanka and Barnabas from HISP Nigeria were very supportive and were happy to answer all of my queries. This research work would not have been possible without all of you.

Last but not least, I want to thank my wife, Shamna, and my daughter, Ayana, for encouraging and supporting me continuously throughout this thesis work. They have kept me motivated during my research work with their positive and kind words.

Contents

List of Tables

Listings

List of Figures

Acronyms

API Application Programming Interface.

APM Application Performance Management.

CPU Central Processing Unit.

CSS Cascading Style Sheets.

DHIS2 Digital Health Information Software 2.

GIN Generalized Inverted Index.

HIS Health Information Systems.

HISP Health Information System Program.

HMIS Health Management Information System.

HTML HyperText Markup Language.

IoC Inversion of Control.

IS Information Systems.

IT Information Technology.

JEE Java Enterprise Edition.

JRE Java Runtime Environment.

MOH Ministry of Health.

NGO Non-Governmental Organisation.

ORM Object-Relational Mapping.

QA Quality Assurance.

REST Representational State Transfer.

SQL Structured Query Language.

SSD Solid State Drive.

UiO University of Oslo.

WAR Web ARchive.

WHO World Health Organisation.

1 Introduction

Modern Information Systems (IS) are used in various fields and with varying scopes. It plays an integral role in business and society. It provides valuable insights by collecting and analyzing data and helps in optimal decision-making. Information systems occasionally suffer from performance bottlenecks due to high data volume or high data access rates. With the advancement in technologies, IS should be able to scale up to support large-scale concrete implementations. Resolving bottlenecks and avoiding common performance pitfalls in Information systems is the key to achieving higher scalability. One such large scale IS is the Digital Health Information Software 2 (DHIS2).

DHIS2 is a global platform developed by the research group Health Information System Program (HISP) under the Department of Informatics at the University of Oslo (UiO) for collecting and aggregating health statistics. Governments in over 73 countries have adopted DHIS2 [8]. These countries are primarily but not limited to developing countries in Africa and Asia. Several prominent NGO's, including the World Health Organisation (WHO), have adopted DHIS2 for their data collection, aggregation, and analysis needs.

Such global adoption of DHIS2 has led to the establishment of several concrete implementations on varying scales across the globe. Some implementations cater to a small region, like a district/state/province, within a country, whereas other implementations cater to nationwide scope supporting the whole population of that country. When the scale of a DHIS2 implementation increases, so do the performance issues faced by that instance.

1.1 Motivation

Scalability is a Non-functional requirement for any large-scale Information System. There is always room for optimizations that can improve the scalability of an enterprise application. The usefulness of any application quickly deters if it has severe bottlenecks and cannot sustain the real-world practical load. Some of the major bottlenecks are due to well-known anti-patterns and developer errors. One of my motivations was to help users of large-scale Information systems to achieve their desired productivity.

As part of my case study, I focus on large-scale DHIS2 implementations. DHIS2 is the world's most used health information management system that aids in the public health of numerous countries. It continues to grow, and more countries are adopting the platform. The continued adoption and growth of DHIS2 in various countries and the diverse implementations make scalability and availability high priority non-functional requirements for DHIS2.

Over the years, DHIS2 is also being used at a national scale by Ministry of Health (MOH) of the respective countries for collecting, aggregating, and

analyzing their country health data. Such large-scale implementation often faces performance issues that block them from using DHIS2 temporarily. These may either be due to server crashes, slow response times, or unavailability of any kind. Due to such performance issues when adopting on a large-scale, other countries or governments may get discouraged to use DHIS2 for their large-scale needs. In a time-critical implementation, there have been reports where users of DHIS2 have had to either switch to paper-based reporting or excel based recording and suffer a decrease in their productivity due to performance issues that affect DHIS2 usability. These bottlenecks have to be investigated and resolved to ensure DHIS2 continues to influence the different socio-economic efforts across the globe. I wanted to support the performance analysis investigations and bottleneck resolutions for large-scale implementations so that DHIS2 continues to be used effectively for even larger scale requirements and domains.

1.2 Research Question, Objective and Scope

The objective of this thesis is to analyze and optimize database queries in large-scale Information Systems. I focus on DHIS2 implementations, as it fits the criteria of a large-scale Information System interacting with a relational database. I analyze the different database queries involved in the Information system and attempt to resolve the performance bottlenecks using optimization techniques available in the existing literature. I also empirically evaluate how several optimization changes have had an impact on the performance of this large-scale DHIS2 implementations. The analysis is then generalized to make it applicable to any large-scale Information System.

In this thesis, I focus on the following three research questions:

1. What are the different performance issues faced by a large-scale Information System? How does it affect the user's day-to-day work?
2. What are some of the optimization techniques available to mitigate performance bottlenecks in a large-scale Information System like DHIS2?
3. What is the impact of the optimizations on DHIS2?

To address these research questions, I look at the DHIS2 version 2.34 and 2.35 along with PostgreSQL version 10.

The scope of this thesis is limited to the actual DHIS2 application source code and the PostgreSQL database. Other factors like Networking problems, Data storage or disk latencies, or other infrastructure-related bottlenecks or performance issues are outside the scope of this thesis.

1.3 Thesis Structure and Overview

This thesis contains the following chapters

Chapter 1: Introduction The current chapter briefs about my motivation and why I chose to focus on large-scale DHIS2 implementations. I also explain the research questions, objective, and scope of this thesis.

Chapter 2: Background In this chapter, I introduce the DHIS2 platform and briefly look at its history and how it has evolved into the go-to health information system. I touch upon the technology and functionality of DHIS2 along with some insights into the real-world concrete large-scale DHIS2 implementations that form the basis of my research.

Chapter 3: Research Methodology In this chapter, I present the research methodology that I used and the different data collection methods. I also explain the collaboration, ethical obligations, and challenges faced during the research process.

Chapter 4: Literature Review In this chapter, I present a review of existing literature that is related to my work. I conclude with a summary of my contribution to the literature.

Chapter 5: Performance Analysis In this chapter, I focus on performance analysis and identifying issues and bottlenecks faced by large-scale production DHIS2 instances and how it affects DHIS2 usability. I have characterized the performance issues into categories and subcategories to explain them in detail.

Chapter 6: Optimizations and Results In this chapter, I explain the different optimizations that resolved the performance issues enlisted in Chapter 4. Empirical evaluation of the optimizations is detailed.

Chapter 7: Discussion In this chapter, I discuss the identified bottlenecks, the optimizations applied, and the results of the optimizations. I also give a brief explanation of some of the limitations of some optimizations. I answer the research questions in this chapter.

Chapter 8: Conclusion In this chapter, I conclude the thesis with a summary of my work and how it answers the research questions. I also hint at some suggested future work.

2 Background

2.1 DHIS2 Platform

HISP originated in South Africa in 1996 as a project for improving health services for the post-apartheid period in South Africa [1]. Researchers from the University of Oslo were part of the HISP team. HISP saw the need for a unified health information system as a way to battle inequity in healthcare. These resulted in the origin of DHIS. They started developing a system for collecting and aggregating health data and introduced it in three health districts in Cape Town, South Africa, in 1998. DHIS continued to grow during the early 2000s and spread to multiple countries in Africa and Asia. The early DHIS system was used primarily for routine health reporting. The scale of the system was also quite small and was often limited to health facilities in certain districts alone. The frequency of reporting was also quite less, often once a month. The optimizations done for the early DHIS system were mainly around analytics processing. Transactional processing was not a feature in DHIS in the earlier system. Early DHIS served as an OLAP (Online Analytical Processing) system like a data warehouse instead of an OLTP (Online Transaction Processing) system.

2.1.1 Technology

DHIS2 is a flexible platform written primarily in Java. Any system where there exists a Java Runtime Environment (JRE) can run DHIS2 with a Java-enabled server or servlet container. A relational database accompanies the Java backend. PostgreSQL is the supported database[11].

The Java backend consists of a set of RESTful Web API to interact with various resources and perform functions within the DHIS2 system. The DHIS2 core also consists of core apps created with web technologies like Javascript, CSS and HTML5. Both core apps and the Java backend are bundled together in the form of a Web ARchive (WAR) format. The primary technologies and frameworks used in developing DHIS2 backend are Java Enterprise Edition (JEE) technology, Spring Inversion of Control (IoC) Framework and Hibernate Object-Relational Mapping (ORM) Framework[25].

The performance of the RESTful APIs affects the overall performance of a DHIS2 instance. The performance of individual database queries affects the performance of the corresponding RESTful APIs.

2.1.2 Functionality

DHIS2 is used to collect, validate, analyze, and present data. It is primarily used for aggregate and patient-based data for health information management purposes. Some of its key features are[27]:

- ”Provide data entry tools which can either be in the form of standard lists or tables or can be customized to replicate paper forms.”
- ”Supports data collection and analysis of transactional or disaggregated data.”
- ”Flexible and dynamic (on-the-fly) data analysis in the analytics modules (like GIS, Pivot Tables, Data Visualizer, Event reports).”
- ”Using the DHIS2 Web-API, allows for integration with external software and extension of the core platform through the use of custom apps.”
- ”Further modules can be developed and integrated as per user needs, either as part of the DHIS2 portal user interface or a more loosely-coupled external application interacting through the DHIS2 Web-API.”

The initial DHIS was designed for the specific situation in South Africa. HISP saw the need for modifications as the design did not sufficiently support the diverse needs of other nations. Modularity and flexibility became essential design goals for the next iteration. HISP wanted the system to be easily tailored and configured to suit any administration. In 2004 they started the development of DHIS2 as a modular web application. It was released in 2006 and has been in continuous development from then until today[2]. Over the years, countries and other implementations saw the need for online transaction processing features in DHIS2. These included the capability to track a specific entity over a period and to be able to capture and fetch data associated with the tracked entity. The early DHIS was never optimized for high transaction rates until OLTP requirements were needed.

Currently, DHIS2 has two main components. The Aggregate component has analytical processing capabilities. The Tracker component has online transactional processing capabilities. The Aggregate component is similar to any traditional Health Management Information System (HMIS), where data is reported in aggregate format. For example, the total number of cases of a specific disease in a given district for one month was captured on paper forms and submitted to a central office for manual entry into a database. This kind of data collection was necessary when computers and the internet were a rarity. But the time delay in reporting made it difficult to take prompt action for addressing any issues that the data revealed. The aggregated nature of the data also made it impossible to isolate and follow-up with an individual patient or case. Here is where the Tracker component provides a solution.

DHIS2 Tracker is a component that expands the DHIS2 data model from aggregate to individual-level data that turns DHIS2 into a powerful tool for managing patient care workflows on a facility or community level. For example, within a Tracker program, you can configure SMS reminders, track missed appointments and generate visit schedules for individual patients. The Tracker component also provides a simple tool for sharing critical clinical health data across multiple health facilities, including linking Tracker to an Electronic Medical Record (EMR) system.

In this thesis, DHIS2 Tracker versions 2.34 and 2.35 were analyzed for performance bottlenecks.

2.2 DHIS2 Implementations

There are several DHIS2 Implementations across the globe. At the time of writing this thesis, DHIS2 serves as the primary solution for collecting and analyzing health data in over 70 countries [9]. In some countries, there are national-level implementations for specific contexts like Measles Immunization or Covid Contact tracing. Some countries may also have multiple implementations for various contexts. For example, Sri Lanka has separate DHIS2 implementations for Covid Vaccination Tracking and Covid Contact tracing. Similarly, there are countries where DHIS2 is used both in the Health domain and Education domain as separate implementations.

DHIS2 implementations around the globe use either one or both of the components of DHIS2. Although the DHIS2 Tracker was first developed in 2010, there were very few implementations using it initially. Since 2016 there has been an increased demand and requirement of the DHIS2 Tracker module for several concrete implementations. The figure 1 shows the rise in the adoption of the DHIS2 Tracker module from 2010 to 2021[10]. The number of implementations using DHIS2 has almost doubled every year since 2016. The Covid Pandemic outbreak in 2019 resulted in several concrete DHIS2 Tracker implementations created for Covid Contact Tracing and Covid Vaccination Tracking. These national-level implementations have politically high visibility. Performance bottlenecks faced by such high-profile national-scale instances are not just an inconvenience, but a national crisis. Therefore the context of this thesis is important and time-critical. Performance analysis and optimization of database queries had to be completed in hours and days rather than weeks and months. This was also a motivation for me as I was able to work with the DHIS2 Core Team to help the different countries to ensure their national health efforts like Vaccination campaigns and Immunization campaigns become a success.

This kind of global adoption also increases the demand for better scalability. During my thesis work, I was able to support multiple large-scale implementations when they faced a crisis due to bottlenecks and performance issues in DHIS2. Some of the performance issues only surface on large-scale implementation. Those were the areas of potential improvement and optimization. In small-scale implementations, such botttlenecks

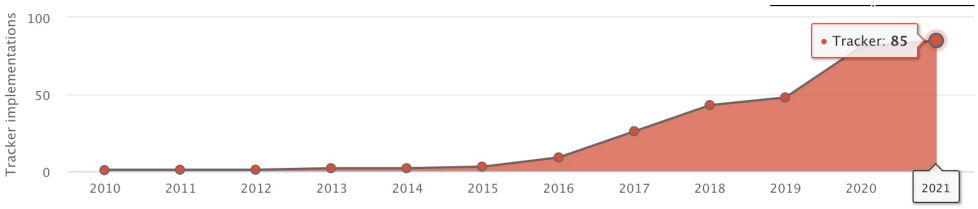

Figure 1: Number of DHIS2 Tracker implementations over the years

remain hidden. The performance issues explored and investigated in this thesis were mainly from the following large-scale DHIS2 Tracker implementations

- Bangladesh - Measles Immunization Tracking Instance [28]
- Sri Lanka - Covid Vaccination Tracking instance [29]
- Rwanda - Covid Vaccination Tracking instance [30]
- Nigeria - Covid Vaccination Tracking instance
- PEPFAR - DATIM instance

Supporting the large-scale implementations when they face performance issues is crucial for the continued adoption and growth of DHIS2. These performance issues make DHIS2 unusable in most cases, which forces countries to look for alternatives like paper-based reporting. Such incidents also discourage future potential implementers from using DHIS2 on their large-scale information system requirements.

In the next chapter, I define the research methodology used and explain the steps involved in the research process. I also list the data collection methods used for this thesis.

3 Research Methodology

This chapter describes the research methodology used in this thesis. I also describe the different data collection methods used for this work. I explain how the data collected were analyzed to achieve the optimization goals and answer the research questions. I reflect on some ethical and lawful considerations along with describing challenges faced during the research work.

The main objective of the study was to identify and resolve performance bottlenecks in a large-scale enterprise application. Bottleneck identification was scoped to include the Application Code and the Database interactions. To illustrate the issues and analyze the impacts of optimizations, I used several large-scale DHIS2 implementations as the basis for my study.

Empirical research is a type of research methodology that uses verifiable evidence to arrive at research outcomes. In other words, this type of research relies solely on evidence obtained through observation or scientific data collection methods. Empirical research on the research questions stated in section 1.2 has been done. I have chosen the DHIS2 Tracker module as the context of my research. This was chosen because it fits perfectly into the category of Large Scale Information systems and because of its relevance in these pandemic times. I observed various large-scale DHIS2 implementations to perform my case study. Due to the recent Covid pandemic, it was very important to help identify bottlenecks and resolve them with optimizations as fast as possible. Several countries were using DHIS2 for their Covid Vaccination tracking needs as well as other use cases. DHIS2 is an open-source information system software and serves as a digital global public good. Hence, ensuring scalability by bottleneck optimization is necessary for DHIS2 to remain a useful global public good.

This work does not follow analytical, mathematical optimization methods, but is rather based on an empirical approach. My approach was based on first identifying performance bottlenecks and limitations in large-scale data-intensive enterprise applications and then designing and implementing techniques to overcome these limitations. I focus on DHIS2 as part of my case study and the optimizations and findings are then generalized so that they can be applied for most enterprise applications that interact with a relational database.

3.1 Research Process and Collaboration

I work in the DHIS2 core development team. This gives me valuable insights and first-hand information on the performance bottlenecks and limitations that several large-scale DHIS2 implementations are facing. I volunteered to take part in performance analysis and experimenting optimizations to overcome these bottlenecks along with others in the DHIS2 Core development team. This thesis was completed in collaboration with the DHIS2 Core development, Quality Assurance (QA) team, and System

administrators (HISP Teams) of various large-scale DHIS2 implementations.

First, I conducted a literature study of recent research papers on the limitations and potential optimizations for large-scale enterprise applications that use the ORM framework and interact with relational databases. The results of this study allowed me to become familiar with the best practices and expert recommendations on large-scale applications. It also revealed common anti-patterns that gave me an insight into possible problems to tackle. To identify a potential bottleneck, I along with the DHIS2 core development team used monitoring tools like Glowroot and Munin and also referred to relevant log files when required. After having collected candidate opportunities for performance improvement, we verified the existence of the bottleneck, by experimenting on our simulated performance test environment. Once we had confirmed the presence of a shortcoming, we experimented with various optimization techniques to improve performance depending on the type of bottleneck. Next, we implemented the optimization technique, while trying to generalize it and ensure use-case independence. Finally, we evaluated our implementation by comparing the modified system which uses our optimization technique, to the original unmodified system. For the evaluation, we used both the synthetic environment and the real-world environment, where possible. In most cases, we used total execution time as the performance measure and also computed overheads separately, when necessary.

When a performance issue is reported by System administrators of large-scale DHIS2 implementations, we first identify the underlying root cause of the specific performance bottleneck. Then optimization possibilities were evaluated. In some cases, the evaluations were done and tested on the actual production implementation of DHIS2 to make sure the bottleneck is resolved. Based on the evaluation results, the optimizations were released along with the subsequent patch release of DHIS2 software. For the evaluation, we used a test database representative of a real-world large-scale DHIS2 implementation database. All comparisons and figures of optimization impact were done using the same application environment to ensure a fair comparison of performance parameters.

We set up a performance testing environment backed by a database that approximately represented real-world DHIS2 databases of large-scale instances. This simulated environment helped us to benchmark optimizations and analyze various performance metrics. During workload testing, workloads must be repeatable and easily reproducible to simulate multiple alternative scenarios with identical settings. We were able to reproduce several issues from the field in the performance environment and analyze them in isolation in our performance test environment. We could experiment with various optimization techniques for the identified bottlenecks and the impacts could be studied in a controlled manner.

There were multiple sources in the data collection process. They are listed below

1. I was in touch with system administrators of several large-scale DHIS2 implementations. System administrators of DHIS2 implementations in Rwanda, Sri Lanka, and Nigeria were all cooperative and supportive of my research work. We had several informal discussions and information sharing using the Slack messaging platform.

2. Glowroot access was provided for observation in several large-scale DHIS2 implementations. Glowroot provides a very useful dashboard with Slow traces and breakdown of various performance metrics. Query processing times, wait/block times, memory allocation, etc are some of the metrics shown in Glowroot slow trace. These slow traces can be exported, saved, and shared. Glowroot also has graphs to show the heap memory utilization and CPU load among other things. Request and Response payload will not be saved by Glowroot which ensures no confidential data (or Personally Identifiable Information) is visible in the Glowroot dashboard. However, request parameters that may include sub-strings of names and phone numbers will potentially be seen. But those were not exported or saved and were only used to identify the access pattern to be able to reproduce them in a simulated environment.

3. Application logs and PostgreSQL logs were shared by System maintainers to help the investigation. Only the relevant error stack traces in the logs were shared and this reduces the risk of sharing any confidential information.

4. Access to production database or their infrastructure was not required. System administrators were very cooperative to share query plans and other relevant information related to our performance analysis. This ensured that they had complete control of the data shared and can anonymize relevant data in case it was needed.

5. We created a performance testing environment that simulated a real-world DHIS2 Covid implementation database. This helped us experiment with several optimization approaches and evaluate the best candidate.

6. We conducted several formal and informal meetings, which involved a mix of QA engineers, System Administrators, Product managers, and members from the DHIS2 Core development team.

3.2 Ethical Obligation and Challenges

I ensured that no sensitive data are included in this thesis. The data that is presented in the listings are anonymous and do not characterize as PII data. I have only collected material relevant to the scope of the thesis. Observations of different performance metrics under varying data access patterns and data dynamics were done on production databases. Whenever I have received production data samples for analysis, I have made sure

to delete them after recording my observation. It was important to study and observe real-world production-grade applications to understand their behavior under load and unique data access patterns.

We faced several challenges throughout this work. First and foremost were the time constraints and urgency. In most cases, the issues were reported by large-scale implementations and I was part of the reactive efforts to identify the bottleneck and implement possible optimization suitable for the problem. Secondly, there were several large-scale DHIS2 implementations used for my thesis case study. This meant that the data dynamics, volume, and access patterns were not the same. So all the optimization techniques had to be made generalized as much as possible to make them applicable for all the systems. In some cases, our synthetic performance test environment did not accurately represent the real-world problem. In such instances, we had to seek the help of the system administrators of the affected implementation to get more information and context of the problem. The biggest challenge among all was infrastructure inconsistencies and issues related to infrastructure. Most of the country implementations were hosted in a private cloud environment maintained by the government or parastatal authorities. Some of the performance issues were caused by the infrastructure and it had a ripple effect on the application and database bottlenecks.

For the experiments, we focused on DHIS2 version 2.34 and 2.35. The exact version depended on the affected version reported from the field. To facilitate reproducibility, the performance test environment can be downloaded from the link provided in Appendix. The corresponding DHIS2 application WAR files can be downloaded from the DHIS2 downloads URL linked in the Appendix. The performance test environment was set up in virtual machines in a cloud environment. For brevity, most of the long listings of queries and query plans have been trimmed in the main section. The original full queries and query plans are added in the Appendix.

In the next chapter, we define what we mean by ”performance analysis”, ”bottleneck” and ”optimization” in the context of this thesis. I also review existing literature related to my research topic in the next chapter.

4 Literature Review

This chapter presents a review of literature that is deemed appropriate for the topics concerned in this thesis. Firstly, the relevant terminologies are defined, namely bottleneck, performance analysis, and optimization. Then a review of existing literature covering the different concepts follows. And finally, a summary of the literature review that explains my contribution to the literature.

4.1 Bottlenecks

In software terms, a bottleneck occurs when the capacity of an application or a computer system is limited by a single component, like the neck of a bottle slowing down the overall water flow. The bottleneck has the lowest throughput of all parts of the transaction path. Therefore developers will try to avoid bottlenecks and direct effort towards locating and tuning existing bottlenecks. Sometimes this happens after the software has been deployed in a live environment to be used by real traffic. Tracking down bottlenecks is called performance analysis.

4.2 Performance Analysis

Cortellessa et al. explain the difference between System vs Software Performance Analysis [6]. In System performance analysis, when a bottleneck is identified, the prevalent corrective actions mainly concern the hardware platform and its load. For example, to relieve an overloaded CPU, it is usually suggested to increase the multiplicity of the CPU or, in the best case, to deviate part of its load (through a load balancing system) toward less stressed CPUs.

On the other hand, in Software performance analysis, when a bottleneck is identified, the corrective action or suggestion is to introduce a change in the software rather than the system hardware. For example, excessive memory utilization can also be relieved by modifying the software to utilize memory more effectively.

Software performance analysis looks at how a specific program is performing daily and chronicles what slows down performance and causes errors now and what could pose a problem in the future. Performance issues are not always built into the software in a way that can easily be spotted through the QA process. Instead, it is something that can emerge over time after the project has been deployed and under diverse load. This thesis focuses on software performance analysis on large-scale DHIS2 implementations.

Several tools exist for performance analysis for various software technologies. In this thesis, the following external tools were used for analyzing the performance of DHIS2.

- Glowroot is an open-source Application Performance Management (APM) tool useful for monitoring Java-based applications [23]. It supports profiling Java applications. Glowroot dashboards were used extensively to monitor and observe the real traffic of large-scale DHIS2 implementations.
- Locust [17] is an open-source load testing tool that allows you to define user behavior and swarm your system with millions of simultaneous users. We used Locust to stress test DHIS2 and simulate workloads comparable with large-scale DHIS2 implementations.
- Apache Jmeter [14] is open-source software and a 100% pure Java application designed to load test functional behavior and measure performance. Jmeter was used locally on my laptop for some ad-hoc stress testing.
- YourKit Java Profile [31] is a fully-featured low overhead profiler for Java EE and Java SE platforms. YourKit was used to profile our performance test environment during stress tests.

Koçi et al. state a data-driven approach to measure usability of Web APIs[19]. Out of the six usability attributes elaborated in their work, the Efficiency attribute directly correlates to the performance of a Web API. The relevant sub-attributes are efficiency *In Task Execution* and efficiency emphIn Tied Up resources. Both of these metrics are significant when analyzing the performance of software through its exposed Web-APIs. In this thesis, response time and resource utilization are the main factors evaluated when analyzing the performance of DHIS2. In their work, Koçi et al. have also done a case study design on DHIS2 where they apply their proposed approach of measuring usability attributes by processing API logs. However, the case study is limited to computing the metrics for the know-ability attribute alone and not the efficiency attribute. This thesis can therefore complement their work by computing some of the metrics for the efficiency attribute of Web-APIs (non-exhaustively) in DHIS2.

Disk type and FileSystem type have an impact on transaction processing performance in PostgreSQL which is proven by Smolinksi's work with Storage space configuration [22]. However, as mentioned in the first chapter, storage configuration falls under infrastructure which is outside the scope of this research work. This thesis focuses on the DHIS2 application source code and the interaction with the PostgreSQL database.

4.3 Optimization

Performance Optimization is the process of modifying a software system to make it work more efficiently and execute more rapidly. Performance optimization is key in having an efficiently functional application. It is done by monitoring and analyzing the performance of an application and identifying ways to improve it. Performance optimization generally focuses on improving just one or two aspects of the system's performance, e.g execution time, memory usage, disk space, bandwidth, etc. This will usually require

a trade-off where one aspect is implemented at the expense of others. For example, increasing the size of the cache improves run-time performance, but also increases memory consumption.

There are numerous works on standard performance issues caused by common anti-patterns. These works also detail the optimizations that can be applied to eliminate the common anti-patterns. Using an Object-Relational mapping framework like Hibernate makes it more vulnerable to introducing such anti-patterns. Tse-Hsun Chen et al. have detected performance anti-patterns for applications developed using Object-Relational Mapping [5]. Their work specifically is focused on Java and Hibernate. ORM has always been a topic of performance analysis. The overhead required to map objects to their relational counterparts, and the amount of non-transparent logic contained in it makes it vulnerable to being a bottleneck. The same kind of analysis, as well as a performance aware refactoring, was done by Boyuan Chen et al. in their Industrial Experience Report [4]. However, their work was more focused on PHP and its ORM framework named Laurel. Gorodnichev et al. explore the use of ORM in their work[16] and conclude that if ORM is used competently by experienced developers then the overhead or impact on performance is negligible. This further reinforces the need to detect and remove anti-patterns in an ORM-backed application like DHIS2.

There are also numerous works on optimizing and configuring PostgreSQL database effectively [3, 21, 20]. However, all of these works explain different database configurations and query optimizations and suggest options. Since there is no one-size-fits-all configuration, fine-tuning the configuration and database schema for a large-scale DHIS2 instance is not trivial. This thesis translated the suggestions from their work into actionable points for specific large-scale DHIS2 implementations involved in my case study.

The work that is very closely related to my thesis work is by Dombrovskaya et al. that not only focuses on PostgreSQL Query Optimization but also delves into common pitfalls and anti-patterns of applications working with PostgreSQL [12]. I have extended their work by empirically analyzing the impact of various optimizations on a real-world large-scale time-critical information system like DHIS2.

4.4 Limitations

DHIS2 is a complex information system with varying scopes and contexts. As mentioned in the previous sections, more than 73 countries are using DHIS2 for collecting and analyzing health-related data. The scope of this thesis is limited to identifying the application and database query bottlenecks. All infrastructure-related bottlenecks are out of scope for this thesis. The different large-scale DHIS2 implementations are hosted by the corresponding countries on secure private cloud environments or physical servers. There are large-scale implementations hosted in the AWS cloud

platform and also on local physical machines with Linux. We do not have direct access to the systems or any sort of control on how much resources are provided for the instances. It is completely up to the implementers to decide how they want to size their system. There are some standard guidelines provided for setting up a DHIS2 instance.

The optimizations, wherever possible, are released as version upgrades. However, several implementations are reluctant to upgrade their versions to get these optimization benefits. There are states in India (Uttar Pradesh and Orissa) that are still using older DHIS2 versions like 2.28 that are not actively supported anymore. However, they wish to remain in the old version as they feel their requirements are satisfied with that version and do not want to upgrade to a more recent version bringing with it more complexities and sophisticated features. Such implementations will never have the benefit of the optimizations done during this research work until their policies change and they upgrade to the latest DHIS2 versions.

We also know that countries like Nepal and Ethiopia are stuck in DHIS2 version 2.30. This is because Nepal and Ethiopia have their native calendars which were supported until 2.30. These countries need date pickers for their native calendars. From version 2.30, front-end applications were modernized to be implemented with ReactJS and related frameworks. However, ReactJS does not have libraries for the Nepalese/Ethiopian calendar. There is no Material-UI support or library, which caused DHIS2 to drop support for native calendars from DHIS2 version 2.31. This technical limitation has forced Ethiopia and Nepal to continue using DHIS2 version 2.30. They are unable to upgrade to a better-performing version.

There are also limitations concerting governance and policies by the Country implementations. HISP teams can only provide recommendations. The ultimate decision is always made by the higher officials in a country and those decisions can be affected or influenced in many ways. Such infrastructural issues, governance, and policies are outside our control and also outside the scope of this research work.

4.5 Summary

This chapter introduced the reader to the concepts of bottleneck, performance analysis, and optimization. It also presents a review of existing literature on these concepts that are related to my research topic. As far as my search goes, there is no existing literature that details performance bottlenecks and evaluates the impact of optimization techniques on production-grade large-scale information systems like DHIS2. Through this case study, I have detailed some of the performance issues identified, the optimizations that were applied, and the impact of these optimizations on large-scale DHIS2 implementations.

5 Performance Analysis

An enterprise application will have several layers that constitute the entire software system. Application architectures are also described with *tiers* rather than *layers*. Tier usually implies a physical separation. An example is Client-server systems that represent a two-tier system, and the separation is physical. However, describing application architecture with layers is mainly a logical separation. These logical layers do not necessarily have to be run on different physical machines and can co-exist in the same machine. Performance analysis of an enterprise application can be done at each layer of the software system.

Layer	**Responsibilities and Components in DHIS2**
Presentation	Display of information to the end users through front-end Apps. Actions done by users in the form of mouse-clicks and keyboard hits translates into a series of HTTP Requests accessing the DHIS2 APIs served by the backend.
Domain	All the logic and functional rules of the system forms this layer. In DHIS2, a part of the backend handles all the business logic.
Data Source	This layer communicates with the database and uses appropriate queries to fetch data from database or persist data into the database.

Table 1: Three Principal Layers in the context of DHIS2

Although there are numerous ways to layer an application, I will be using one of the oldest and most popular architecture approaches as shown in table 1. This kind of architecture and its three principal layers have been a topic of discussion since the early 2000s. Fowler explains it clearly along with several other architectural patterns in their work from 2002 [15]. This kind of layering is very generic and can be applied to a large number of application implementations. In the table 1, the components and functionalities of DHIS2 that fall into each of these layers are also explained. In this thesis, for the case study with DHIS2, I have merged the Presentation layer and Domain layer into a single layer called *Application Layer* for the sake of simplicity. The *Data Source Layer* is the second layer that is referenced in this thesis. From my case study on large-scale DHIS2 implementations, I categorized the identified bottlenecks into one of the two layers - Application Side or Database Side.

Tracker Metadata Concept	Description and Examples
Tracked Entity Type	These are the types of entities that you want to track timeline data for. This can be a ”Person” or ”Vehicle” or ”Student” or any other type that you want to track based on the domain and use-case. In the health sector, the tracked entity type is usually a ”Person” or ”Patient”. This concept falls under metadata. Defining a tracked entity type is done by the system administrator as part of the system setup and is a one-time job.
Tracked Entity Attribute	These are the attributes that can be configured to be attached to either tracked entity type or program. Examples of tracked entity attribute that can be attached to a ”Person” tracked entity type are ”First Name”, ”Sex”, ”Date of Birth” and so on. Some attributes can be configured to be unique, for example, ”National Identification Number”.
Program	This is the definition of a program to which an instance of a tracked entity type can be enrolled. For example, Covid Vaccination Program, Maternity, and Child Health Program, and Tuberculosis Program. A specific patient, an instance of a tracked entity type, can be enrolled into one of these programs, and their treatment or participation in the program can be tracked from thereon.
Program Stage	These are the different stages that can be configured in a program. Each program can have one or many program stages. For example, in a Covid Vaccination Program, First Dose can be a program stage, Second Dose can be the second program stage. These stages can be configured to unlock sequentially as well.

Table 2: DHIS2 Tracker Metadata Concepts

Certain terminologies need to be familiarized to fully understand the performance bottlenecks in DHIS2 and their implications. The table 2 gives a brief description of the different metadata concepts in DHIS2 Tracker. Even though the list is not exhaustive, all the essential concepts of DHIS2 Tracker Metadata are covered based on the scope of this thesis.

The table 2 focuses on DHIS2 Tracker Metadata concepts. Metadata is part of the system configuration. Since DHIS2 is a generic application that can be tailored for different contexts by configuring the metadata. This customizability is crucial for enabling DHIS2 to support the concrete implementation context at hand. The configuration and metadata setup is done by the system administrators as a one-time activity as part of the system setup. Once the metadata setup is complete, there is rarely a need to change the metadata configuration. The table 2 also has examples for the specific metadata concepts that can be customized for a health domain use-case, like Covid Vaccination Tracking. Along with the concepts mentioned in the table, an additional concept worth mentioning is *Organisation Unit*.

In DHIS2 the location of the data, the geographical context, is represented as organisation units. Organisation units can either be a health facility or department/sub-unit providing services or an administrative unit representing a geographical area (e.g. a health district). Organisation units are located within a hierarchy, also referred to as a tree. The hierarchy will reflect the health administrative structure and its levels. Typical levels in such a hierarchy are the national, province, district, and facility levels. In DHIS2 there is a single organisational hierarchy so the way this is defined and mapped to reality needs careful consideration. Which geographical areas and levels are defined in the main organisational hierarchy will have a major impact on the usability and performance of the application[26].

Metadata is only a mechanism to set up a DHIS2 instance and configure it to collect data. The memory footprint for storing metadata is low as it does not include the actual data collected by the users. Data is entered based on this metadata, and therefore as long as the data collection continues, the memory footprint for storing this data keeps on increasing. In the table 3, some of the DHIS2 Tracker Data concepts are listed. It also has examples for the specific data concepts that have been customized for a health domain use-case, like Covid Vaccination Tracking.

The database schema design also plays an integral role in application and query performance. The best way to graphically represent a database schema design is using an Entity-Relationship diagram for the tables in the database. Figure 2 shows the E-R diagram of the relevant database tables of DHIS2 discussed in this thesis. Only those tables and columns that are relevant to the scope of this thesis are shown in the diagram. In multiple parts of the thesis, I have referenced the figure to better explain the bottleneck or related optimization.

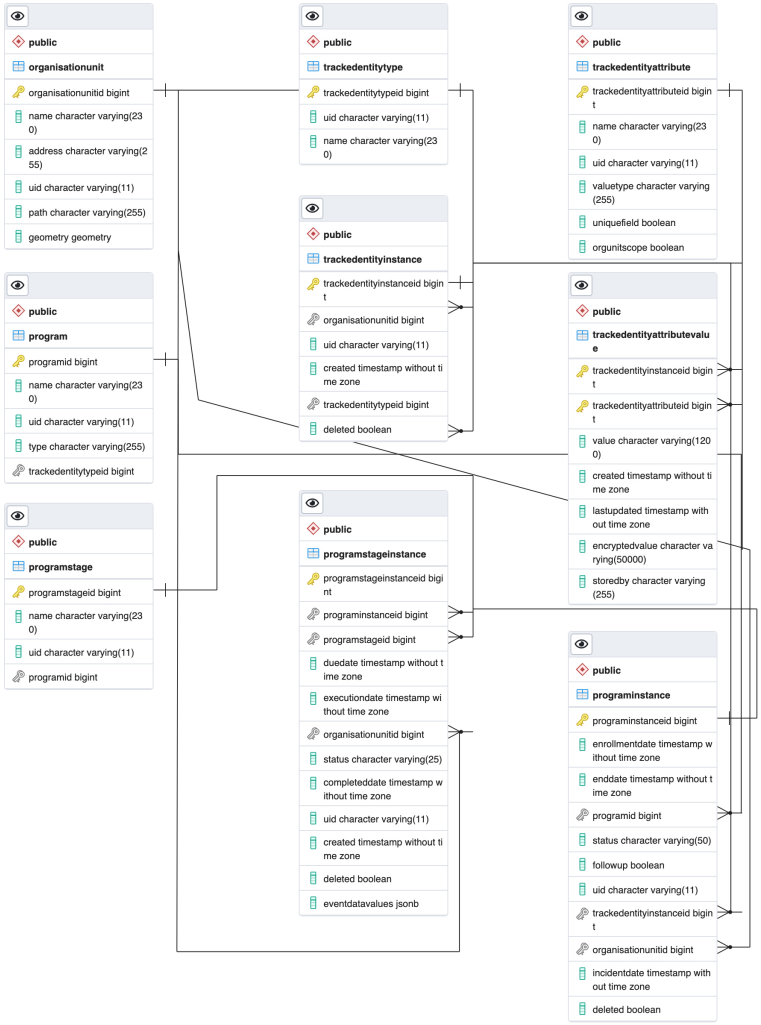

Figure 2: Entity-Relationship diagram of tables in DHIS2 Tracker

In the remaining part of this section, I list out the bottlenecks identified during my research work on the Database Side and Application Side separately.

Tracker Data Concept	**Description and Examples**
Tracked Entity Instance	This is an instance of the tracked entity type. If the type was ”Person”, then examples of tracked entities will be specific individual persons. In a national level implementation of DHIS2 Tracker, there will be as many tracked entities as the population of that country. Every citizen is a tracked entity instance. This concept falls under data. In health information instances, this is the actual patient data.
Tracked Entity Attribute Value	These are the values of a tracked entity attribute for a specific tracked entity instance. For example, ”David” is the value of tracked entity attribute ”First Name” for a specific Tracked Entity instance. This data is also crucial for searching and finding specific tracked entity instances.
Program Instance	A Program instance is an enrollment of a tracked entity instance into a program. Program Instance and Enrollment can be used interchangeably. For example, an Enrollment of a patient (a tracked entity) into the Covid Vaccination Program is a Program Instance.
Program Stage Instance	They are also called Events. These are the instances of program stages. For example, when the patient is enrolled into the Covid Vaccination program and then visits a clinic to get their first dose, an Event (Program Stage Instance) gets created. The details of that stage are populated in the system like the date of vaccine, type of vaccine, etc. This is also data and is attached to the enrollment as events.

Table 3: DHIS2 Tracker Data Concepts

5.1 Data Source Bottlenecks

The Data Source Layer mainly consists of the actual database and the technologies or tools used to communicate with the database. In the case of DHIS2, the database is PostgreSQL. Communication with the database mainly implies querying the database to fetch data and to insert or update specific records in the database. In the case of DHIS2, these communications are done using Hibernate ORM tool and Spring JdbcTemplate, all of which internally use the PostgreSQL JDBC driver. The performance bottlenecks in the data source layer are usually in-efficient queries or the absence of indexes. The bottlenecks explained here are also applicable to most Information Systems and not just DHIS2.

5.1.1 In-Efficient Queries

In-efficient queries are queries structured in such a way that the database query engine is unable to optimize the data fetch/update and thus leads to a significant response time. In-efficient queries are not declarative enough which does not give the optimizer better execution paths. There have been incidents in large-scale implementations where the query response time has been as large as 600000ms, which is 10 minutes. This essentially boils up to the DHIS2 API response time and the user is left hanging after performing a mouse-click and waiting for something to happen. Since Hibernate ORM is extensively used in DHIS2, there is a chance that in some cases hibernate constructs in-efficient queries under the hood as well. In other cases, it may be the application developer that has inadvertently modified an SQL query to add a feature and thereafter make it in-efficient.

Sri Lanka's Covid Vaccination Tracker instance suffered from an in-efficient query that caused a huge bottleneck. The bottleneck rendered DHIS2 useless and the health facilities were unable to meet the Sri Lankan Ministry of Health (MOH) deadlines and targets for vaccination per day. Each vaccine that was provided took a long time due to the high wait time in the application. The bottleneck and its impact were severe and several health facilities had to resort to paper-based and excel based tracking.

The issues reported from the field in Sri Lanka were the following

1. Delay in loading the front page list of the Tracker Capture App.
2. Delay in searching for a Tracked Entity Instance based on a unique attribute
3. Delay in registration of a new Tracked Entity Instance

On investigating the API calls that are invoked during these front-end user actions, there was a common API involved in all of the 3. The API that was performing poorly and taking minutes to respond was `https://base-url/api/trackedEntityI`

We used Glowroot to track down the individual queries generated within this API and identified an in-efficient query that consumed a lot of time. The corresponding query has been trimmed for brevity and is shown in listing 1. The full query can be found in Appendix.

```
SELECT instance,..., enrollment_status,
 "tJz1lz2sGrl".value AS ..,"m8xiBGIwDOT".value AS ..,
 "wi1E4HGW2zn".value AS ..,"XELfe4q9YMx".value AS ..,
 "xom4oPe793b".value AS .., "I2kOTyjBaL7".value AS ..,
FROM trackedentityinstance INNER JOIN trackedentitytype
 ON ..
INNER JOIN
    ( SELECT ... FROM trackedentityprogramowner
       WHERE  programid = 17609) AS tepo
ON ..
INNER JOIN
    ( SELECT  trackedentityinstanceid,
        Min( CASE WHEN status='ACTIVE' THEN 0
              WHEN status='COMPLETED' THEN 1
              ELSE 2 END) AS status
       FROM programinstance pi WHERE
       pi.programid= 17609 AND pi.deleted IS false
       GROUP BY trackedentityinstanceid ) AS en
ON ..
INNER JOIN organisationunit ou ON ..
INNER JOIN trackedentityattributevalue AS "tJz1lz2sGrl"
ON .. AND "tJz1lz2sGrl".trackedentityattributeid = 17633
        AND lower("tJz1lz2sGrl".value) = '783063093v'
LEFT JOIN trackedentityattributevalue AS "m8xiBGIwDOT"
ON .. AND "m8xiBGIwDOT".trackedentityattributeid = 26234
LEFT JOIN  trackedentityattributevalue AS "wi1E4HGW2zn"
ON .. AND "wi1E4HGW2zn".trackedentityattributeid = 17621
LEFT JOIN  trackedentityattributevalue AS "XELfe4q9YMx"
ON .. AND "XELfe4q9YMx".trackedentityattributeid = 357636
LEFT JOIN trackedentityattributevalue AS "xom4oPe793b"
ON .. AND "xom4oPe793b".trackedentityattributeid = 357642
LEFT JOIN trackedentityattributevalue AS "I2kOTyjBaL7"
ON .. AND "I2kOTyjBaL7".trackedentityattributeid = 17585
LEFT JOIN trackedentityattributevalue AS "pSTSMtz1Wpl"
ON .. "pSTSMtz1Wpl".trackedentityattributeid = 357648
LEFT JOIN  trackedentityattributevalue AS "edxMtP94nYO"
ON .. "edxMtP94nYO".trackedentityattributeid = 17592
LEFT JOIN  trackedentityattributevalue AS "E3rF2khHBXS"
ON .. "E3rF2khHBXS".trackedentityattributeid = 24772
LEFT JOIN  trackedentityattributevalue AS "fFXrgNH7SY6"
ON .. "fFXrgNH7SY6".trackedentityattributeid = 29959
WHERE tei.trackedentitytypeid IN (17581)
AND (ou.path LIKE '/GYBZ1og9bk7%') AND tei.deleted IS false
ORDER BY en.status ASC, lastupdated DESC
```

Listing 1: Identified In-Efficient Query

The query was sometimes taking several minutes to get a response from the database and this was happening on a powerful national DHIS2 tracker instance. A glimpse at the query made it obvious that this query does not scale well. The more attributes the implementation has configured, the more left joins with the *trackedentityattributevalue* table occurs. On analyzing the query plan, which is also added in Appendix, the specific areas of bottleneck within the query were identified.

The query plan analysis gave the following observations

- The chaining of joins (line 24 to 40) for every *trackedentityattributevalue* is not scalable. For the Sri Lanka Covid Vaccination Tracker instance, there were more than 10 attributes, and that resulted in more than 10 chained joins of a table that has over 200 million records. The default value of *join_collapse_limit* PostgreSQL configuration parameter is 8. This parameter caps the number of tables in a join that will still be processed by the cost-based optimizer. This means that if the number of tables in a join is eight or fewer, the optimizer will perform a selection of candidate plans, compare plans, and choose the best one [12]. But if the number of tables is nine or more, it will simply execute the joins in the order seen in the SELECT statement. Hence, this kind of chained left joins block the optimizer from choosing the best execution path. An alternative way of fetching the same set of data without these many joins had to be investigated.
- The default sorting is based on a computed column of status in the table *programinstance*. A decision had to be made whether the default sorting has to be such a costly one. Default sorting is used when the API client does not explicitly request sorting by any specific parameter. A non-costly default sorting would have been more optimal to avoid this bottleneck.

By understanding the problematic areas in the query structure, we rewrote this query as elaborated in section 6.1.1. The rewrite considered the above observations, alleviated the scalability issues while still satisfying the requirement of the API. The optimization results are also explained in section 6.1.1.

5.1.2 Absence of Indexes

In the majority of the cases, the SQL query is already declarative and optimized structurally. There may not be a way to rewrite it to make it more efficient. Despite being optimized on paper, there are cases where such a query is performing poorly on large databases. This is because the query optimizer does not have the necessary help to speed up data fetching. If the execution path involves full table scans to fetch data from a large table, then it will certainly be slow. A tool that can help the query optimizer to further speed up data lookup is an Index.

What exactly is an index? Indexes are redundant data structures that are invisible to the application and are designed to speed up data selection based on criteria [12]. The redundancy means that an index can be dropped without any data loss and can be reconstructed from data stored in the tables. Invisibility means that an application cannot detect if an index is present or absent. Any query produces the same results with or without an index. An index is created to improve the performance of a specific query or several queries.

If appropriate indexes are not created, the query engine sequentially scans the entire table and that consumes time. To obtain the execution plan for a query, the EXPLAIN command is run. This command takes any grammatically correct SQL statement as a parameter and returns its execution plan[12]. Analyzing the execution plan gives valuable insights into how the query optimizer decided to execute the query. It displays the usage of indexes (if any), estimations of costs, the expected number of rows in the output for that specific query execution.

In this section, I list two major types of performance bottlenecks that multiple large-scale DHIS2 instances faced due to lack of indexing. Both of the issues revolve around one of the largest tables in the DHIS2 database, the *trackedentityattributevalue* table. If an instance has 10 attributes configured for a *trackedentitytype*, then the number of *trackedentityattributevalues* will be 10 times the number of *trackedentityinstances*. The relations between these tables can be referred to in figure 2. In the Sri Lankan Covid Vaccination tracker instance, there were 17 Million *trackedentityinstances* and more than 10 *trackedentityattributes* configured. This meant the size of the *trackedentityattributevalue* table was well over 200 Million records.

Nigeria DHIS2-Vaccination instance reported the slow query shown in listing 2. Such queries with cross joins are created internally by Hibernate ORM used within DHIS2, more of this is covered in the next section 5.2. In this section, the slow query is analyzed to identify the bottleneck.

```
1 SELECT trackedent0_.uid AS col_0_0_
2 FROM   trackedentityinstance trackedent0_
3 WHERE  ( EXISTS (
4 SELECT trackedent1_.trackedentityinstanceid,
```

```
5  trackedent1_.trackedentityattributeid
6  FROM   trackedentityattributevalue trackedent1_
7     CROSS JOIN trackedentityattribute trackedent2_
8      WHERE  trackedent1_.trackedentityattributeid =
9           trackedent2_.trackedentityattributeid
10      AND trackedent1_.trackedentityinstanceid =
11          trackedent0_.trackedentityinstanceid
12      AND trackedent2_.uid = 'izttywqePh2'
13      AND Lower(trackedent1_.value) = 'ng-tm10697501lx'
14      ) ) AND trackedent0_.deleted = false
```

Listing 2: Slow Query reported by Nigeria DHIS2 Vaccination Instance

We executed the **EXPLAIN** command on the query in listing 2 and analyzed the query plan. The performance bottleneck was evidently in one of the filter checks corresponding to Line 13. The figures 3 and 4 shows a graphical representation of the node in the query plan that causes the bottleneck. The filtering check on the **lower(value)** comparison is not part of an index and therefore is causing the bottleneck.

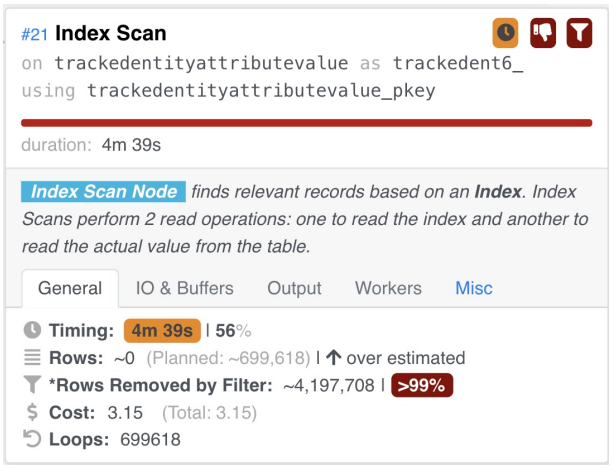

Figure 3: Query plan node that consumes the most time

In figure 3, we see the time taken to process this specific operation by the query executor is 4 minutes and 39 seconds. We also see that over 99% of the rows are removed by the filter. For such a restrictive filter, an index is essential [12]. Even though the operation is an Index Scan, the index that is scanned is the primary key of the *trackedentityattributevalue* table and does not help to speed up the filter condition on **lower(value)**. This is evident from figure 4, where the index condition and filter are shown clearly.

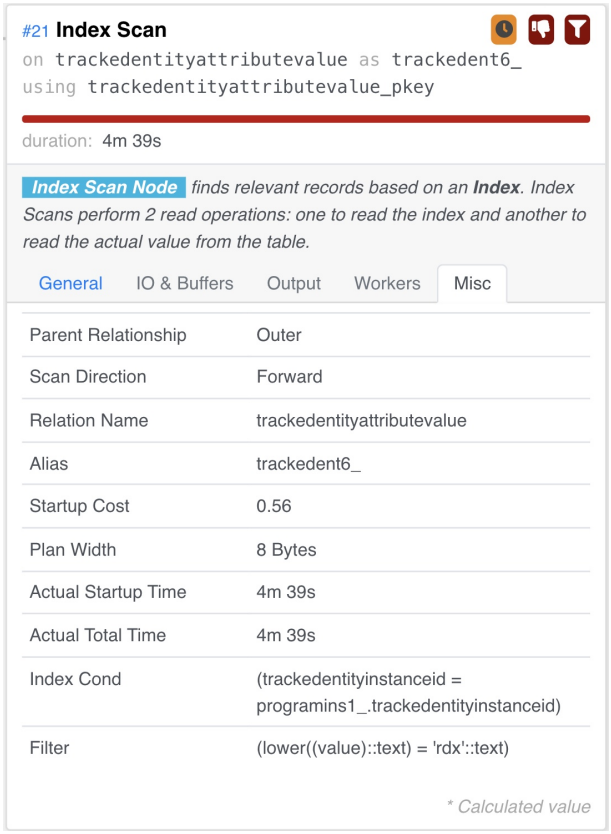

Figure 4: Query plan node details showing the filter applied

Another type of performance issue on a specific query was reported by the Rwanda DHIS2 Covid Vaccination tracker instance. The issue, if left unattended, would have compromised their National Covid Vaccination Campaign which was running for a few weeks and had an estimate of 100k vaccinations per day. The query is shown in listing 3 which also uses the *trackedentityattributevalue* table. The query is trimmed for brevity, the full query can be found in Appendix.

```
SELECT instance, ... ,inactive,
String_agg(TEA.uid  || ':' || TEAV.value, ';')
FROM (SELECT ...
FROM (SELECT ...
     FROM   trackedentityinstance TEI
     INNER JOIN trackedentityattributevalue "ciCR6BBvIT4"
     ON "ciCR6BBvIT4".trackedentityattributeid = 3465
     AND "ciCR6BBvIT4".trackedentityinstanceid =
       TEI.trackedentityinstanceid
     AND Lower("ciCR6BBvIT4".value) LIKE '%0784003172%'
     INNER JOIN trackedentityprogramowner PO
     ON .. AND PO.programid = 3541
```

```
13        INNER JOIN organisationunit OU
14        ON .. AND ( OU.path LIKE '/Hjw70Lodtf2%' )
15        WHERE  TEI.trackedentitytypeid IN ( 3501 )
16        AND TEI.deleted IS FALSE
17        AND EXISTS (SELECT PI.trackedentityinstanceid
18           FROM   programinstance PI
19           WHERE  PI.trackedentityinstanceid =
20           TEI.trackedentityinstanceid
21           AND PI.programid = 3541
22           AND PI.deleted IS FALSE)
23        ORDER  BY TEI.trackedentityinstanceid ASC
24        LIMIT  21 offset 0) TEI
25        LEFT JOIN trackedentitytype TET ON ..
26        LEFT JOIN trackedentityattributevalue TEAV
27        ON .. AND TEAV.trackedentityattributeid IN ( ... )
28        LEFT JOIN trackedentityattribute TEA ON ..
29        GROUP  BY ...
30        ORDER  BY TEI.trackedentityinstanceid ASC;
```

Listing 3: Slow attribute search query reported by Rwanda DHIS2 Vaccination Instance

On analyzing the query plan for the query in listing 3 the performance bottleneck was once again in one of the filter checks on lower(value) corresponding to Line 10. The figures 5 and 6 shows a graphical representation of the node in the query plan that causes this bottleneck. The filtering check on the **lower(value)** comparison is not part of an index and therefore is causing the bottleneck.

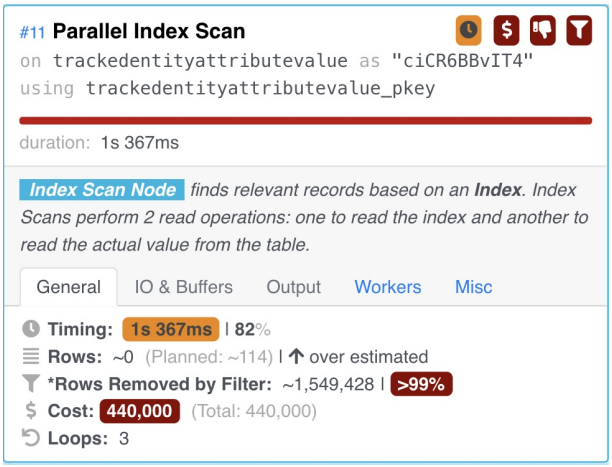

Figure 5: The bottleneck Query plan node in Rwanda

It has to be noted that the query in listing 3 is an optimized version of the in-efficient query in listing 1. Even after making the required structural

changes for better efficiency, the query still struggled to perform under load for certain criteria checks.

In figure 5, we see the time taken to process this specific operation of like comparison with a double-ended wildcard on **lower(value)** is 82% of the total time required to execute the query. The timing of 1 second 367 milliseconds should not be taken lightly, as the query plan was obtained during off-peak hours from the Rwanda DHIS2 instance. With the vaccination campaign going in full swing during the peak time, the number of similar queries hitting the database concurrently is high and the bottleneck was stressing the database. For such a high execution frequency of a query, 82% of the time spent on filtering the condition becomes a bottleneck that is worth optimizing.

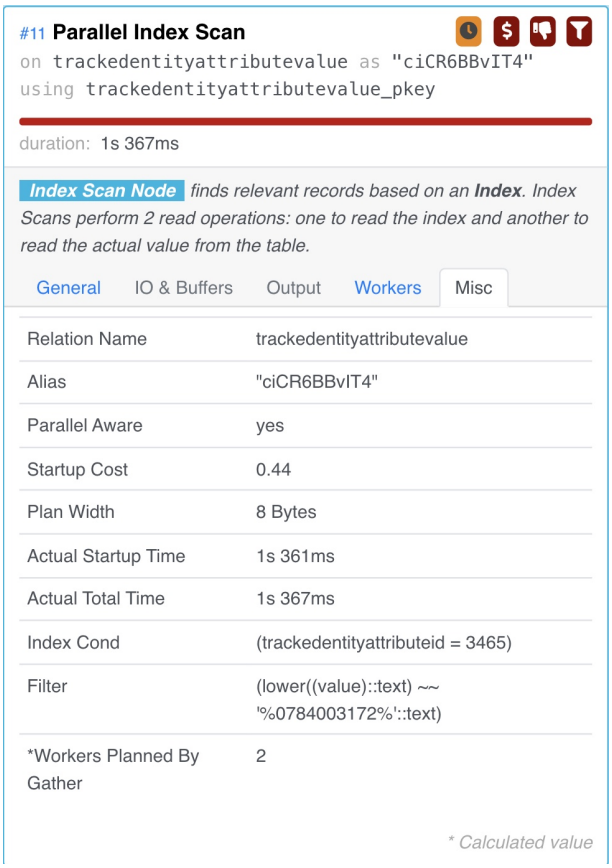

Figure 6: Query plan node details showing the *like* comparison filter with double ended wildcard (%)

A very similar performance issue on a specific query was reported by the Nigeria DHIS2 Covid Vaccination tracker instance. They had their metadata configured in such a way that, Covid vaccination verification QR code was set up as an event data value. The query to search for a specific QR

code was very slow. The search functionality was used at airports and other ports of entry to verify if the passengers were vaccinated by searching for the QR code presented by them. We got reports that passengers were unable to board their flights on time because of this performance bottleneck and huge response time for searching. The corresponding trimmed query is shown in listing 4. The full query can be found in Appendix.

```
SELECT *
FROM   (SELECT ..
lower(psi.eventdatavalues #>> '{LavUrktwH5D, value}')
..  FROM   programstageinstance psi
INNER JOIN programinstance pi
ON ..
INNER JOIN program p
ON ..
INNER JOIN programstage ps
ON ..
INNER JOIN categoryoptioncombo coc
ON ..
INNER JOIN categoryoptioncombos_categoryoptions
ON ..
INNER JOIN dataelementcategoryoption deco
ON ..
LEFT JOIN trackedentityinstance tei
ON ..
LEFT JOIN organisationunit ou
ON ..
LEFT JOIN organisationunit teiou
ON ..
LEFT JOIN users auc
ON ..
LEFT JOIN userinfo au
ON ..
LEFT JOIN (SELECT categoryoptioncomboid,
        Count(categoryoptioncomboid)
         FROM   categoryoptioncombos_categoryoptions
        GROUP  BY categoryoptioncomboid) AS cocount
         ON ..
LEFT JOIN (SELECT ..
     FROM   dataelementcategoryoption deco
     LEFT JOIN dataelementcategoryoptionusergroupaccesses
     ON ..
     LEFT JOIN dataelementcategoryoptionuseraccesses
     ON ..
     LEFT JOIN usergroupaccess
     ON ..
     LEFT JOIN useraccess ua
     ON .. WHERE  ..) AS decoa
ON ..
```

```
43 WHERE  Lower(psi.eventdatavalues #>> '{LavUrktwH5D,␣value}')
44         LIKE '%nphcda000005013%'
45 AND p.programid = 64519
46 AND psi.deleted IS FALSE
47 AND ( p.uid IN ( .. ) )
48 AND ( ps.uid IN ( .. ) )
49 ORDER  BY psi_lastupdated DESC
50 LIMIT  2 offset 0) AS event
51
52 LEFT JOIN (SELECT ..
53       FROM   programstageinstancecomments psic
54       INNER JOIN trackedentitycomment psinote
55       ON ..
56     LEFT JOIN users usernote
57     ON ..
58     LEFT JOIN userinfo
59     ON ..) AS cm
60 ON ..
61 ORDER  BY psi_lastupdated DESC
```

Listing 4: Slow QR Code search query reported by Nigeria DHIS2 Vaccination Instance

The bottleneck for the query in listing 4 was in one of the filter check on `lower(psi.eventdatavalues #>> '{LavUrktwH5D, value}')` corresponding to Line 43-44. The figures 7 and 8 shows a graphical representation of the node in the query plan that causes this bottleneck. The filtering check on the `lower(psi.eventdatavalues #>> '{LavUrktwH5D, value}')` comparison is not part of an index and therefore is causing the bottleneck.

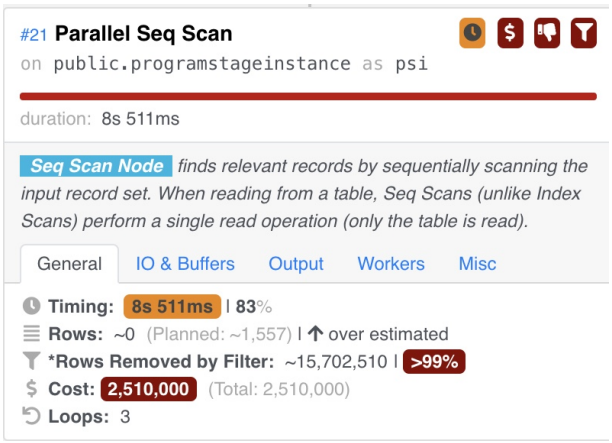

Figure 7: Query plan node in Nigeria that shows bottleneck with QR code searches

In figure 7, we see the time taken to process this specific operation of like comparison with double ended wildcard on `lower(psi.eventdatavalues #>> '{LavUrktwH5D, value}')` is 83% of the total time required to execute the query.

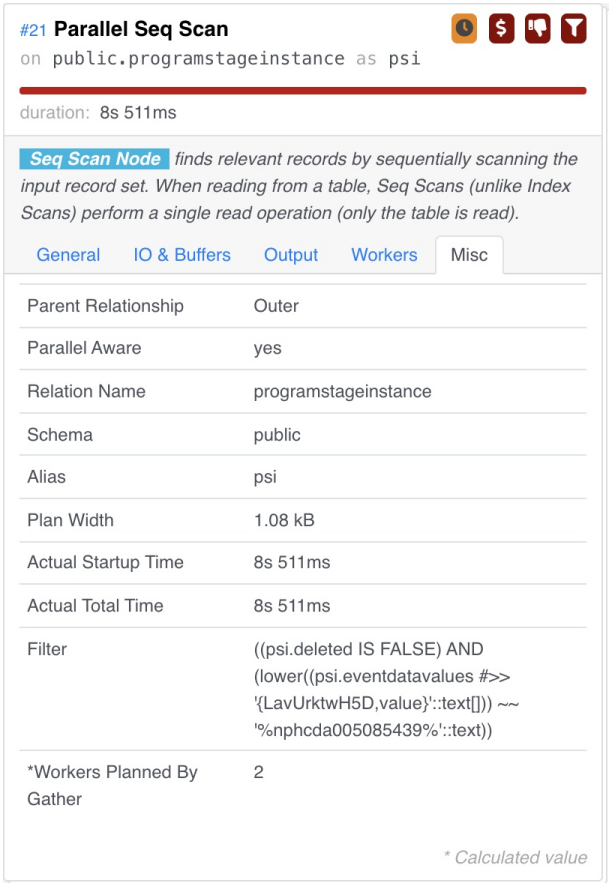

Figure 8: Query plan node details from Nigeria showing the *like* comparison filter with double ended wildcard (%)

The bottleneck for both types of issues was identified as the filters with like or equality comparisons on large tables. The type of the first issue used an *equality* comparison operation whereas the latter issue used the *like* operation with a double-ended wildcard (%). Hence, both are treated differently and the optimization for each is different. Details of how these 2 performance bottlenecks were optimized are explained in section 6.1.2.

5.2 Application Bottlenecks

In the previous section 5.1, I focused on Database bottlenecks which were mainly about optimizing database query performance. Database queries are parts of an application, and this section concerns optimizing processes within the application rather than individual queries. If we do not address process deficiencies, it could easily cancel any performance gained from individual queries. In this section, I focus on listing the identified bottlenecks on the Application Layer. Out of the layers described in table 1, the Presentation Layer and Domain Layer collectively form the Application layer discussed in this section.

A bottleneck in the Application layer means there is an inefficiency in a section of the code. Application bottlenecks can increase the response times and decrease the throughput of all the RESTful APIs that touch the inefficient section of the code. Such application Bottlenecks can either be on the client-side or server-side. The frontend (client-side) is the presentation layer that is developed using JavaScript and related technologies. There could exist a sub-optimal function or flow in the frontend JavaScript application which in large-scale systems can even cause the client-side browser to crash or hang. Sub-optimal code execution can also exist in the Java-based backend application running in a JEE compliant container like Tomcat. This can lead to resource exhaustion and system crashes. Just as database bottlenecks, application bottlenecks also tend to be prominent only in large-scale Enterprise applications with user access patterns.

Faroult illustrated a great analogy for common performance bottlenecks on the application side in their work [13]. The analogy is called the *Shopping List Problem*.

Suppose you have a shopping list for the grocery store. In real life, you would get into a car, drive to the grocery store, pick up all the items on your list, get them into the car trunk, drive home, bring them inside, and put them into your fridge. But Imagine that, instead of this, you would drive to the store, come in, pick just the first item from your shopping list, drive back home, place this item in the fridge, and head to the store again. Imagine repeating the same sequence of actions for each item on your list. This sounds extremely inefficient for shopping. The fact is that such an inefficient mechanism is seen in many applications when it comes to their interaction with databases or APIs.

To put things further into perspective, imagine that to improve the speed of shopping, experts suggest that we should increase the width of the isles in the store or build better highways or equip the car with a more powerful engine to reach the store faster. Some of these suggestions could, indeed, improve the situation. But no improvement can be compared with the gains achieved with one simple process improvement - picking up all groceries during one single trip.

This shopping list problem is often translated into application behavior. Most performance problems are caused by too many queries that are too small or too many HTTP requests that are very closely related. The same analogy on better highways and powerful car engines can be applied on the application side like listed here [13]

1. More powerful computers do not help much, as both the application and the database are in a wait state for 99% of time.
2. Higher network bandwidth does not help either. High-bandwidth networks are efficient for the transfer of bulk amounts of data but cannot significantly improve the time needed for roundtrips. Time depends on the number of hops and the number of messages but does not depend significantly on message size. Furthermore, the size of the packet header does not depend on the message size; hence, the fraction of bandwidth used for payload becomes small for very short messages.
3. Distributed servers might improve throughput but not response time as an application sends data requests sequentially.

Therefore, the best way to optimize the shopping list problem is by reducing round trips. The same pattern of the shopping list problem exists in most enterprise applications. In the remaining sections of this chapter, I focus on such performance analysis of application bottlenecks in the DHIS2 application.

5.2.1 In-efficient API Access Pattern

Sometimes the RESTful API itself be very optimized, but an inefficient way of accessing the APIs (the process) becomes the bottleneck. For example, assume there is a customer registration form that is rendered as part of a front-end App. There would naturally be a Customer Registration API or Customer Details Update API in the backend which the front-end App will have to consume. But if the front-end app invokes the API for every input field entered by the user, the amount of API invocation will be large. Instead, if the front-end app waits till all the user input fields are populated and then sends all the data using a single API invocation, the number of round-trips to the backend decreases considerably and performance improves drastically. Invoking the API for every input field is similar to the Shopping List problem described in the previous section 5.2.

A performance issue of a similar nature was reported in Sri Lanka DHIS2 Covid Vaccination Tracker Instance. The Tracker Capture App was used to track the vaccination doses given to each resident in Sri Lanka.

System administrators observed that there existed intermittent database row-level locks when updating event data values, table *programstageinstance* shown in 2. We investigated a typical end-user flow through the Tracker Capture App and observed that for each field input in the Event

Data Values section as shown in figure 15, there was an HTTP request sent to the backend to update that single field value. During peak hours of vaccinations in various facilities, this means that the number of APIs being consumed in the backend was a multiplier of the number of the input fields. The same end-user was filling in multiple inputs within a few seconds and a sequence of API updating the event data values for the same enrollment was fired into the backend server. In a large scale highly concurrent system, these excessive API round-trips have exponential performance degradation.

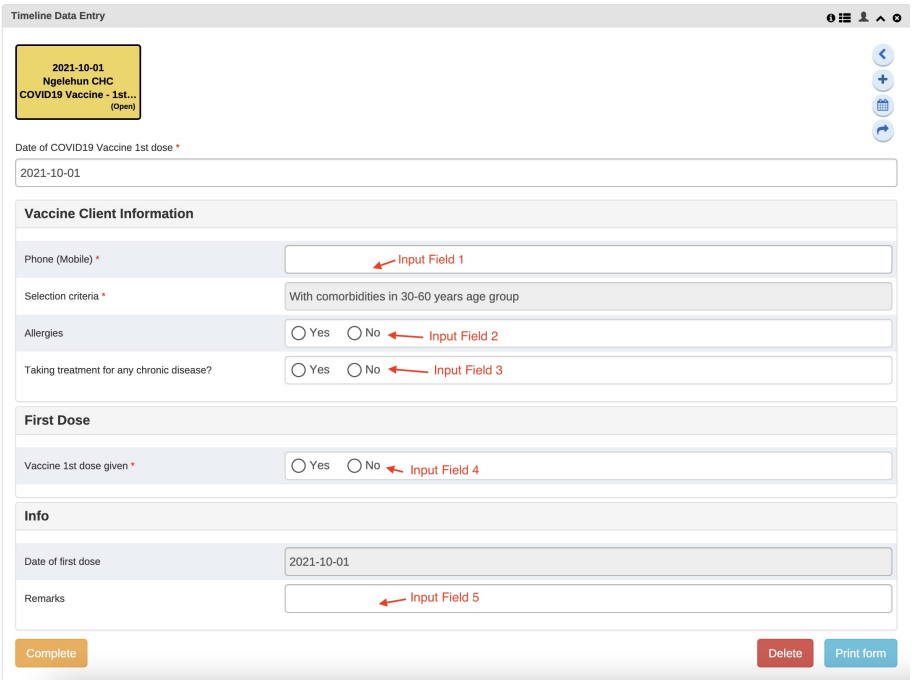

Figure 9: Tracker Capture App in Sri-Lanka with 5 to 10 input fields

From the entity-relationship diagram figure 2, the table *programstage-instance* stores the data for events. As seen in the figure, the table has a column *eventdatavalues* which is of type jsonb. All the event data values corresponding to a single event are stored in a denormalized way in a single column for each event. This further aggravated the problem of multiple API hits to update individual parts of the same event row and thereby increasing the chances of database row-level locking and waiting. The figure 10 shows the User Interface, which shows a green background color when an input field is populated by the user. This reflects an API request hitting the backend and if the response is a success, the background turns green. Note that only the specific event data value, the *Phone(Mobile)* field is sent as part of the API request for updating. This process repeats for each input

field. For the Sri Lanka DHIS2 Covid Vaccination tracker instance, there were on average 5 to 10 input fields.

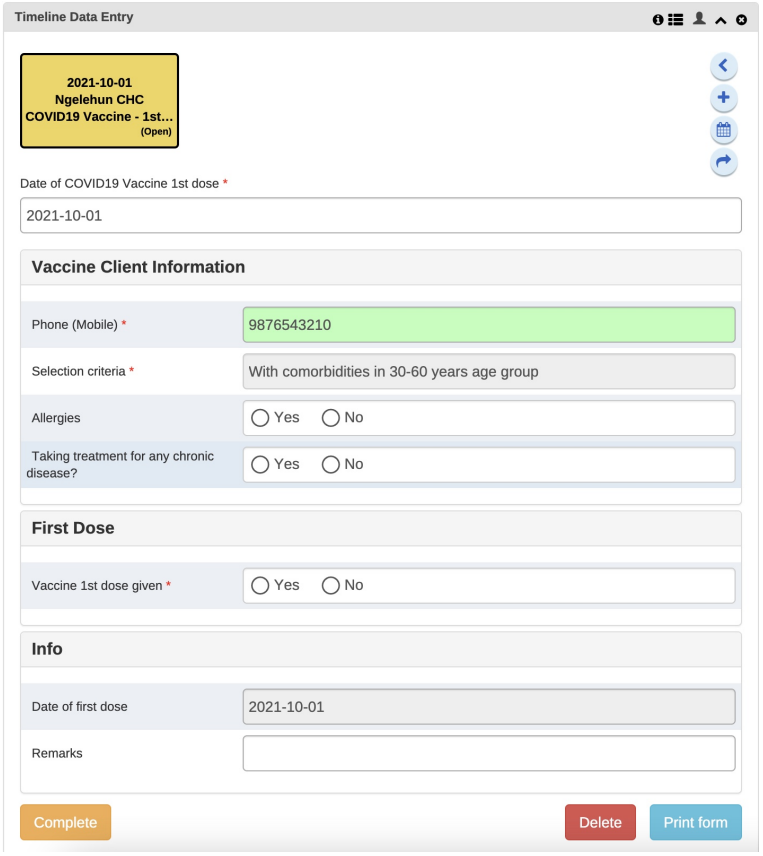

Figure 10: Updated input fields shown with green background colour in Tracker Capture App

The listing 5 shows the details of the event data value update API that updates a single event data value for each API requested. An example payload is also shown in the listing. The payload has only one event data value contained in it and only that event data value will be updated in this specific API. The figure 11 shows the response timing information as recorded by the Google Chrome browser. The TTFB (Time to First Byte) is the time it took for the first byte of the response to reach the browser.

```
URL: /dhis/api/30/events/*/*
Request Method: PUT
Request Payload:
{
    "event": "iERafKJQ3aP",
        "orgUnit": "DiszpKrYNg8",
    "program": "aLZQ5fSVdQc",
    "programStage": "fZ3diyIwzDF",
    "status": "ACTIVE",
    "trackedEntityInstance": "iWxmynYsPGr",
    "dataValues": [{
            "dataElement": "TYn17QNtyNV",
            "value": "false",
            "providedElsewhere": false
    }]
}
```

Listing 5: Request Payload example for single event data value update API

This proved to be a performance bottleneck in the Front-end Application, namely *Tracker Capture App*. This in-efficient behavior has existed in the App for several years. But only in the large-scale covid vaccine implementation, it starts creating a bottleneck due to the magnitude and frequency of data entry in the field. The high concurrency of the event data value updates for the Covid Vaccination implementations exposed the bottleneck in the application.

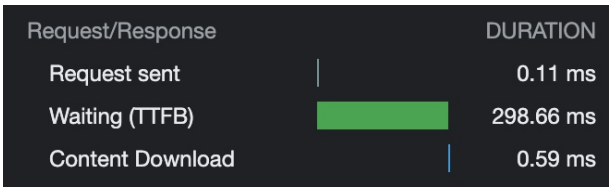

Figure 11: Response timing of single event data value update API

5.2.2 ORM Pitfalls

In this section, I cover the common pitfalls and anti-patterns that arise when using the ORM framework in an enterprise application. I have also explained how Hibernate ORM framework used within DHIS2 has created some performance bottlenecks.

An object-relational mapper (ORM) stands for a program that maps a database object to the in-memory application object. An ORM usually provides means for retrieving data from the database and mapping them into

Java objects. ORM also runs arbitrary database queries which may not be optimized. However, in practice, generated queries are almost always used due to time pressures and the simplicity with which they are created in the application. Because the actual database code is obscured from the developer, database operations on sets of objects end up happening very similarly to this: an ORM method returns the list of object IDs from the database, and then each object is extracted from the database with a separate query (also generated in the ORM). Thus, to process N objects, an ORM issues N+1 database queries, effectively implementing the shopping list pattern described in the previous section.

There are also other factors to consider when using an ORM framework like Hibernate. Fetching whole objects and their associated graphs into memory eats a huge chunk of the available heap memory. Such high memory allocation leads to the application starving of memory and performing poorly. In worst cases, such high memory consumption can also crash the application. This kind of performance issue is a result of using ORM mapping directly and letting it fetch the entire associations under the hood. The reason for poor performance from ORM is the incompatibility of database models and programming language models that can be expressed via the impedance mismatch metaphor. The power of the expressiveness and efficiency of database query languages does not match the strengths of imperative programming languages. Both imperative programming languages and declarative query languages work extremely well to accomplish the tasks they were designed for. The problems start when we try to make them work together. Thus, the reason for poor performance is an incompatibility of database models and programming language models. However, that does not mean ORM frameworks have to be avoided. Gorodnichev et al. has validated in their work that the use of the ORM framework competently introduces negligible performance issues [16].

A performance issue was reported in the Bangladesh DHIS2 Measles Immunization instance where the Front-end app was crashing during one of their workflows. The issue was in the ”Capture App”. The App fetches all the *categoryOptions* (A domain entity mapped by ORM) from the backend using an API request. Each *categoryOption* has an associated list of *OrganisationUnits*. In a large-scale DHIS2 instance like the Bangladesh DHIS2 Measles Immunization instance, this translates to around 50,000 category options with each category option being associated with more than 5000 organisation units. The organisation Unit association indicates whether the corresponding categoryOption can be used/accessed by the organisation unit.

In a typical DHIS2 installation, users will not have access to all the organisation units. Organisation unit can be compared to a health facility. Therefore, only the health workers in that facility will have the authority to write data into that organisation unit. This meant that the front-end app was unnecessarily fetching the entire list of 250,000 (50,000 multiplied by 5,000) objects from the backend and then filtering out the inaccessible ob-

jects by traversing through them. This traversing caused several browser clients to crash. The fact was that out of these 250,000 objects only a very small percentage would be relevant to the logged-in user based on the user's access permissions.

An optimization fix was implemented to enable the Bangladesh National Immunization campaign to proceed without blockers. The optimization involved creating a new parameter *restrictToCaptureScope* in the same API request, which when set to true, the backend server will perform all the filtering and traversing logic. The response sent back then will be a small payload that contains domain objects relevant to the logged-in user and no further traversing or filtering was required from the Front-end App. This ensured the frontend App never crashed due to the small payload it received back from the server, and also the payload contained only the relevant data.

The listing 6 shows the significant code snippet on how the quick fix was done for the traversing and filtering logic. Note that, the entities here are all Hibernate ORM mapped entities. The method invocation of *getOrganisationUnits()* on line 7 will internally generate a database query and fetch all the associated organisation unit objects from the database for that specific categoryOption alone. This happens in a for loop as shown in line 5 of the listing 6. This is another example of the hibernate n+1 query problem which is also illustrated by the Shopping list problem in the previous section. The fix has eliminated the category options that are not accessible to the logged-in user but still has to trim the associated organisation unit list by traversing through them.

```
1  Set<String> orgUnits = organisationUnitService.
       getOrganisationUnitsByQuery(params).stream().map(orgUnit - >
        orgUnit.getUid()).collect(
2      Collectors.toSet());
3
4  //This for loop does the traversal of ORM entities
5  for (T entity: entityList) {
6      OrganisationUnitAssignable e = (OrganisationUnitAssignable)
           entity;
7      if (e.getOrganisationUnits() != null && e.
           getOrganisationUnits().size() > 0) {
8      //for each entity, the association is fetched and filtered.
9          e.setOrganisationUnits(
10             e.getOrganisationUnits().stream().filter(ou - >
                   orgUnits.contains(ou.getUid())).collect(
                   Collectors.toSet()));
11     }
12 }
```

Listing 6: Java code snippet showing a common ORM anti-pattern

In addition to the multiple round trips to the database because of the n+1 queries problem, each list of associations is also large. The quick fix

helped the Bangladesh National Immunization campaign only because of their user access configuration. However, the fix introduced a new performance bottleneck in another large-scale instance, the PEPFAR DATIM Instance. The memory allocated to serve the same request was huge that the backend application crashed with an OutOfMemory error due to GC (Garbage Collection) overhead limit getting exceeded.

Further investigation into the DHIS2 PEPFAR DATIM instance revealed that most of their users had access to all the organisation units (Admin Users). This essentially meant that the entire set of 250,000+ objects was loaded into memory and traversed to check if any filtering is required. This huge memory footprint exhausted the heap memory of JVM which caused it to crash. We realized that the quick optimization fix applied earlier was not an optimized solution and proper optimization was required to support the feature irrespective of how the users in the system are configured.

We simulated the same user configuration in a performance testing environment and were able to reproduce the massive memory allocation to serve a single request. The figure 12 shows the metrics captured by Glowroot [23]. 1GB of memory was allocated to serve a single request. Note that 1GB was observed in a simulated performance test environment. This figure can go beyond 10 GB depending on the size of the instance. The cost of invoking the method to get the associated objects in an ORM mapped entity is often overlooked by developers and goes unnoticed until its being used in large-scale implementations. This leads to transactions with timeouts or hangs in large-scale systems.

5.2.3 In-Efficient Resource Utilization

A system resource is any physical or virtual component of limited availability within a computer system. All connected devices and internal system components are resources.

Effective resource management includes both preventing resource leaks (not releasing a resource when a process has finished using it) and dealing with resource contention (when multiple processes wish to access a limited resource). CPU time, Random-access memory (RAM), Disk drives, Cache Spaces, Network sockets, Input/Output operations are all examples of some of the general system resources. A robust application ensures efficient use of the resources of the system.

Actual resource management is a broad topic that I do not cover in this thesis exhaustively. However, I focus on one specific aspect of resource management which is always essential in any enterprise application that interacts with a database. Database interaction in Java applications is done using JDBC drivers that establish a connection to the Database. The drivers connect to the database using a TCP connection. Once the connection is

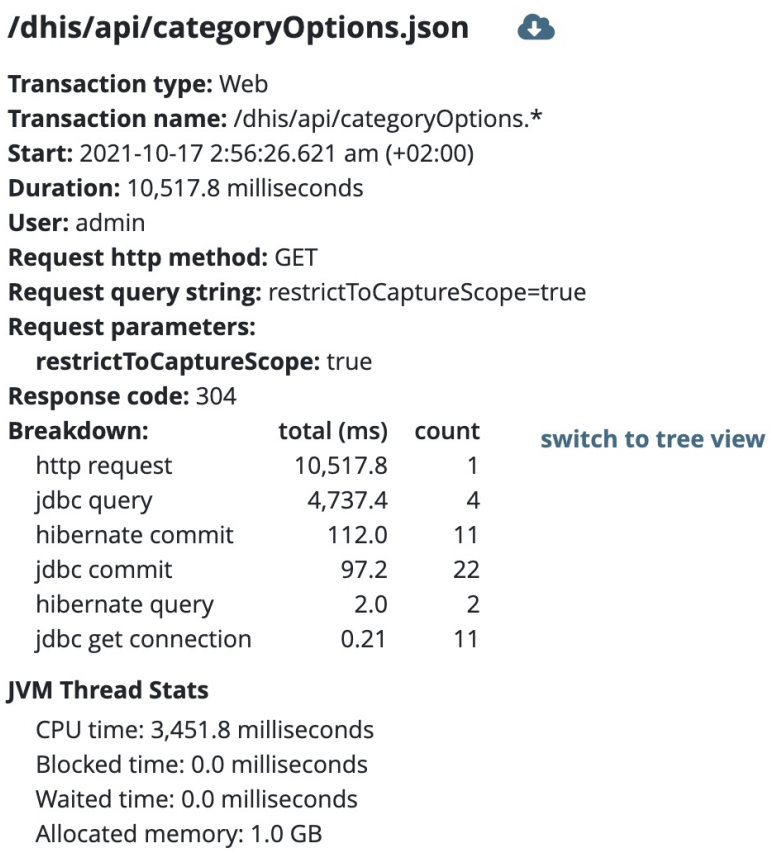

Figure 12: Glowroot Slow Trace showing 1GB Memory allocated for a single API request

established, the session has begun. Once the session ends the connection is destroyed. Sessions are designed to be long-lived. This means that the application connects once, performs many requests, and eventually disconnects when the application is being shut down or if the connection is no more needed. There is an overhead for connection creation. Hence connecting and disconnecting repeatedly is a bad practice as the overhead of connection creation and disconnection causes a performance bottleneck. This is where Connection Pooling is required.

On investigating the Rwanda DHIS2 Covid Vaccination instance for a high CPU consumption issue, with the help of the PostgreSQL database logs, we confirmed that the application was creating and destroying a connection for a specific HTTP Request. This HTTP Request was being invoked by all the users multiple times in a short period which caused the CPU to struggle. The CPU time needed to create and destroy a connection for every single request could have been saved if proper connection pools were utilized. Connection pools allow pre-connected sessions to be quickly served when needed and save the overhead of establishing or terminating the connection unnecessarily.

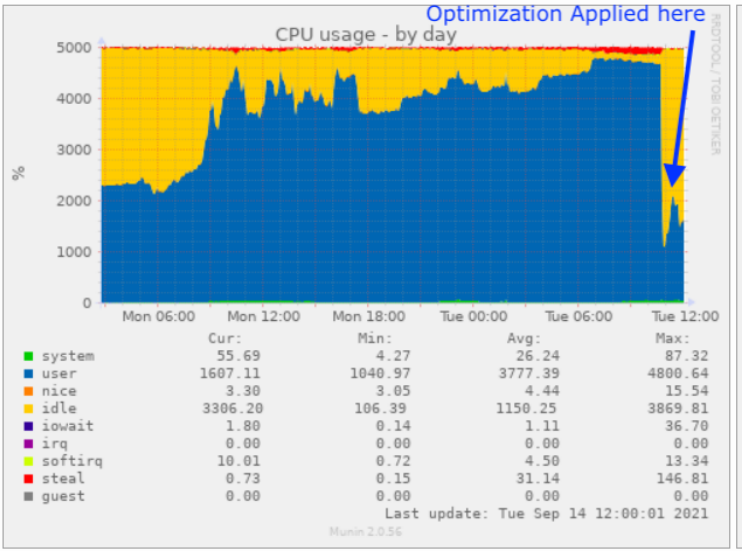

Figure 13: Munin dashboard showing CPU struggling due to sub-optimal resource utilization

Figure 13 shows a screenshot of the Munin CPU monitoring dashboard of Nigeria DHIS2 Covid Vaccination tracking instance. The screenshot also shows the effect of the optimization. It can be observed from the figure 13 that right before the optimization was deployed, the CPU usage was very high for an extended duration in the peak hours. Inefficient resource management caused unnecessary pressure on the CPU.

6 Optimizations and Results

The objective of any optimization is to attain a better performance of the whole system. It might be response time or throughput or a balance of both. Optimizations that have no impact on overall performance are not worth the effort. Optimizations have to be SMART (specific, measurable, achievable, results-based, and time-bound)[12]. The table 4 is inspired by SMART goal framework for optimization goals.

Goal	**DHIS2 Example**
Specific	Registration of TEI should be completed before the defined threshold time.
Measurable	Response time for search tracked entity instances by unique attributes query should not exceed 5 seconds.
Achievable	When data volume grows, the query response time should not grow more than logarithmically
Result-based	Event data value updates should not cause any database locking
Time-bound	We should release an optimized event data value update flow in the patch released next month

Table 4: SMART goals for DHIS2 optimizations

Optimization may involve optimizing the software development processes as well. An example is when developers seek further clarification of the business requirement and question the status quo. Such discussions can lead the way to create more declarative and performant queries and make changes in the application to better optimize it. On the application side, even if a single query takes less than 0.1 seconds if the API takes 10 seconds to respond, then there may be too many small queries being executed by the application on the database to serve that request.

The life of an application does not end after release in production, and optimization is a continuous process. We should continually keep an eye on

the system performance, not only on the execution times but on database volume trends and access patterns. A query may be very performant in the beginning, but query execution time may change because data volume increased or the data distribution changed or execution frequency increased. In addition, there will also be new indexes and other improvements in each new PostgreSQL release, and some of them may be so significant that they prompt further optimizations.

In the remaining part of this section, I elaborate on the optimizations done for the bottlenecks identified in section 5.2. Both the Application side optimizations and Database side optimizations are explained separately in the corresponding subsections of this chapter.

6.1 Database Optimizations

The best approach when trying to optimize database queries is to *Think like a database!* [12]. Observing the query from the point of view of a database engine, and imagining what it has to do to execute that query. The aim has to be to avoid imposing suboptimal execution plans.

Since two queries yielding the same result may be executed differently, utilizing different resources and taking a different amount of time, optimization and thinking like a database are core parts of SQL development and optimization. The database engine interprets SQL queries by parsing them into a logical plan, transforming the results, choosing algorithms to implement the logical plan, and finally executing the chosen algorithms.

When queries are not written declaratively, the original purpose of a query might not be evident. Developers writing the database queries should be well aware of the business requirement. Asking more questions to understand what exactly is needed and what is not is perhaps the first and the most critical optimization step. Some of those answers will help to transform a non-performant query into a performant one. An SQL query cannot be optimized in isolation, outside the context of its purpose and the environment in which it is executed.

6.1.1 Query Rewriting

A poorly constructed query, in other words, an in-efficient query, can be rewritten to satisfy the same requirement but better optimized for performance. The main points to remember when rewriting an in-efficient query are the following.

- Avoid fetching whole tables unless it is needed. Pagination should be enabled by default and only a subset of records should be fetched at a time.
- When constructing a complex query, we should make sure the query is declarative enough for the optimizer to find the best execution path. It is beneficial to start with adding all the clauses and conditionals that are responsible for narrowing or limiting the results. This means all the filtering predicates are applied early to limit the result size. This way, the subsequent left joins will have fewer records to join with.
- Avoid unnecessary usage of computed columns and sorting based on computed columns. On large tables, the columns have to be computed for every row by the database engine. Unless the computed column is indexed, the computation happens every time the query is executed.

In section 5.1.1, we have seen the inefficient query in listing 1 that created a bottleneck in the Sri Lanka Covid Vaccination Tracker instance. Keeping the above points in mind we rewrote the query. The new query is as shown in listing 7. The query is trimmed for brevity.

```
1  SELECT instance, ... ,inactive,
2  String_agg(TEA.uid  || ':' || TEAV.value, ';')
3  FROM    (SELECT ...
4           FROM   trackedentityinstance TEI
5           INNER JOIN trackedentityattributevalue "eYViMjtiWRA"
6           ON .. AND "eYViMjtiWRA".trackedentityattributeid = 17593
7           AND Lower("eYViMjtiWRA".value) LIKE '%0000000000%'
8           INNER JOIN trackedentityprogramowner PO
9           ON .. AND PO.programid = 17609
10          INNER JOIN organisationunit OU
11          ON .. AND ( OU.path LIKE '/GYBZ1og9bk7%' )
12          WHERE  TEI.trackedentitytypeid IN ( 17581 )
13          AND TEI.deleted IS FALSE
14          AND EXISTS (SELECT PI.trackedentityinstanceid
15                      FROM   programinstance PI
16                      WHERE  PI.trackedentityinstanceid =
17                       TEI.trackedentityinstanceid
18                       AND PI.programid = 17609
19                       AND PI.deleted IS FALSE)
20          ORDER  BY TEI.trackedentityinstanceid ASC
21          LIMIT  11 offset 0) TEI
22        LEFT JOIN trackedentitytype TET ON ..
23        LEFT JOIN trackedentityattributevalue TEAV
24        ON .. AND TEAV.trackedentityattributeid IN (
25          17593, 17621, 357636, 357642,
26            17585, 357663, 17586, 17587,
27             357648, 357649, 17592, 50842561,
28               24772, 25029, 29959, 54166040,
29                54166531, 17633, 50109767, 26234 )
30        LEFT JOIN trackedentityattribute TEA ON ..
31 GROUP  BY ..
32 ORDER  BY TEI.trackedentityinstanceid ASC
```

Listing 7: Optimized Rewritten Query

The main optimizations in the optimized query 7 are

- An inner subquery (line 3 to 21) is generated which limits the total number of results as per the pagination rules. Pagination is also enabled by default. This gives a minor boost in performance as there is an intermediate inner joined table that is restricted with *limit* and *offset*. This intermediate inner joined table is more scalable as the total size of the raw tables has a lesser impact. Whereas, in the in-efficient query 1, all of the left joins were done on the full table.
- All the chaining of left joins that existed in the inefficient query 1 has been replaced with an aggregate function *String_agg*. This eliminates the scalability problem when having a large number of *trackedentityattributes* in the system. No matter the number of attributes, there is

always only a single left join to the *trackedentityattributevalue* table as shown in line numbers 23 to 30 in the optimized query 7. This provides a significant performance boost and also ensures an overall predictable cost irrespective of the number of attributes in the instance.

- Unnecessary computed column usage has been removed. The default order is switched to the primary key *trackedentityinstanceid* column of the *trackedentityinstance* table. This ensures that the default ordering, if not explicitly requested, will always be based on the most performant primary key column that will already be indexed. This improvement was only possible when more questions were asked about the necessity of sorting by enrollment status which was present in the unoptimized query. The requirement was then updated in favor of achieving better performance.

String_agg is an aggregate function that concatenates a list of inputs with the specified delimiter. In line 2 of the above optimized query 7, the delimiter is semi-colon ";" and the list is a concatenated string of *TEA.uid* and *TEAV.value*. Using this aggregate function, all the relevant attributes for a single tracked entity instance are concatenated and returned as a single cell value. The application logic then splits this to get the individual values. This enables the query to scale better by avoiding chaining of left joins with the same table.

The query execution time for the in-efficient query in listing 1 was on an average **10 to 15 seconds**. The query was rewritten to make it more declarative as shown in listing 7. This optimized query allowed the query optimizer to choose a better execution path while satisfying the same requirement. The query execution time for the optimized query was on an average **190ms**. The figures are directly obtained from Sri Lanka's DHIS2 Covid Vaccination Tracker Instance.

6.1.2 Indexing

Queries can be categorized as Short queries and Long queries. It is important to distinguish between these categories as the optimization technique depends on the type of query. A query is short when the number of rows needed to compute its output is small, no matter how large the involved tables are. Short queries may read every row from small tables but read only a small percentage of rows from large tables. Whereas a query is considered long when query selectivity is high for at least one of the large tables; that is, almost all rows contribute to the output, even when the output size is small [12]. DHIS2 Tracker is mostly an OLTP (Online Transaction Processing) System and hence mainly has Short queries.

Short queries require indexes for faster execution. If there is no index to support a highly restrictive query as seen in section 5.1.2, one needs to be created. The best performance possible for a short query occurs when the most restrictive indexes (i.e., indexes with the lowest selectivity) are used. Selectivity is the ratio of rows that are retained (after a filter) to the

total rows in the stored table. Higher selectivity means, even after a filter operation, most rows are retained. Lower selectivity means, after a filter operation, most rows are eliminated. To create correct indexes, we must know about the different types of indexes along with their intended usage for maximum benefits. In this section, I explain the type of indexes that we used to optimize the identified bottlenecks in section 5.1.2. I also briefly explore alternative options and why those were not ultimately used.

The standard index type is the B-tree, where the B stands for balanced. A balanced tree is one where the amount of data on the left and right sides of each split is kept even so that the amount of levels you have to descend to reach any individual row is approximately equal. The B-tree can be used to find a single value or to scan a range, searching for key values that are greater than, less than, and/or equal to some value. They also work fine on both numeric and text data. They are suited for range querying and comparisons (greater than, between, less than, and equality)[20].

A hash index uses a hash function to calculate the address of an index block containing an index key. This type of index has better performance than a B-tree index for equality conditions. The hash index type can be useful in cases where you are only doing equality (not range) searching on an index, and you don't allow null values in it. The advantages to using a hash index instead of a B-tree are usually small [12, 20].

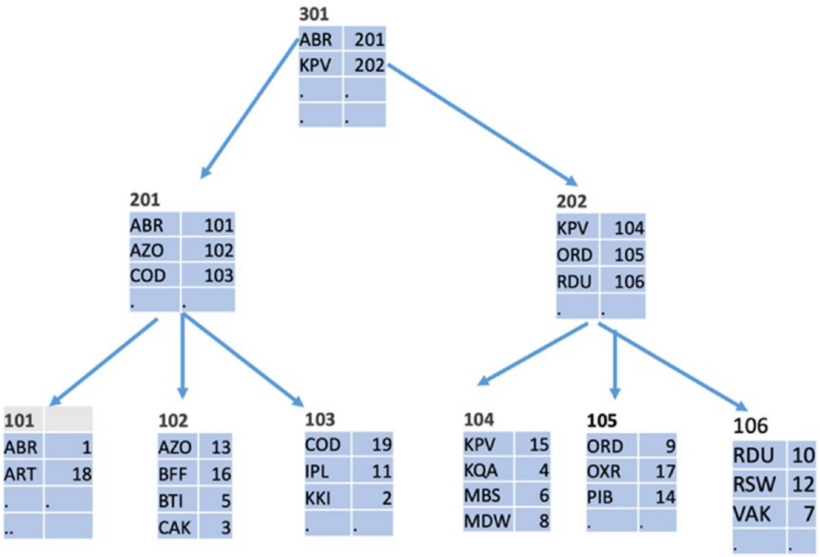

Figure 14: B-Tree index structure

The leaf nodes (shown in the bottom row in Figure 14) contain index records. These index records contain an index key and a pointer to a table

row. Non-leaf nodes (located at all levels except the bottom) contain records that consist of the smallest key in a block located at the next level and a pointer to this block. All records in all blocks are ordered, and at least half of the block capacity is used in every block.

Any search for a key K starts from the root node of the B-tree. During the block lookup, the largest key P not exceeding K is found, and then the search continues in the block pointed to by the pointer associated with P until the leaf node is reached, where a pointer refers to table rows. The number of accessed nodes is equal to the depth of the tree. The key K is not necessarily stored in the index, but the search finds either the key or the position where it could be located. B-trees also support range search (expressed as a between operation in SQL). As soon as the lower end of the range is located, all index keys in the range are obtained with a sequential scan of leaf nodes until the upper end of the range is reached. A scan of leaf nodes is also needed to obtain all pointers if the index is not unique (i.e., an index value may correspond to more than one row).

B-trees are a sensible default in PostgreSQL. No lookup algorithm can find an index key among N different keys faster than in log N time (measured in CPU instructions). This performance is achieved with a binary search on an ordered list or with binary trees. However, the cost of updates (such as insertions of new keys) can be very high for both ordered lists and binary trees: insertion of a single record can cause complete restructuring. This makes both structures unusable for external storage. In contrast, B-trees can be modified without significant overhead. When a record is inserted, the restructuring is limited to one block. If the block capacity is exceeded, then the block is split into two blocks, and the update is propagated to upper levels. In the worst case, the number of modified blocks cannot exceed the depth of the tree. In PostgreSQL, a B-tree index can be created for any ordinal data type; that is, for any two distinct values of the data type, one value is less than the other. This includes user-defined types.

It may also be useful to have multiple b-tree indexes in the same table depending on querying and filtering patterns. A bitmap is an auxiliary data structure that is used internally in PostgreSQL for several different purposes. Bitmaps can be considered a kind of index: they are built to facilitate access to other data structures containing several data blocks. Typically, bitmaps are used to compactly represent the properties of table data. Usually, a bitmap contains one bit for each block (8192 bytes). The value of the bit is 1 if the block has a property and 0 if it hasn't. Figure 15 shows how bitmaps are used to access data through multiple indexes[12].

The database engine starts by scanning both indexes and building a bitmap for each that indicates which data blocks contain table rows with requested values. These bitmaps are shown in the rows labeled Index 1 and Index 2. As soon as these bitmaps are created, the engine performs a

Block#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Index 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1
Index 2	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0	1
Logical AND ⇓																
	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1

Figure 15: Using bitmaps for table access through multiple indexes

bitwise logical AND operation to find which blocks contain requested values for both selection criteria. Finally, data blocks corresponding to 1s in the final bitmap are scanned. This means that blocks that satisfy only one of the two criteria within a logical AND never have to be accessed. Note that requested values may reside in different rows in the same block. The bitmap ensures that relevant rows will not be missed, but does not guarantee that all scanned blocks contain a relevant row. Bitmaps are very compact; however, bitmaps may occupy several blocks for very large tables. To speed up processing such bitmaps, PostgreSQL builds a hierarchical structure: an upper level indicates the blocks of the lower-level bitmap to be processed[12].

Now that some of the index types and structure has been explained, we proceed to resolve the identified bottlenecks from section 5.1.2. The first issue shown in figure 4 was that the filter condition was very restrictive but no index existed. We created an index for it as it is a very good candidate for indexing in this scenario. However, note that there will not be any benefit if the column *value* is indexed. Since the query uses column transformation to convert the value into lowercase with **lower(value)**. This meant that we had to create a functional index on lower(value). When a functional index is built, PostgreSQL applies the function to the values of the column (or columns) and then places these values in the B-tree. Similar to a regular B-tree index, where the nodes contain the values of the column, in a functional index, a node contains the value of the function. In this case, the function is lower(). We created the indexes using the SQL statements shown in listing 8.

```
1 ALTER TABLE trackedentityattributevalue
2   ALTER COLUMN VALUE SET DATA TYPE VARCHAR(1200);
3
4 CREATE INDEX in_trackedentity_attribute_value
5     ON trackedentityattributevalue
6 USING btree (trackedentityattributeid, lower(value));
```

Listing 8: Creating a functional index on lower(value)

Once the indexes were created we ran the ANALYZE command so that PostgreSQL can collect updated data statistics which can be used by the optimizer to find the best execution plan. On analyzing the query plan after index creation, for query in listing 2, we saw that the new functional index was used by the query planner as shown in figure 16. The most restrictive criteria were processed first with the help of the index and thereby optimizing the performance of this query.

The query execution time for the query in listing 2 was around **3536ms**. After analyzing the bottleneck in the query plan and adding an index as shown in listing 8, the query execution time for the same query in the same environment became **5ms**. The figures are directly obtained from Nigeria's DHIS2 Covid Vaccination Tracker Instance.

The second issue, which was reported by the Rwanda DHIS2 tracker instance, was the bottleneck for the query in listing 3. The identified bottleneck is shown with the corresponding query plan node in figure 6. It was observed again that the filter condition was very restrictive removing over 99% of the parsed rows but no index existed to help the query executor. In this case, a B-tree index will not suffice. This is because of the usage of the **like** operator along with double-ended wildcards (%). This is where we experimented with trigram indexes as it was a good fit for this use case.

Regular indexes are optimized for the case where a row has a single key-value associated with it so that the mapping between rows and keys is generally simple. The Generalized Inverted Index (GIN) is useful for different sorts of organizations. GIN stores a list of keys with what's called a posting list of rows, each of which contains that key. A row can appear on the posting list of multiple keys too. GIN is useful for indexing array values, which allows operations such as searching for all rows where the array column contains a particular value or has some elements in common with an array being matched against. It's also one of the ways to implement full-text search [20].

Indexes do not have to cover the entirety of a table. Smaller and targeted indexes can be created that satisfy a particular WHERE clause, and the planner will use those when appropriate. Such index is called a Partial Index.

B-tree index cannot be used for searches like '%aaaa' (starting wildcard) or searches like '%aaaa%' (double-ended wildcard). To index for this type of search, the pg_trgm extension is needed. This extension also indexes ilike (insensitive like) searches. Trigram indexes are GIN indexes. To create a composite multi-column trigram index on a set of columns having at least one primitive column type (bigint in our case) then another extension *btree_gin* is also required. We created trigram gin indexes to solve the performance bottleneck with the *like* search filter as shown in figure 6.

```
1 CREATE EXTENSION pg_trgm;
2 CREATE EXTENSION btree_gin;
3
4 CREATE INDEX in_gin_teavalue_3461
5 ON trackedentityattributevalue USING
6 gin (trackedentityinstanceid, lower(value) gin_trgm_ops)
7 WHERE trackedentityattributeid = 3461;
```

Listing 9: Creating a partial trigram gin index on lower(value)

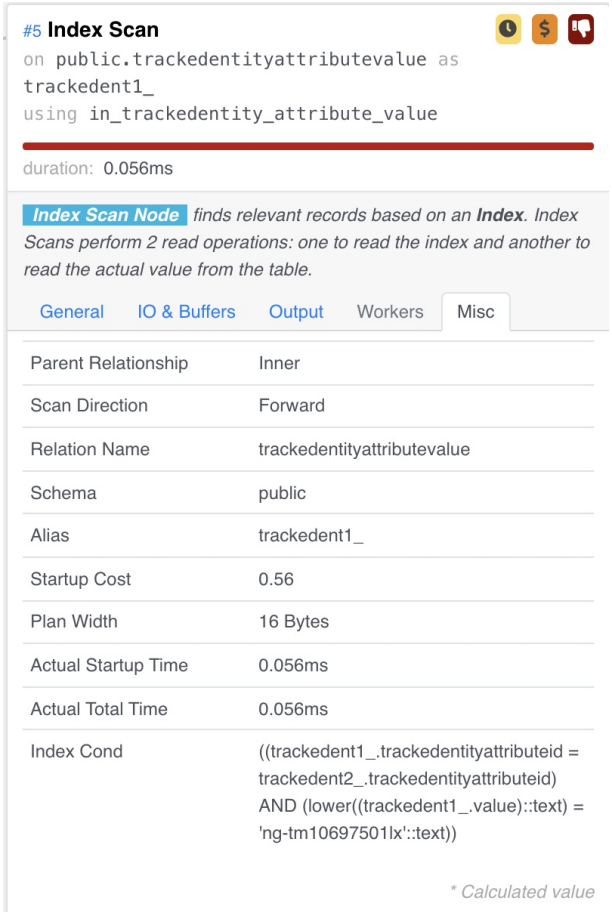

Figure 16: Optimized Query plan node in Rwanda showing the new index being used effectively.

Since both of the extensions, *pg_trgm* and *btree_gin* are already part of the PostgreSQL installation bundle, we created the extensions right away in

Rwanda DHIS2 Covid Vaccination instance. We then created 3 specific partial multi-column trigram indexes on the *First Name* attribute, *Surname* attribute and *Phone Number* attribute in the table *trackedentityattributevalue*. Listing 9 shows the sql statements used to create the extensions and one of the 3 partial indexes.

Once the extensions and trigram indexes were created, we analyzed the query plan node again. The same query in listing 3 was used to obtain the execution plan and the relevant query plan node is shown in figure 16. We confirmed that the new trigram index was used very effectively by the query planner. Since the index had low selectivity, the speed of look-up increased multiple folds.

The query execution time for the query in listing 3 was around **1281ms**. After analyzing the bottleneck in the query plan and adding an index as shown in listing 9, the query execution time for the same query in the same environment decreased to **13ms**. The figures are directly obtained from Rwanda's DHIS2 Covid Vaccination Tracker Instance.

We resolved a very similar bottleneck with trigram indexes in the Nigeria DHIS2 Covid Vaccination instance where their QR code searches were very slow. We created a specific trigram index on the specific jsonb attribute that was searched in the table *programstageinstance*. Listing 10 shows the index creation statements.

```
1 CREATE INDEX in_gin_psi_edv_64527_233047
2 ON programstageinstance USING
3 gin(lower(eventdatavalues #>> '{LavUrktwH5D,␣value}')
4 gin_trgm_ops);
```

Listing 10: Creating a trigram gin index on a jsonb column for a specific attribute

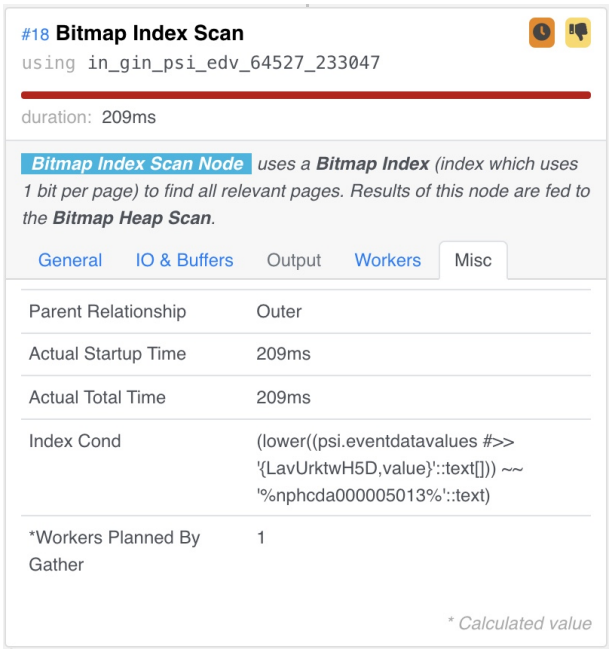

Figure 17: Optimized Query plan node in Nigeria showing the new index being used effectively.

We obtained the execution plan for the same query after creating the trigram index. The relevant query plan node is shown in figure 16. We confirmed that the new trigram index was used very effectively by the query planner. Since the index had low selectivity, the speed of look-up increased multiple folds. The query execution time for the query in listing 4 was around **8.5s**. After analyzing the bottleneck in the query plan and adding an index as shown in listing 10, the query execution time for the same query in the same environment decreased to **210ms**. The figures are directly obtained from Nigeria's DHIS2 Covid Vaccination Tracker Instance.

6.1.3 Summary of Results

The table 5 shows a summary of the results observed when resolving database bottlenecks and applying their corresponding optimizations.

Bottleneck	**Optimization**	**Improvement**
In-Efficient Query (Listing 1)	Rewritten Declarative Query (Listing 7)	Query Execution time decreased from 12500 ms to 190 ms. **6500% increase in performance observed in Sri Lanka DHIS2 Covid Vaccination Instance**
Attribute Uniqueness Check (Listing 2)	Added Functional Index (Listing 8) .	Query Execution time decreased from 3536 ms to 5 ms. **70600% increase in performance observed in Nigeria DHIS2 Covid Vaccination Instance**
Search with *like* operator (Listing 3 and Listing 4)	Added Trigram Gin Index (Listing 9 and Listing 10)	Query Execution time in Rwanda decreased from 1281 ms to 13 ms. Query Execution time in Nigeria decreased from 8511 ms to 209 ms. **9753% increase in performance observed in Rwanda DHIS2 Covid Vaccination Instance 3972% increase in performance observed in Nigeria DHIS2 Covid Vaccination Instance**

Table 5: Database Optimization Results

6.2 Application Optimizations

In this section, I explain the optimizations done on the bottlenecks identified in section 5.2. This mainly deals with optimizing the processes and code flow within the application.

6.2.1 Efficient API Access Pattern

As explained in section 5.2.1, the in-efficient pattern of accessing API is potentially a severe bottleneck. The bottleneck identified in the Sri Lanka DHIS2 Covid Vaccination instance is a real example of the Shopping list problem. Instead of sending an HTTP PUT request for every input field separately, the front-end App was rewritten to wait till all input fields were populated. Only when the user clicks on a ”Save and Complete” button, the HTTP PUT Request to update all the fields together are sent. This eliminated the unnecessary round-trips from the Client-side (Front-end app) to the Server side.

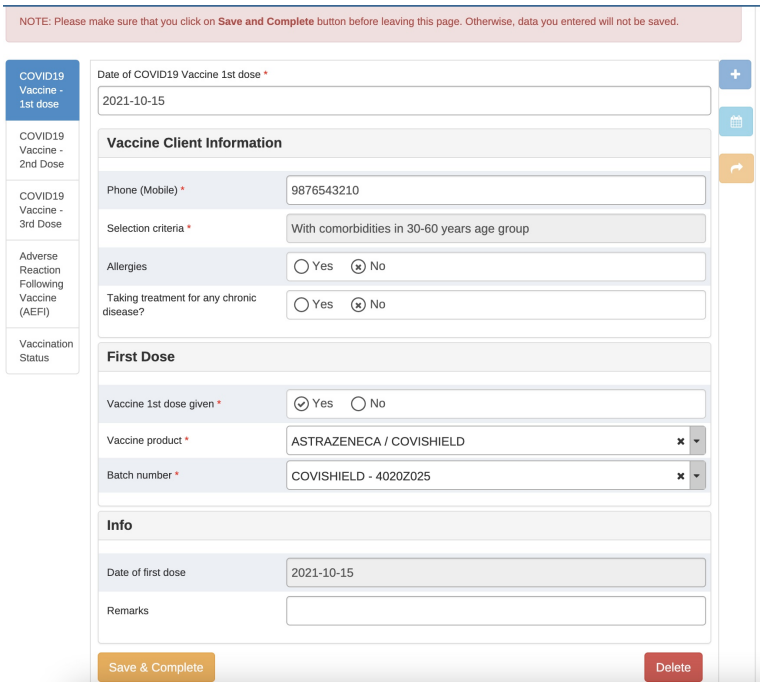

Figure 18: Rewritten App user interface for collective event data values updation.

Changing the API access pattern to collectively update the event data values instead of doing it per input field decreased the number of round-trips to the backend by a factor of the number of input fields. In Sri-Lanka DHIS2 Covid Vaccination instance, it decreased by a factor of al-

most 10. The figure 18 shows the rewritten front-end app rendered in a client browser. The extra piece of information highlighted in red indicates the change in behavior so that users will not lose data. The button label was changed to ”Save and Complete” instead of ”Complete” to indicate the collective update action happening in the end.

```
URL: /dhis/api/30/events/*
Request Method: PUT
Request Payload:
{
    "dataValues": [{
            "value": "true",
            "dataElement": "sEgbpR5sGP6"
    }, {
            "value": "criteria4",
            "dataElement": "CUTNLPqkmNn"
    }, {
            "value": "product1",
            "dataElement": "J1HZdZNWqMb"
    }, {
            "value": "batch1",
            "dataElement": "xv7LXLV8RLT"
    }, {
            "value": "9876543210",
            "dataElement": "aqlnlgpRDdI"
    }, {
            "value": "false",
            "dataElement": "TYn17QNtyNV"
    }, {
            "value": "false",
            "dataElement": "Z3dcWqQz6e6"
    }, {
            "value": "2021-10-15",
            "dataElement": "WcMFAfaJ46U"
    }],
    "event": "qOHWS1SwApY",
    "program": "aLZQ5fSVdQc",
    "programStage": "fZ3diyIwzDF",
    "orgUnit": "DiszpKrYNg8",
    "trackedEntityInstance": "iWxmynYsPGr",
    "status": "COMPLETED",
    "eventDate": "2021-10-15",
    "completedDate": "2021-10-15"
}
```

Listing 11: Request Payload example for Event Update API that collectively updates all event data values

The listing 11 shows the details of the event update API that collectively updates all event data values together. An example payload is also shown in the listing. The payload has eight event data values contained in it and all eight will be updated in this specific API. The unoptimized version would use eight separate API requests to update the same number of event data values. The figure 19 shows the response timing information as recorded by the Google Chrome browser. The TTFB (Time to First Byte) is the time it took for the first byte of the response to reach the browser.

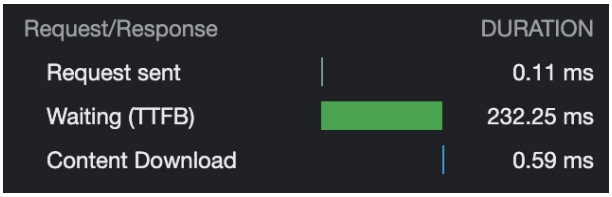

Figure 19: Response timing of Event Update API that collectively updates all event data values

6.2.2 Avoiding ORM pitfalls

As observed in section 5.2.2, the Hibernate ORM was the root cause of the identified bottleneck in the DHIS2 PEPFAR DATIM instance. This is mainly due to developers overlooking the fact that any method invocation that retrieves the associated list of hibernate mapped entities, will trigger a new database query. On large-scale instances, this creates a significant bottleneck. As reported by the system administrators of the PEPFAR DATIM instance, their implementation got shut down due to OutOfMemory errors.

This first step to optimizing this bottleneck was to not use Hibernate to fetch the data. The ORM mapped objects bring with them unnecessary complexities whereas in this case, the requirement was very straightforward. A new purpose-built API was created independently which fetches the required data for the front-end client. The *categoryOptions* along with its associated OrganisationUnit identifiers (Unique ID) that are relevant to the logged-in user are fetched using Spring JDBCTemplate. Since the association is only a list of strings, the memory allocation within the application reduces drastically. Spring JDBCTemplate uses native queries provided by the developer and therefore the developer can create performant queries tailored for the requirement at hand. The figure 20 shows the slow trace recorded by Glowroot. It can be seen that the memory allocation for the same requirement is considerably lower here as compared to the unoptimized slow trace in figure 12.

Figure 20: Lower Memory allocation for the purpose-built API as recorded by Glowroot

This optimization was effective for both the Bangladesh DHIS2 Immunization instance as well as the PEPFAR Datim instance. The performance was acceptable irrespective of how the users and organisation units were configured.

6.2.3 Connection Pooling

As illustrated in section 5.2.3, it is important to utilize and manage system resources efficiently. The issue in the Rwanda DHIS2 instance was that connections to the database were continuously being created and destroyed for every HTTP request. This caused the CPU to struggle and have unnecessarily high CPU utilization.

As a temporary relief, the instance was provided with double the amount of CPUs while the investigation was being done. Providing more CPUs is not considered as a performance optimization. However, the additional CPUs did relieve the stress in the system until a proper fix was identified.

DHIS2 application has a Connection Pooling mechanism for database connections. The Connection pooling library c3p0 had already been configured. Even though there was a connection pool configured, the PostgreSQL logs of the Rwanda DHIS2 Covid Vaccination instance confirmed that there was an area in the application that was creating ad-hoc connections into the database without using the connection pool.

Digging deeper into the issue, we learned that an external library used by the DHIS2 core application was creating ad-hoc connections and destroying them on demand. The external library had to be modified to accept the connections from the connection pool, instead of creating its connections to the database. This saved significant CPU time and associated bottleneck.

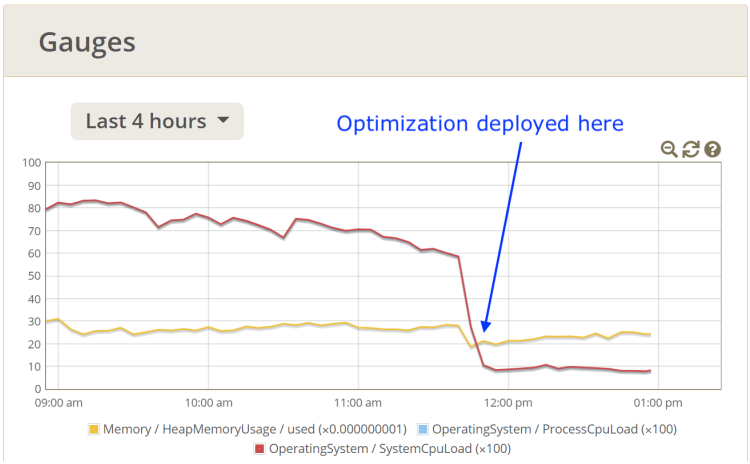

Figure 21: Glowroot Guage chart showing the effect of the optimization on CPU load

Figure 13 and figure 21 both show CPU usage and load monitored by Munin and Glowroot. The effect of the optimization is seen on both screenshots. This also shows the importance of connection pooling. The dashboard is from the production instance of Nigeria's DHIS2 Covid Vaccination Tracker instance.

6.2.4 Summary of Results

The table 6 shows a summary of the results after resolving application bottlenecks and applying their corresponding optimizations.

Bottleneck	**Optimization**	**Improvement**
In-Efficient API Access (Section 5.2.1)	Rewrite Front-End App to reduce API round-trips	Total API Request time drastically reduced by more than 80%. Time taken for individual API was averaging 300ms. Total Event Update API Request time before optimization was 2.1s. This decreased to 300ms. **600% increase in performance observed in Sri Lanka DHIS2 Covid Vaccination Instance for Event updates**
High Memory allocation by ORM object graphs	Purpose-built API created	The memory allocation for the same request was reduced considerably. **85% decrease in memory allocation observed in simulated load on Performance Test Environment**
In-efficient connection creations	Enforce connection pool usage	This optimization is hard to quantify. However, the figures 13 and 21 shows the effect of the optimization. **Approximately 60% decrease in CPU usage observed in Nigeria DHIS2 Covid Vaccination instance**

Table 6: Application Optimization Results

7 Discussion

Through this thesis, we understood some of the performance issues faced by large-scale Information Systems. By studying the performance issues on large-scale DHIS2 implementations across the globe, we understood how these bottlenecks affected the day-to-day work of users of the system. Health workers and officials are all affected deeply by performance bottlenecks. We resolved some of the performance bottlenecks using optimization techniques available in the existing literature. We also empirically evaluated how the optimization changes have had an impact on performance. These optimizations applied to DHIS2 will be generalized in this chapter. Generalized optimizations make them relevant to any large-scale enterprise application that interacts with a relational database.

The bottlenecks faced by large-scale DHIS2 implementations are categorized into two, Data Source Layer and Application Layer. Existing literature on optimization techniques was extensively studied and carefully applied to the real-world problem at hand, the DHIS2 Tracker country implementations. The results of the optimizations demonstrate that the bottlenecks have been resolved for the identified issues.

Most of the optimization techniques applied were derived from existing literature. Dombrovskaya et al. in their work [12] covers the most common performance issues faced by applications interacting with the PostgreSQL database. The work also covers common pitfalls in ORM frameworks, Application development, and database interactions. Although the examples were generic, this thesis extends their work by identifying the bottlenecks and applying the suggested optimizations on a production-grade large-scale DHIS2 implementation. The results obtained support the optimizations. There was a need for application downtime to experiment with some of the optimizations. But the downtime was always worth the performance improvement gained once the optimizations were done. The results contribute to a clearer understanding of the impact of these optimizations. We observed an increase in throughput, a decrease in response time, and better resource utilization.

The generalized learnings during the resolution of database bottlenecks were:

1. SQL queries should be as declarative as possible. This allows the query optimizer to do its job of optimizing the execution path. Bottlenecks will be created if a sub-optimal execution path is enforced.

2. Scalability should be considered when dynamically building SQL queries based on data. The chaining of left joins depending on some aspect of data is not ideal. The default value of *join_collapse_limit* PostgreSQL configuration parameter is 8. This parameter caps the number of tables in a join that will be processed by the cost-based optimizer. If the number of tables in a join is eight or fewer, the optimizer will perform

a selection of candidate plans, compare plans, and choose the best one [12]. But if the number of tables is nine or more, it will simply execute the joins in the order seen in the SQL and hence enforcing a sub-optimal plan.

3. Selectivity should be considered when creating indexes. The most restrictive filter with a low selectivity is an ideal candidate for an index.

4. Query access pattern should be monitored to find out appropriate index types to create. Appropriate Index types have to be used. For example, the Trigram index type should be used for like comparisons, and the B-tree index type should be used for range and equality comparisons. GIN index type for full-text searches.

The generalized learnings during the resolution of application bottlenecks were

1. Avoid unnecessary round trips from the client-side (Front-end App). Multiple smaller APIs performing related operations for a user flow could be merged into a single API doing the same tasks.

2. Avoid unnecessary round trips to the database from the application. These are a common performance pitfall in any ORM-based application. Hibernate n+1 query problem is an example of such a pitfall.

3. Inspect the queries generated by ORM under the hood. ORM can generate sub-optimal queries.

4. Keep an eye on memory allocation per request. These give an idea of potential resource leakage or even a sub-optimal in-memory computation that will not scale well. Although request processing time may be low, if it consumes high resources, it will still be a bottleneck.

Different factors expose a bottleneck. It can either be due to a unique data access pattern or a unique data configuration or data volume of specific entities. There are cases where a bottleneck in one instance of a system is not immediately visible in another instance of a similar system. For example, the application bottleneck in section 5.2.2 was resolved by a quick optimization fix, and the bottleneck seemed to be resolved in the Bangladesh DHIS2 Measles Immunization instance. However, the same optimization created a new application bottleneck in another large-scale DHIS2 implementation, the PEPFAR DATIM instance. This was because of the difference in the implementation context and how the metadata and users were set up. The PEPFAR DATIM instance had a large number of users having access to the Root Organisation Unit along with a very large number of organizational units in the hierarchy. This was different from how Bangladesh had configured their system with the majority of the users only having access to just a handful of organisation units. In any information system, such configurations, data access patterns, data volume, and data access frequency are significant factors that influence performance characteristics. Covid Vaccination tracking was the concrete context for three of

the five implementations involved in my case study. The optimizations applied were equally relevant to all these 3 Covid related implementations due to their similarity in configuration and data access patterns. The Covid Vaccination tracking implementation exposed several bottlenecks due to its magnitude of data entry and frequency of data access. An optimization done to resolve a bottleneck should not create an equal or greater bottleneck in another part of the system. Such changes are not optimizations and lead to further degradation of the system.

One of the optimizations to resolve the bottleneck in 6.1.2 was to create partial trigram indexes (Listing 9). However, a limitation of the trigram index is that it is ineffective for substrings shorter than three. For example, the pattern '%01%' requires a sequential search even if the trigram index is present. This behavior created an issue in Nigeria, where some end users were searching with a search text size of less than three. The same bottleneck resurfaced for which the trigram index was created in the first place. A fix in the Custom Frontend App maintained by HISP Nigeria was done to enforce a three-character search text rule. The change ensured that the client does not use the API if the search text is less than three letters.

Sri Lanka had reported a performance issue related to 5.2. On investigating the PostgreSQL logs, we observed database locks held for extended periods(30 seconds). However, on analyzing their infrastructure using Munin Monitoring tool [7], the disk latency was momentarily creeping up as high as 1 second. The snapshot from Munin is added in Appendix 26. An ideal disk latency should have been under 20ms, and even under 1ms for SSDs. Due to this poor disk latency, excessive locking was observed. Even though infrastructural performance issues are outside the scope of this thesis, it has to be noted that such infrastructural issues magnify the bottlenecks present in the application. Since these issues are outside the control of developers, it is important to resolve as many potential application/database bottlenecks as possible.

Some of the performance optimizations were done only on specific production instances by analyzing their data distribution and query patterns. The trigram indexes mentioned in listing 9 is one of them. These optimizations have not yet been included in the DHIS2 Core release. However, seeing the performance improvement the optimizations have led to, the DHIS2 core development team is planning to generalize the optimizations and include them as part of a future DHIS2 release. The same principle can be applied to any enterprise application that wants to improve searching based on text comparisons in PostgreSQL.

There were several challenges faced during the course of my thesis work. Most of the DHIS2 Tracker implementations involved in my case study are based in developing countries. For a developed country like USA or Norway, it was possible to vertically scale the system by adding more computing power on demand in case a performance bottleneck was identified. However, this was not always possible in developing countries where the

instance was deployed and maintained in Government or Parastatal data centers. Resources were often limited and it was not possible to increase the computing resources on demand. This further signifies the scope of my thesis where we had to resolve the bottlenecks within the confines of the infrastructural environment it was running in. Another challenge was, the system administrators in developing countries were often inexperienced in diagnosing complex performance problems. Fortunately, Glowroot provided a window into several DHIS2 implementations without opening up access to their sensitive data. The different metrics captured in Glowroot helped us in the analysis of database queries and API behaviors to resolve bottlenecks. It provided a vital bridge between the DHIS2 Core team and the different concrete implementations. Another challenge faced during my thesis work is the time-critical nature of the Covid Vaccine implementations. Unlike usual long-term HIS implementations where performance issues can eventually get fixed over time, Covid Vaccination implementations are different. Besides being large-scale, they are also time-critical. National vaccination campaigns are launched by country officials to achieve maximum vaccinations. These campaigns typically last a few weeks. When the system fails to perform, it is not merely an inconvenience. It threatens the viability of the whole campaign and needs to get fixed quickly. This determined the urgency of the optimization responses and fixes were required within hours or days and not weeks or months.

Apart from the challenges faced, there were optimization techniques that could not be applied due to various limitations. Covering index is a new feature that was introduced in PostgreSQL 11. This feature allows the query executor to use index-only scans instead of index-scans [18]. The non-indexed columns required by the select projection can be included in an existing b-tree index. This is different from multicolumn indexes with the same columns. Size and maintenance of covering indexes will be lower than multi-column indexes. Covering index could not be applied to DHIS2 version 2.34 or 2.35 as the minimum supported PostgreSQL version of DHIS2 version 2.34 and 2.35 is PostgreSQL 9.6. Due to this limitation, optimizations based on Covering indexes were not actively explored. Such PostgreSQL version requirement upgrades can only be made on a later major release version (like 2.38) with an explicit announcement.

Infrastructural and architectural decisions and changes were out of the scope of my thesis work. However, several optimization opportunities can be explored by modifying the application architecture. For example, a micro-service architecture can be used to modularize and scale parts of the application based on usage and demand. There are benefits to a micro-service architecture-based application compared to a Monolith application. DHIS2 is currently a Monolith but has the potential to be modularized into micro-services. Such micro-services can be deployed and scaled independently. Horizontal scaling is also a possibility for optimizing performance. However, it needs more computing resources to be able to host multiple nodes of the system and have a load balancer balancing the requests to the system. PEPFAR DATIM based in the US is one of the few implementations that

have horizontal scaling enabled. The developing countries cannot horizontally scale DHIS2 due to resource limitations. Therefore, this thesis work has had a significant impact on implementations in developing countries.

In the research work related to this thesis, most of the optimizations were reactively done. It means that once a large-scale DHIS2 implementation reports a performance issue, we then analyze the bottleneck and resolve it reactively. Existing literature indicates that there are several more possibilities to proactively optimize certain parts of the DHIS2 core application and its interaction with the PostgreSQL database. Such extensive optimization is also constrained by the amount of development time required in comparison with the urgency of the performance issue being a blocker in large-scale DHIS2 instances. Hence due to time limitations, such optimization experiments could not be done in this thesis.

One of the potential proactive optimization to avoid common pitfalls of the ORM framework was explained by Dombrovskaya[12]. The concept of NORM (No ORM) was introduced in their work. Traditional ORM works as shown in figure 22. The fact that ORM fails to represent complex objects present in the database forces the application to make multiple round trips to fetch data.

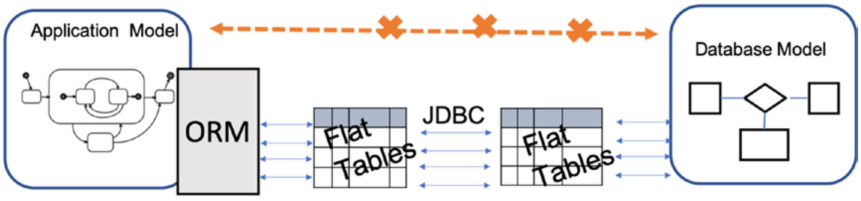

Figure 22: How ORM works

In the optimization suggestion with the NORM framework, the interaction between the application and the database can be summarized as:

1. The application serializes data into JSON format then converts it into an array of text strings and sends it to the database by calling a corresponding database function.
2. A database function parses the JSON that was passed as a parameter and executes whatever the function is supposed to do: either a search or data transformation.
3. The result set is converted to JSON (or rather an array of strings, which represents an array of JSON objects) and passed to the application, where it is deserialized and is ready to be consumed by the application.

The figure 23 shows how NORM helps the application interact with the database more effectively by reducing the round trips and fetching complex data records in one trip. Further studies would be needed to verify if NORM framework can further optimize DHIS2.

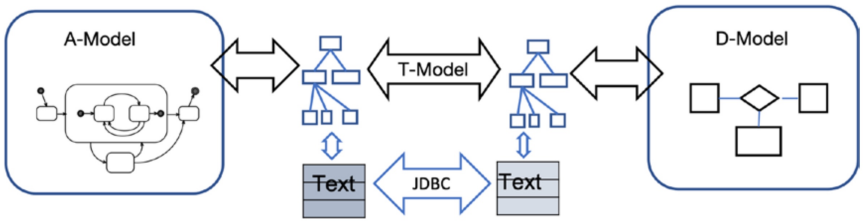

Figure 23: How NORM works

The NORM framework is said to be tried and tested on production enterprise applications with positive results. This can be a possible optimization path for Large Scale information systems that use ORM frameworks and face performance bottlenecks due to the common ORM pitfalls.

Even though the performance analysis was done on large-scale DHIS2 systems, the explanation of the bottlenecks and the optimizations are generalized so that they can be applied to any large-scale enterprise application that interacts with a relational database. I used large-scale DHIS2 implementations to illustrate the common performance bottlenecks that can occur in any large-scale Information System. I also explain how these bottlenecks can be optimized and resolved by taking examples from DHIS2 implementations.

This research has shown how important it is to integrate performance analysis into the development life-cycle of any large-scale enterprise application development. Over the course of my case study, there have been positive changes in the DHIS2 Core development process to include performance analysis and stress testing as part of the development life-cycle. This has helped in the early identification of bottlenecks and also helped to replicate the production bottlenecks in a performance test environment accessible to developers.

The important point is to always remember Amdahl's law. The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is used. One may increase the performance of one part of an API by a factor of 10, but if the application only spends 1% of its time in that code, then the overall improvement is reduced to only a factor of 0.1 (10 times 0.01). Therefore we must prioritize optimizing parts of a system that is used most often.

Performance Analysis is an ongoing process. Query execution time may change because of data volume increase or data distribution change or execution frequency increase. In addition, there will be new index types and other improvements in each new PostgreSQL release, and some of them may be so significant that they prompt rewriting original queries. Requirements may change forcing new features to be added to the application. These can also have a rippling effect on the overall performance of the system. No part of any system should be assumed to be optimized forever, and monitoring the system should happen continuously. This research work only covers some of the performance bottlenecks and is not an exhaustive attempt to resolve all bottlenecks.

8 Conclusion

This research aimed to identify performance issues in large-scale Information systems by performing a case study on multiple large-scale DHIS2 Tracker implementations. I illustrated how performance analysis was done and how bottlenecks were resolved using various optimization techniques. The results of the optimizations indicate that each of the optimizations is increasing the scalability of DHIS2. However, there are still more optimization opportunities available to improve scalability even further.

The impacts of these bottlenecks are significant. For DHIS2, users had to switch to less effective alternative ways to store information like paper-based records. Users were unable to meet their deadlines. Bottlenecks in large-scale systems render the system unusable. Passengers could not board their flights because the Covid Vaccine QR code searching was a bottleneck in the system. It affects the day-to-day lives and work of users. The viability of Covid Vaccination campaigns that last only a few weeks is threatened due to performance bottlenecks.

Several optimization techniques available from existing literature were applied to multiple large-scale production information systems. The impact of the optimizations was analyzed with experiments on simulated environments as well as real-world production environments. The results confirm significant performance improvements. System administrators of all the DHIS2 implementations I have worked with during my research have confirmed that the optimizations applied have saved their day and made their jobs easier. All the learnings mentioned in section 7 have been generalized and can be applied to any large-scale enterprise application.

Further research is needed to determine the effects of other optimization techniques available in the existing literature which were not explored in this thesis due to technical or time limitations. Future work is encouraged to investigate bottlenecks and apply optimizations proactively into DHIS2 or other large-scale Information Systems. For example, the application of the NORM framework in DHIS2 as mentioned in section 7. Similarly, covering indexes and other new features released in later PostgreSQL versions will warrant more optimization changes within DHIS2. Infrastructural and architectural decisions were outside the scope of this thesis. However, further research can also explore the optimization opportunities possible by enabling efficient horizontal scaling capability or using microservices architecture in an Information System.

I conclude my thesis with a word of caution that we cannot assume anything is optimized forever. Information systems will have to be monitored continuously for changes in queries, data volume, value distributions, data access patterns, and other characteristics that can interfere with performance. System administrators and developers should be vigilant. There is always room for more optimizations, both on the application side and the database side.

Bibliography

[1] Jørn Braa and Calle Hedberg. "The Struggle for District-Based Health Information Systems in South Africa". In: *The Information Society* 18.2 (2002), pp. 113–127. DOI: 10.1080/01972240290075048. eprint: https://doi.org/10.1080/01972240290075048. URL: https://doi.org/10.1080/01972240290075048.

[2] Jørn Braa and Sundeep Sahay. "The DHIS2 Open Source Software Platform: Evolution Over Time and Space". In: Apr. 2017. ISBN: 9780262338127.

[3] Chitij Chauhan. *PostgreSQL high performance cookbook : mastering query optimization, database monitoring, and performance-tuning for PostgreSQL*. eng. Birmingham, England, 2017.

[4] Boyuan Chen et al. "An Industrial Experience Report on Performance-Aware Refactoring on a Database-Centric Web Application". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 653–664. DOI: 10.1109/ASE.2019.00066.

[5] Tse-Hsun Chen et al. "Detecting Performance Anti-Patterns for Applications Developed Using Object-Relational Mapping". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 1001–1012. ISBN: 9781450327565. DOI: 10.1145/2568225.2568259. URL: https://doi-org.ezproxy.uio.no/10.1145/2568225.2568259.

[6] Vittorioa Cortellessa, Antinisca Di Marco, and Paola Inverardi. "What Is Software Performance?" In: *Model-Based Software Performance Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–7. ISBN: 978-3-642-13621-4. DOI: 10.1007/978-3-642-13621-4_1. URL: https://doi.org/10.1007/978-3-642-13621-4_1.

[7] Munin Dev. *An open source networked resource monitoring tool*. URL: https://munin-monitoring.org/. (accessed: 07.10.2021).

[8] DHIS2. *DHIS2*. URL: http://www.dhis2.org. (accessed: 20.05.2021).

[9] DHIS2. *DHIS2*. URL: http://www.dhis2.org/about. (accessed: 25.05.2021).

[10] DHIS2. *DHIS2 Tracker In Action*. URL: https://dhis2.org/tracker-in-action/. (accessed: 31.10.2021).

[11] DHIS2. *Technology Platform*. URL: http://www.dhis2.org. (accessed: 17.07.2021).

[12] Henrietta Dombrovskaya, Boris Novikov, and Anna Bailliekova. *PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries*. eng. Berkeley, CA: Apress L. P, 2021. ISBN: 9781484268841.

[13] Stéphane Faroult. *The art of SQL*. eng. Sebastopol, California, 2006.

[14] Apache Software Foundation. *Apache JMeter*. URL: https://jmeter.apache.org/. (accessed: 25.08.2021).

[15] Martin Fowler. *Patterns of enterprise application architecture*. eng. Boston, Mass, 2003.

[16] Mikhail Gorodnichev et al. "EXPLORING OBJECT-RELATIONAL MAPPING (ORM) SYSTEMS AND HOW TO EFFECTIVELY PROGRAM A DATA ACCESS MODEL". In: *PalArch's Journal of Archaeology of Egypt / Egyptology* 17.3 (Nov. 2020), pp. 615–627. DOI: 10.48080/jae.v17i3.141. URL: https://archives.palarch.nl/index.php/jae/article/view/141.

[17] Jonatan Heyman et al. *Apache JMeter*. URL: https://locust.io/. (accessed: 28.04.2021).

[18] Salahaldin Juba and Andrey Volkov. *Learning PostgreSQL 11: A Beginner's Guide to Building High-Performance PostgreSQL Database Solutions, 3rd Edition*. eng. Birmingham: Packt Publishing, Limited, 2019. ISBN: 9781789535464.

[19] Rediana Koçi et al. "A Data-Driven Approach to Measure the Usability of Web APIs". In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2020, pp. 64–71. DOI: 10.1109/SEAA51224.2020.00021.

[20] Enrico Pirozzi. *PostgreSQL 10 High Performance: Expert Techniques for Query Optimization, High Availability, and Efficient Database Maintenance*. eng. Birmingham: Packt Publishing, Limited, 2018. ISBN: 9781788474481.

[21] Hans-Jürgen Schönig. *Mastering PostgreSQL 9.6 : a comprehensive guide for PostgreSQL 9.6 developers and administrators*. eng. Birmingham, 2017.

[22] Mateusz Smolinski. "Impact of Storage Space Configuration on Transaction Processing Performance for Relational Database in PostgreSQL". In: *Beyond Databases, Architectures and Structures. Facing the Challenges of Data Proliferation and Growing Variety*. Ed. by Stanisław Kozielski et al. Cham: Springer International Publishing, 2018, pp. 157–167. ISBN: 978-3-319-99987-6.

[23] Trask Stalnaker. *Glowroot*. URL: https://glowroot.org/. (accessed: 15.02.2021).

[24] BAO Systems. *Dhis2 Symposium*. URL: https://www.dhis2symposium.org/agenda/dhis2-roadmap. (accessed: 06.10.2021).

[25] DHIS2 Core Development Team. *DHIS2 Codebase on Github*. URL: https://github.com/dhis2/dhis2-core. (accessed: 10.02.2021).

[26] DHIS2 Documentation Team. *DHIS2 Implementation Guides*. URL: https://docs.dhis2.org/en/implement/database-design/organisation-units.html. (accessed: 30.09.2021).

[27] DHIS2 Documentation Team. *DHIS2 User Guides*. URL: https://docs.dhis2.org/en/use/use.html. (accessed: 12.08.2021).

[28] DHIS2 Project Support Team and HISP Bangladesh. *Bangladesh uses DHIS2 to manage immunization of 35+ million children in their MR mass campaign*. URL: https://dhis2.org/bangladesh-immunization-campaign/. (accessed: 07.09.2021).

[29] DHIS2 Project Support Team and HISP Sri Lanka. *Innovative management of COVID-19 vaccine delivery in Sri Lanka*. URL: https://dhis2.org/sri-lanka-covid-vaccine/. (accessed: 07.09.2021).

[30] DHIS2 Project Support Team and HISP Rwanda. *Rwanda uses DHIS2 as an interactive system for rapid and paperless COVID-19 vaccination*. URL: https://dhis2.org/rwanda-covid-vaccination/. (accessed: 07.09.2021).

[31] YourKit. *YourKit Java Profiler*. URL: https://www.yourkit.com/java/profiler/features/. (accessed: 21.03.2021).

A Appendix

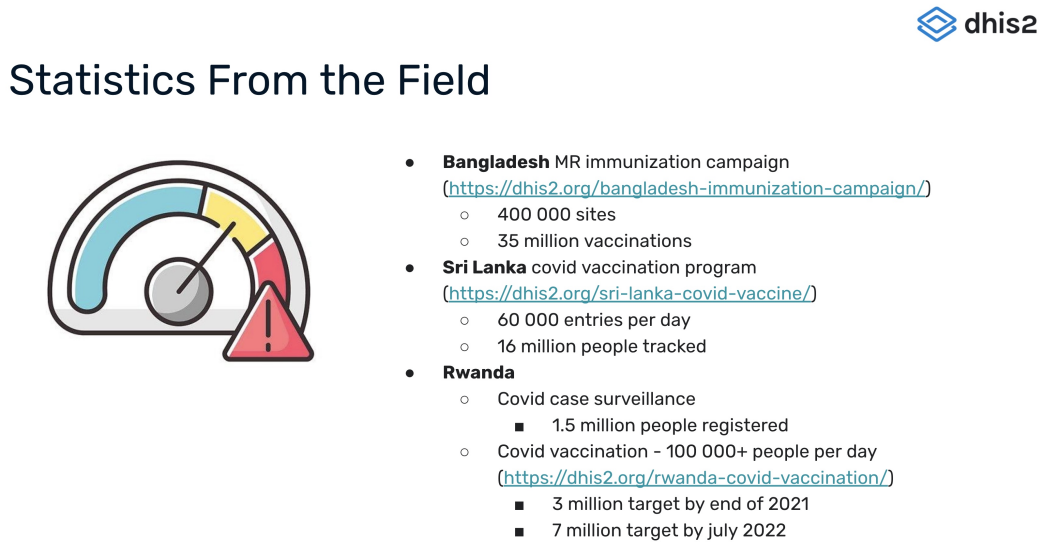

Figure 24: Statistics from field, presented by Lars Øverland, Tech Lead DHIS2, during the DHIS2 Symposium 2021 [24]

Due to size limitations, some of the raw data associated with this research are not added here. However, they have been uploaded into a Github Repository (Clickable link) for reference. It also includes some guidelines on how to reproduce my research. The repository contains the following

- Database dump that can be used to restore the same database used in our performance testing environment. (Synthetic data)
- Full queries for the queries that were trimmed in listing 1,2, 3 and 4.
- Full query plans of queries in listing 2, 3 and 4 before optimization.
- Full query plans of queries in listing 2 and 4 after optimization

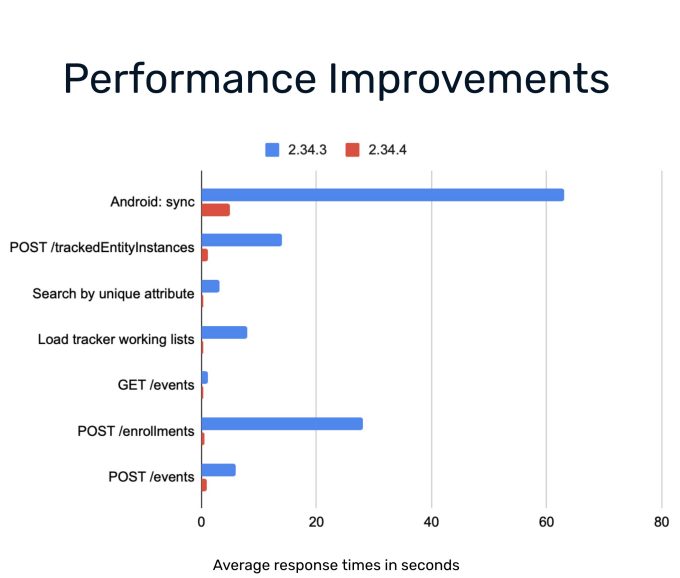

Figure 25: Performance Improvement from 2.34.3 to 2.34.4, presented by Lars Øverland, Tech Lead DHIS2, during the DHIS2 Symposium 2021 [24]

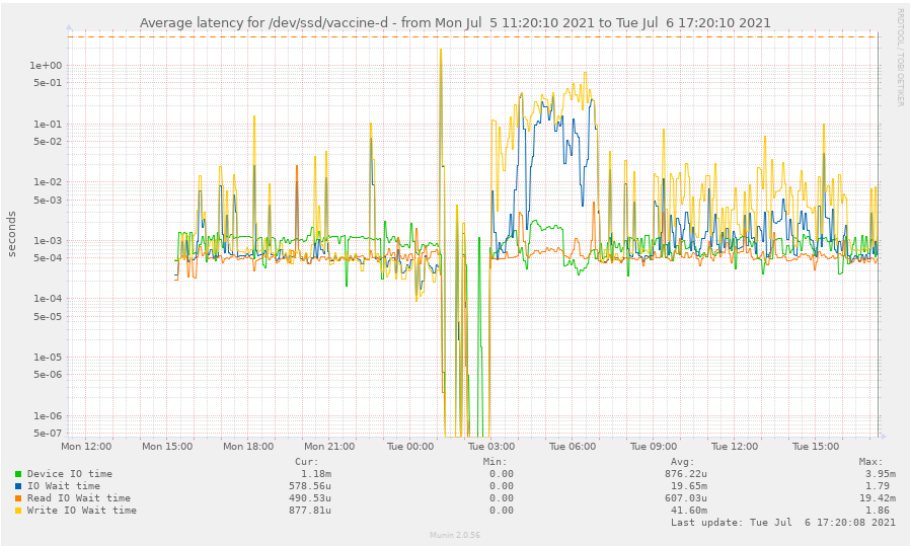

Figure 26: Munin dashboard showing Disk Latency issue in Sri Lanka that caused excessive database locking. [12]