Automated detection of performance regressions in web applications using association rule mining

Master’s Thesis

Žmicier Žaleźničenka
Automated detection of performance regressions in web applications using association rule mining

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Žmicier Žaleźničenka
born in Homiel, BSSR, USSR
© 2013 Žmicier Žaleźničenka

Front cover image © 2013 Matthew Inman
Automated detection of performance regressions in web applications using association rule mining

Author: Žmicier Žaleźničenka
Student id: 4134575
Email: D.V.Zhaleznichenka@student.tudelft.nl

Abstract

Performance testing is an important stage of developing web applications intended to operate with high availability under severe load. However, this process still remains to a large extent elaborate, expensive and unreliable. Most often the performance testing activities are being done manually, and this significantly affects development time and costs. This thesis report describes an approach aimed at automating the analysis of performance tests by maintaining a repository with the results of previously completed test runs and comparing them with the new runs to reveal deviations in software performance behavior. Detection of performance degradations is executed in a fast way using well-known data mining techniques. The results of conducted case studies clearly indicate that the suggested approach may successfully assist software engineers with detecting performance regressions in the evolving software.

Thesis Committee:
Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. A. Zaidman, Faculty EEMCS, TU Delft
Company supervisor: O. Sigida, PND Web Engineering, TomTom International BV
Committee Member: Dr. Ir. Johan Pouwelse, Faculty EEMCS, TU Delft
Preface

This thesis report was written to describe a graduation project I completed during my Master studies at the Faculty of Electrical Engineering, Mathematics and Computer Science at Delft University of Technology. This project took place at TomTom International BV in Amsterdam and contained a research of scientific methods for automated detection of performance regressions in evolving web applications.

Here I would like to thank the organizations and people who made this project possible and were helping me while working on it. First of all, I’m extremely grateful to the Dutch Ministry of Education, Culture and Science and Nuffic\(^1\) for awarding me a scholarship to study in the Netherlands. Four years spent in this wonderful country were very useful for my development and left only the pleasant memories.

Secondly, I thank all the teachers from INHolland University of Applied Sciences in Diemen and Delft University of Technology for helping me both to advance my theoretical computer science knowledge and master practical engineering skills. I would like to give special thanks to Dr. Andy Zaidman for giving academic supervision for this project in the course of the past year.

Also, I’m grateful to TomTom for accepting me as an intern for my Bachelor and Master graduation projects and providing with all necessary support to successfully complete both of them. I especially would like to thank TomTom employees Oleg Sigida for supervising my project, Mykola Gurov for answering my endless engineering questions and Aliaksandr Antonik for the thorough review of this document and many useful suggestions to improve the approach I present here.

Last but not least I would like to thank my parents for trusting in me and helping with everything I did and Dasha for motivating me to work hard the way only she can.

Žmicier Žaleźničenka  
Delft, the Netherlands  
November 5, 2013

---

\(^1\)Dutch organization for international cooperation in higher education.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of tables</td>
<td>ix</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Research questions</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Thesis structure</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Performance engineering practices at TomTom PND Web Engineering</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Company overview</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Web Engineering department</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Testing environments and release process</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Performance testing practices</td>
<td>8</td>
</tr>
<tr>
<td>2.5 Performance monitoring tools and techniques</td>
<td>9</td>
</tr>
<tr>
<td>2.6 Caching policies</td>
<td>10</td>
</tr>
<tr>
<td>2.7 Summary</td>
<td>11</td>
</tr>
<tr>
<td><strong>3 Literature study</strong></td>
<td>13</td>
</tr>
<tr>
<td>3.1 Automatic generation of realistic workloads</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Automatic detection of performance regressions</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Summary</td>
<td>21</td>
</tr>
<tr>
<td><strong>4 Performance testing repository structure and functionality</strong></td>
<td>23</td>
</tr>
<tr>
<td>4.1 Related software</td>
<td>23</td>
</tr>
<tr>
<td>4.2 Repository characteristics</td>
<td>24</td>
</tr>
<tr>
<td>4.3 Technological stack</td>
<td>25</td>
</tr>
</tbody>
</table>
## CONTENTS

4.4 Repository structure and terminology ........................................ 26  
4.5 Performance counters .......................................................... 26  
4.6 Basic analysis tools .................................................................. 30  
4.7 Workflow .................................................................................. 31  

5 Application of association rule mining for performance testing .......... 35  
5.1 Classification of data mining algorithms ...................................... 35  
5.2 Association rule mining ............................................................. 36  
5.3 Performance testing challenge .................................................... 39  

6 Dasha: a performance testing mining algorithm ............................... 43  
6.1 Dasha overview ......................................................................... 43  
6.2 Sample datasets ......................................................................... 44  
6.3 Support tree structure and generation ........................................ 45  
6.4 Dasha implementation ............................................................... 46  
6.5 Summary .................................................................................. 57  

7 Case study: Dell DVD Store ............................................................ 59  
7.1 Testing setup .............................................................................. 59  
7.2 Experiments ............................................................................... 62  
7.3 Report generation ...................................................................... 68  
7.4 Summary .................................................................................. 69  

8 Case study: TomTom PND Web Engineering .................................... 71  
8.1 Web application under study ...................................................... 71  
8.2 Testing setup ............................................................................. 73  
8.3 Experiments ............................................................................... 75  
8.4 Summary .................................................................................. 79  

9 The algorithm’s performance study ................................................ 81  
9.1 Discussion ................................................................................ 81  
9.2 Testing setup ............................................................................ 82  
9.3 Experiments ............................................................................. 83  
9.4 Measures to improve performance ............................................ 93  
9.5 Summary ................................................................................ 97  

10 Conclusions and future work ...................................................... 99  
10.1 Contributions ........................................................................... 99  
10.2 Discussion/Reflection .............................................................. 100  
10.3 Future work ............................................................................ 102  
10.4 Conclusions .......................................................................... 104  

Bibliography .................................................................................. 105
List of figures

4.1 List of environments. ......................................................... 31
4.2 Environment details. ......................................................... 31
4.3 Test details. ................................................................. 32
4.4 Test run details. ............................................................ 33
4.5 Data files processing. ...................................................... 34

6.1 Support trees for the sample datasets. ................................. 50

7.1 Time-series charts and boxplots for experiment 7.2.2.2. .......... 65
7.2 Categories distribution for experiment 7.2.2.2. ...................... 66
7.3 Test details page. ............................................................ 68
7.4 Analysis results page. ...................................................... 70

8.1 Slow requests affecting latency counter. .............................. 77

9.1 The Dasha algorithm’s performance on one-liner datasets. ....... 85
9.2 The Dasha algorithm’s performance on T10.I4.D100K with fixed mindiff. ... 88
9.3 The Dasha algorithm’s performance on T10.I4.D100K with fixed minsup. ... 89
9.4 The Dasha algorithm’s performance on Mushroom dataset. .......... 93
List of tables

5.1 An example of a performance testing database. ......................... 40
6.1 Sample datasets. ............................................................... 45
7.1 Performance counters tracked during DS2 analysis. .................... 61
9.1 The Dasha algorithm’s performance on one-liner datasets. .......... 85
9.2 The Dasha algorithm’s performance on T10.I4.D100K with fixed mindiff. 88
9.3 The Dasha algorithm’s performance on T10.I4.D100K with fixed minsup. 89
9.4 The Dasha algorithm’s performance on Mushroom dataset. ........... 92
Chapter 1

Introduction

1.1 Motivation

Software testing is an important part of the software development lifecycle. Its primary goal can be stated as observing the execution of a software system to validate whether it behaves as intended and identify potential malfunctions [9]. A number of researchers report that around 50 percent of elapsed time for an average programming project is spent on testing the system being developed [8, 38]. The total costs spent on testing procedures are usually even higher and do not tend to decrease with evolution of development tools and methodologies [38]. Such tremendous expenditures on testing processes during software development imply that improvements to it should lead to significant decrease of development time and costs, hence software testing is always a hot topic in the research community. However, it still remains most probably the least studied part of the development process [38]. Despite all the attempts to formalize the testing activities, they still remain largely ad hoc, expensive and unpredictably effective [9].

System validation with the testing techniques is meant to be performed against a set of system requirements that can either be functional or non-functional. From this separation of the requirements it is logical to split the software testing process into functional and non-functional testing, with each part of the process being responsible for validation of the respective subset of system requirements. While it can be said that functional testing should answer the question about what the observed system has to do, non-functional testing is responsible for answering how the system behaves while performing its tasks.

Non-functional testing is itself a broad research field as it addresses such important and complicated topics as performance, scalability, usability, security, accessibility and more [33]. Testing the software system for compliance to these qualities requires different techniques and skills. This thesis addresses automated detection of performance regressions that is an important challenge in performance testing.

Those researchers who define performance testing or its goals in their papers, usually do it by means of performance metrics, the most important of which are latency (also defined as response time or responsiveness), throughput and scalability [15, 32, 33, 34]. Molyneaux in [37] does not operate with the metrics in his definition but says that the performance
1. **Introduction**

Testing is ultimately about the user’s perception and “a well-performing application is one that lets the end user carry out a given task without undue perceived delay or irritation”.

Performance testing is usually an elaborate and time-consuming process. Much effort is often spent by performance analysts on executing and analyzing test plans to reveal non-affordable performance degradations during the software evolution process before the final product is delivered to the customers. Large and complicated test plans used for thorough testing of the software products may report thousands of performance counters and in order to get proper understanding of the processes in a system and their effect on eventual performance results, all of them have to be studied attentively. Usually, this work is done manually, the engineers try only to analyze a small subset of the most important counters, which often leads to the incomplete picture of the system behavior.

During software products development, performance testing activities are usually left to the ending phase of the development iterations. Thus, execution and analysis of performance tests is often close to a deadline. This imposes additional time challenges on testing engineers not allowing them to conduct and analyze all the tests they want [39]. From this it follows that automating the process of detecting changes in performance between different versions of software systems does not only help to cut development costs and time but may provide detailed feedback which would simply not be achievable using manual and heuristics-based approaches. In addition, introduction of an automated approach to assist software developers with performance regressions detection may mitigate the effects of other drawbacks usual for manual heuristics-based performance testing, e.g. undocumented baselines, subjectivity of certain performance analysts [19], inconsistence of performance counters [39], etc.

The graduation project described in this thesis document aims to devise a technique for automated detection of performance regressions and provide a publicly available open-source implementation of an environment supporting the analysis methods for this process.

1.2 **Research questions**

The research question investigated in this thesis is *How to identify performance degradations and their causes in enterprise web applications?* It can be split into a number of sub-questions, namely

1. How to store the results of performance tests from different sources for the further analysis?

2. What useful information can be extracted from the performance test results and what data mining techniques can be used for its retrieval and analysis?

3. How to detect performance degradations in a performance test run having the results of previous (successful) runs? Is association rule mining a suitable technique for this task?

4. How to represent the results of automatic performance testing analysis to provide the analysts with the detected performance degradations and their justification?
1.3 Thesis structure

The rest of this thesis is organized as follows. Chapter 2 describes the company where the research project was completed and performance engineering practices used there. Chapter 3 contains a literature study covering the state-of-the-art research in automatic generation of realistic workloads and automated detection of performance regressions. Chapter 4 describes an implementation of a performance testing repository aimed at storing the results of performance test runs. Chapter 5 introduces association rule mining and presents the most important algorithms implementing this technique. Also, this chapter contains a discussion on applicability of association rule mining algorithms for performance testing tasks. Chapter 6 contains a description of a new data mining algorithm called Dasha that is aimed to detect regressions between performance test runs using association rule mining techniques. Chapter 7 presents a case study testing Dasha’s applicability to detect performance degradations on a simple example of Dell DVD Store\(^1\) web benchmark. Chapter 8 discusses the applicability of a research prototype for performance testing of a large enterprise web application. Chapter 9 investigates the performance of the Dasha algorithm for different datasets and suggests a number of measures to improve it. Chapter 10 concludes the thesis and contains directions for future work.

\[^1\]http://linux.dell.com/dvdstore/
Chapter 2

Performance engineering practices at TomTom PND Web Engineering

2.1 Company overview

TomTom International BV is a Dutch company manufacturing portable navigation devices and providing supplementary services to improve drivers’ experience. The company has over 3500 employees located in nearly 40 offices all around the world. Its headquarters is in Amsterdam.

A significant number of the company’s employees are involved in designing, building, testing and delivering software, both for the mass market and internal use. These software include firmware for the manufactured devices, content updates (i.e. map changes), smartphone applications, web site and web shop, fleet management tools and more.

It is natural that due to variability of applications developed by the TomTom engineers, software testing practices between the company’s departments differ significantly. What is suitable for testing an embedded navigation system usually can hardly be applied for benchmarking a distributed web application serving customers all over the globe.

2.2 Web Engineering department

In this chapter the performance engineering practices at TomTom Portable Navigation Devices Web Engineering (PND WE) department are described. PND WE is a part of Consumer group responsible for delivering products for the mass market. The main responsibility of the department is to develop and maintain web services oriented at the end users having TomTom PNDs or one of the navigation applications for the smartphones. The services developed at PND WE include management of customers’ accounts and states of their associated devices, management of subscriptions to regularly updated content (maps, traffic, safety cameras and more), product catalog and order handling for the paid content, promotions activation, a dedicated web shop to let customers choose and download the content via web interface, common API layer to let PNDs connect to the services and more.

All these services interact with each other and are logically organized in a single system
that is deployed at several dozens of computational nodes serving as front-end, back-end and database servers. Such a large number of the servers is needed as each component of the system is deployed at several machines and the requests to the systems are distributed between the nodes using the load balancers. Also, a number of servers is used to maintain legacy systems that are not actively developed any more but are still used by the customers possessing old navigation devices. Mutual dependence of the services from each other, different caching policies applied at the system’s components and inconsistencies in the monitoring solutions supported by the different components make performance testing of the services in a system a non-trivial task.

2.3 Testing environments and release process

2.3.1 Development environments

The company has a number of testing environments that aim to emulate the structure of the production system for testing purposes. These environments are built to serve different needs. First of all, there are five development environments to host unstable code coming from early development branches. The most recent versions of the software run on these environments and the developers use them to perform functional tests addressing the core functions of the components and integration of new and updated features with the rest of the code base. Also, the developers usually connect to these environments from their local machines having one of the components deployed locally. While developing a particular part of a system, it is needed to establish communication with the other components and the development environments serve this purpose. These environments are small and compact, they have relatively poor performance which makes them less efficient for testing integration or performance issues, but they are perfectly suited for early functional testing and development support needs.

When all the feature branches of a system component pass initial functional tests at the development environments, the introduced changes get merged into a single release branch. As a general rule, after the initial merge of the feature branches, the release branch freezes from addition of the new features. It is only allowed to commit bug fixes into it. Release candidate code gets installed at one of the development environments. This is the starting point for the new release. After a successful deployment, a development environment with release candidate code is used by software engineers for alpha-testing.

2.3.2 Early integration environment

When the release branch passes the functional tests in a development environment, it receives a sign-off from a product owner and gets installed at an early integration environment for more thorough testing. This environment is used to host stable code builds that have already passed the initial testing procedures. While development environments are mostly used to test code changes within a certain component, the code deployed at the early integration environment is tested to ensure correct inter-component communication.
The early integration environment is actively used by quality assurance engineers (QA). While the tests at the development environments are mostly run by the software developers whose main responsibility is to write program code, integration and regression testing is performed by the QAs who have more advanced tools, run more tests and verify the correctness of the deployed code at a higher level. Apart from using the automated tools to run different software testing scenarios, they also have a wide range of portable navigation devices produced by TomTom to test interaction with the system from the prospective of real users.

Another purpose of the early integration environment is to serve as an initial site for non-functional testing. Its logical structure resembles the structure of the production environment which allows the engineers to test deployment procedures on the release candidate build and verify the correct work of monitoring solutions. However, as the physical structure of the early integration environment does not look alike to the production system, it is not suitable for performance testing.

### 2.3.3 Pre-production environment

After the release branches of system components pass integration and regression testing stages at the early integration environment, they can be deployed at a pre-production environment for the final validation of deployment procedures and performance testing. The pre-production environment differs from the early integration environment as it resembles not only the logical structure of the system, but also its physical structure: all the components of the system are deployed at several physical computational nodes, communicate with several replicated database servers and are located behind a load balancer and a firewall. While the number of computational nodes differ at production and pre-production environments, the deployment procedures for them are exactly the same which means that once the release candidate version is successfully deployed to the pre-production environment, it can also be deployed to the production system following the same procedure. Also, performance of the pre-production environment is comparable to the production system which allows performance engineers to run their tests at the pre-production system and use the test results as a baseline for estimating performance of a release at the production system.

The pre-production environment is also used to investigate causes of incidents that happen at production system. If network engineers report strange behavior of the deployed software (e.g. unexpected errors, cache misses, customer complaints about the wrong data) and the cause of an incident is unclear to the developers after inspecting the logs, the pre-production environment gets synchronized with the production system and the engineers repeat failing scenarios at the pre-production system to locate the code causing the incident. In certain situations it is impossible to use lower-level environments for this tasks, as the components there are not distributed across several nodes. Some errors, however, appear only at the distributed environments and are caused not only by the flaws in the software, but also by its misconfiguration, state of the caches, infrastructural problems or hardware failures.
Only after the new release of a system component passes the final tests at the pre-production environment, it can be deployed to the production environment. During the deployment process and after it the network engineers constantly monitor the state of the servers to immediately detect the errors and roll the release back for investigation if deployment fails.

Thus, at TomTom PND WE there are three different layers of testing environments and the production environment that accepts requests from the customers. All the environments have different roles and are used at different stages of testing and release processes. Keeping a number of different testing environments allows the engineers to simultaneously test several versions of the software, effectively investigate the incidents at the production system not affecting its usual operations and maintain automated release procedures to easily deploy new versions of the software to the production servers and roll them back in case of sudden failures.

2.4 Performance testing practices

Performance testing at PND WE is a responsibility of the performance engineering team. Usually, the performance tests are run at the pre-production environment when a stable release candidate code is deployed there. The performance tests are written using the HP LoadRunner\textsuperscript{1} tool.

During the performance testing process, the engineers issue a large number of requests to the system. These requests represent the activities of virtual users that complete the most common tasks the real users do. These tasks include login into the system, retrieving the device status and available updates, downloading the list of available products, purchasing and downloading the content, etc.

After the performance tests are run, the engineers manually analyze the resulting charts looking for general performance levels and unexpected spikes. In most situations, the new results are in line with the previous ones, which means that no additional investigation has to be performed. If a performance test shows suspicious results, they are reported to the development teams.

The results of the test runs are accessible in the intranet. Any software developer may look at performance of the components he is responsible for. However, there is no possibility to automatically compare the results of the previous runs with the results of the most recent test. It is a responsibility of an engineer to define a performance baseline and decide whether the performance in the latest test has degraded or improved.

If the tests reveal performance degradations in a new software release, the developers analyze its execution log and use dedicated monitoring and profiling tools to find the causes of the regressions. Performance degradations detection process is not automated, in every case all the investigation is performed manually using well-known heuristics.

\textsuperscript{1}http://www8.hp.com/us/en/software-solutions/software.html?compURI=1175451
2.5. Performance monitoring tools and techniques

2.5.1 AppDynamics

The most advanced monitoring tool used to track the behavior of PND WE software deployed online is AppDynamics\(^2\). This is a dynamic application performance management suite that allows to monitor and troubleshoot web applications in real-time. It can analyze all the existing API calls and gather statistics for each of them as well as track the hardware counters. AppDynamics can build a map of the whole network environment in a company and show the request flows between the different servers. Also, it is able to work as a real-time software profiler with low overhead for the application servers and as an advanced database monitor for the database servers. It has flexible automation, visualization and notification options. AppDynamics is a very powerful and intelligent monitoring application that suits virtually every need of software developers and network engineers.

Despite the fact that AppDynamics covers most of the use cases needed for the software monitoring and troubleshooting needs, a number of less advanced monitoring-related tools are also used at PND WE. These tools are needed as even such advanced software as AppDynamics has its drawbacks. First of all, this software is very expensive and its price depends on the number of installed licenses. Hence, for some of the servers, especially not used to support the applications serving real customers (e.g. at testing environments) it is more cost-effective to use cheaper solutions. Secondly, AppDynamics is the new software that is being developed very actively and it has been initially deployed at PND WE just around a year ago. Its functionality and documentation change quite often and require significant time investments from the engineers who want to learn the most efficient ways to work with it. On the other hand, prior to AppDynamics deployment there were different monitoring solutions accepted at the department. They work well and changing all the custom scripts written to address the business needs for AppDynamics will not add any significant value.

2.5.2 Munin

Thus, the Munin\(^3\) monitoring tool is still used widely for monitoring state of the application servers. This software is relatively old and it does not improve with the same pace as AppDynamics. It is a rather simple application that gathers different performance counters, stores them in a round-robin database with fixed amount of data\(^4\), plots them and outputs the charts to a web page.

Munin has a number of serious drawbacks, as certain throughput issues or inability to gather the metrics with resolution less than five minutes. However, it has a large number of available plugins that allow to track many different counters with ease. Also, due to its simplicity it is very easy to build new plugins extending its functionality in whatever way

\(^{2}\)http://www.appdynamics.com  
\(^{3}\)http://munin-monitoring.org  
\(^{4}\)http://oss.oetiker.ch/rrdtool/tut/rrdtutorial.en.html
needed. The existing Munin deployments at the department are supported with Nagios\(^5\) installations providing the alerting functional.

### 2.5.3 OpenTSDB

Another monitoring tool deployed at a number of TomTom servers and aimed to replace Munin for tracking and visualizing the performance counters is OpenTSDB\(^6\). Contrary to Munin, this tool is built on top of Apache HBase\(^7\) distributed database. This is a new application that was initially released just a year and a half ago. It does not have the drawbacks of Munin, as it scales well and allows to gather the counters with a second precision. Also, it does not override and merge the gathered data as Munin does for saving the disk space, thus OpenTSDB allows the engineers to get back to any gathered monitoring results for further analysis. However, this software is still not mature and its plotting capabilities are rather weak. Many improvements are still to be made to it before it becomes as easy to use as Munin is. It is also possible to write simple plugins for OpenTSDB, but as the software is new and its community is not yet large enough, not many ready-to-use plugins are available for immediate installation. For these reasons, many engineers still prefer Munin to OpenTSDB for monitoring the states of their servers.

TomTom developers are currently working on a common monitoring infrastructure that would allow to write a single plugin tracking any important performance counters and to use its output in all the monitoring applications currently deployed at the department. However, switching to this infrastructure requires updating all the existing performance measuring scripts, which means this process will take a lot of time.

While AppDynamics tool is used both by software developers and network engineers, the other tools are mostly used by the network engineers only. As the software written in the department mostly does not have serious performance issues, there is no need to constantly monitor the state of the applications from the developers’ side. However, any of the tools described earlier in this section can be used by the developers while investigating the performance-related incidents.

### 2.6 Caching policies

As the applications developed within the department serve millions of customers worldwide, they are usually used under high load. Thus, it is vitally important to correctly use caching to improve performance of the software and decrease the load in every way possible. For Java applications written at PND WE, Ehcache\(^8\) is an adopted distributed caching framework. It is used both to store generated internal data between the similar requests and

\(^5\)http://nagios.org  
\(^6\)http://opentsdb.net  
\(^7\)http://hbase.apache.org  
\(^8\)http://ehcache.org
to keep the responses to SQL queries from an object-relational mapping framework. Until recently, every development team within the department had their own code responsible for interacting with the caching layer. Nowadays, the developers have switched to the common caching layer, which allows to decrease work on maintaining its code and to improve integration between different applications.

Apart from supporting caching at the applications level, certain caching policies are applied at the servers. These caches work on a higher level by processing the results of the requests to the company applications at a whole.

2.7 Summary

Performance is a serious concern for the engineers developing web applications at TomTom PND WE, as the software being created and maintained within the department serves millions of customers and operates under a high load. To ensure the quality of delivered code and the availability of the servers both during normal operation and in peak load condition, TomTom engineers apply different performance engineering practices, including benchmarking the applications at the testing environments, running various monitoring tools and observing their outputs, maintaining several software caching levels. Correct application of these practices allows to avoid most performance-related problems with the software being developed.
Chapter 3

Literature study

This chapter contains an overview of the most significant papers presented in recent years that address two important problems related to performance testing of web systems. The first problem is the generation of traces for workload generators. Such generators are used for load testing the applications to analyze their behavior under high load and prove they can sustain the required load on available hardware. This task is solved with generating synthetic traffic distributed according to the requirements specific for a given application and applying it to an application deployed in the test bed. For successful load testing it is required from the synthetic traffic to resemble the patterns occurring at production servers. Many important load testing challenges arise when designing methodologies for generating these realistic workload traces. The studies approaching this problem with user equivalents and aggregate traffic generation techniques are overviewed. The strengths and weaknesses of both approaches are discussed. A number of other important concepts relevant for the field are also presented here.

The second problem is related to the detection of performance degradations. A common way to address it nowadays is to inspect performance testing results between different releases of the software manually. However, this method is not very efficient as it is rather elaborate. Moreover, due to the huge amount of performance parameters to be investigated during the performance testing process and many different scenarios of software usage, even complicated and elaborate manual performance testing scenarios are not always able to reveal all the performance problems that affect the application during its lifecycle. A number of approaches aiming at automating the process of detecting performance regressions in evolving software were proposed recently. Such performance regressions detection techniques as association rule mining, control charts, configuration space exploration, performance signatures and execution log analysis are discussed in this chapter. Their applicability for the industrial usage is analyzed.

3.1 Automatic generation of realistic workloads

The existing contemporary load testing tools are mature and powerful enough to assist performance engineers with creating and executing test plans of virtually arbitrary complexity.
It is relatively easy to load test the applications with their help nowadays. However, while these utilities help analysts to efficiently simulate flows of virtual users accessing web applications under study, they do not assist with building realistic user scenarios covering all the patterns that are important for the given application. Identification of these scenarios has to be performed by the analysts prior to building test plans and executing them using the load testing tools. In other words, it can be said that generation of synthetic workloads consists of trace generation and request generation steps. While the trace generation step is responsible for creating a synthetic trace of server requests adhering to the workload model, at the request generation step the requests from the trace are submitted to the system ([27, 29]). The existing performance testing tools mostly assist with requests generation, not with traces generation. While the request generation stage is a fairly straightforward process, realistic trace generation is a far more challenging task that has attracted much attention from the scientific community during past two decades. Common approaches to this problem can be separated into analytical (modeling) and empirical (execution log analysis) ones.

The empirical approach to generating realistic workloads in most situations can be implemented relatively easy. However, it has certain drawbacks. First of all, if using the empirical approach, the workload is treated as a black box and it is not possible to tune workload characteristics finely which can be needed for some testing scenarios. Secondly, the empirical approach depends on a realistic execution log that does not exist for newly created or significantly changed systems.

The analytical approaches are based on constructing workloads using performance modeling and do not have these drawbacks, but usually require bigger time investments on building correct performance models reflecting all the important characteristics of the studied application.

It is rather difficult to devise a methodology for generating realistic traces for load testing web applications as this process involves a number of challenges that have to be resolved. The most important of them are accuracy of a methodology that is determined by how closely the generated synthetic workloads resemble the real workloads, flexibility or fine-grained control over the workload attributes and wide application acceptance for covering different classes of applications with significantly varying load patterns [5].

### 3.1.1 Important observations and concepts

One of the most important concepts discovered in modeling workloads for the web is self-similarity of web traffic. The initial study revealing this traffic property was conducted by Leland et al. in [30]. The results achieved in this research were then extended in many publications (e.g. [13, 14]). Self-similarity means that at a very wide range of time scales from milliseconds to hours the workload consists of bursty subperiods separated by less bursty subperiods. The degree of self-similarity depends on utilization of network. Self-similar or fractal-like behavior of real traffic flows differs from conventional telephone traffic and prior formal models of packet traffic and has to be taken into account when developing generators of synthetic workloads for load testing web applications.
3.1. Automatic generation of realistic workloads

An important study covering construction of distributional models for analyzing performance of web sites was performed by Barford and Crovella in [6]. In this paper the authors have introduced a technique that is widely used in many consequent studies exploring analytical approaches to generation of realistic web workloads. It is called User Equivalent (UE) and means a single process alternating between issuing HTTP requests and lying idle. The authors claimed that for the testing process it is important to model the distributional and correlational properties of web traffic not only for issuing web requests, but also for idle times. Each UE is thus called an ON/OFF process and it is assumed that the UEs are to be implemented as separate threads or processes. The authors have shown that ignoring idle times destroys self-similarity in generated synthetic workloads.

An entirely different way to build a request generator using analytical modeling was described by Kant in [26, 27]. These papers contain theoretical basis behind aggregate traffic generation (AGT) and description of Geist traffic generator that employs this approach.

In the AGT, the aggregated traffic is generated directly, it is not formed as super-position of traffic generated by the individual users. Application of direct aggregate traffic generation is aimed to mitigate such drawbacks usual for the UEs as difficulty to control aggregate traffic properties, impossibility to correctly reflect network effects and poor scalability. The AGT approach helps to generate traffic with asymptotic self-similarity, multifractal properties and nonstationarity which are commonly observed in Internet traffic. A drawback of this approach is that it may run into difficulties when detailed user modeling is needed, as it cannot properly model interrequest dependencies. Also, the AGT is based on more complicated computing formulae than most of the approaches based on the user equivalents. More detailed comparison of the UE and the AGT techniques can be found in [27].

Menascé et al. in [35] have introduced Customer Behavior Model Graphs (CBMG) which are a concept used both in analytical and empirical approaches to generating realistic workloads. CBMG is a state transition graph used to describe behavior of groups of users exhibiting similar navigational patterns. Web performance metrics can be derived from the analysis of these graphs. CBMG is a probabilistic model using Markov chains which is used for characterizing web workloads by a number of researchers in their generators. CBMGs are derived from the execution log of the web servers.

Krishnamurthy, Rolia and Majumdar in [29] address the importance of interrequest dependencies for session-based web systems. Interrequest dependencies arise because some of requests to the server depend on results of the previous requests in a session. For example, it may only be possible to submit a commentary at the forum after logging in. The authors refer to the systems having interrequest dependencies as to session-based systems and call systems not having interrequest dependencies request-based systems. For the session-based systems it is important to reflect interrequest dependencies in the synthetic workloads. In the approach discussed by Krishnamurthy et al., the interrequest dependencies are deduced automatically.

3.1.2 Recent analytical approaches based on user equivalents

Draheim et al. proposed load testing web applications using form-oriented analysis in [16]. In this methodology a web application is described as a typed bipartite state machine con-
sisting of pages, actions and transitions between them. User interaction is performed by means of forms placed at the web pages and offering a way to submit the information. Form-oriented models are visualized as formcharts being bipartite direct graphs. The formcharts allow to specify possible user behavior as well as model its timing for realistic simulation, thus implementing a UE-based approach. This approach employs a stochastic model to estimate probabilities of transitions between different pages. To calculate probability distributions, session histories are used. As well as generating the realistic pages transitions, this approach allows to generate input data for the chosen forms at the web pages. The suggested approach can be useful for load testing the applications that can be efficiently modeled using form-oriented analysis, however the paper discussing it lacks a case study validating that the form-oriented approach is able to generate realistic load traces.

The results achieved by Barford and Crovella in [6] were used by van Hoorn et al. in [44] to develop an extension to JMeter tool called Markov4JMeter. In their work, the authors have developed two types of mathematical models aimed to generate realistic simulated workloads. An application model considered as a hierarchical finite state machine is used to define possible interactions with a web application and underlying protocols. A number of user behavior models corresponding to the analytical model are used to specify probabilistic and intensity-varying workloads by means of Markov chains. The user behavior models are based on the concept of CBMGs presented in [35].

Workload specification of the probabilistic model for Markov4JMeter consists of an application model being a hierarchical finite state machine, a number of user behavior models specified as Markov chains, a user behavior mix being probabilities of the individual user behavior models to occur during workload generation and workload intensity specified as the number of users to simulate during the experiment.

Bahga and Madisetti employed user equivalents technique to develop a generic methodology for workload characterization, modeling and generation for cloud computing applications in [5]. The authors motivate their work with a lack of a standard approach for specifying workload attributes for different application benchmarks. The proposed methodology aims to cover a whole range of tasks used for the synthesis of realistic workloads. It includes techniques for extracting semantic and time behaviors from applications, benchmark and workload models, a workload specification language for defining application workloads to use in workload generation and the workload generation techniques based on workload specifications of the enterprise applications. However, the proposed methodology is work in progress and has to be improved in many aspects before it can be recommended for practical usage.

3.1.3 Analytical approach based on aggregate traffic generation

A technique of aggregate traffic generation discussed earlier in this chapter was implemented by Kant, Tewari and Iyer in the Geist traffic generator [27]. The Geist model considers temporal distribution of the traffic, user behavior impacts on it (self-similarity), network dynamics effects (multi-fractal properties), traffic nonstationarity and its transactional (GET/POST requests distribution, secure/insecure requests etc.) properties. Virtual user streams in Geist are created based on input parameters of desired aggregate traffic charac-
3.1. Automatic generation of realistic workloads

teristics and the transactions of these users are marked according to the Markov chain model. Geist was used in Intel corporation to generate traffic for studying SSL performance, proxy servers and overload control strategies for web servers as well as in the other projects. This traffic generator seems to be an efficient tool for thorough load testing web applications, however its source and compiled codes are not publicly available, thus no conclusions on its effectiveness in comparison with the other tools can be drawn.

3.1.4 A hybrid analytical approach

Weber and Hariharan have attempted to combine user equivalents and aggregate traffic generation techniques into a hybrid approach based on queuing models in [47]. In their paper the UE model is considered a closed system of two queues feeding one another, the AGT is modeled as an open system with one queue. The proposed hybrid queuing model combines the concepts of open and closed queues mitigating the drawbacks of transience and recurrence by introducing additional processes responsible for letting a client process to go through several ON/OFF cycles during the session. Transience is a property of open queuing models leading to the inability for content cache usage by the clients. Recurrence is a property of closed queuing models leading to caching of all the incoming requests. In the hybrid approach, each client is associated with the own cache and the requests are cached similarly to the policies used in the clients’ web browsers. However, the models discussed in this paper were rather simple in comparison with the source models and superiority of the presented hybrid approach over the underlying techniques was not shown clearly, thus the hybrid approach did not attract much attention from the research community.

3.1.5 An empirical approach

Krishnamurthy, Rolia and Majumdar in [29] have presented a technique to construct a synthetic realistic workload based on analysis of execution log containing user session identifiers. The suggested approach allows to construct a workload model that simultaneously satisfies characterizations of the model attributes and maintains correct interrequest dependencies.

In contrast to the UE-based techniques that employ the Markov chains, the approach suggested in the discussed paper does not have two significant drawbacks, namely inability to address interrequest dependencies and lack of flexibility. The first drawback relates to the fact that the first-order Markov chains used for workload generation have states representing request types and it is assumed that the transitions between the states depend only on the current state. However, they may also depend on the previous states and not considering it may lead to violating interrequest dependencies. Due to the second flaw it is impossible to vary workload mix and session length distributions independently of one another for the first-order Markov chains. If comparing the suggested approach with the AGT technique, using the latter one it is possible to submit a request in a session prior to receiving a response to the previous request which is also a serious violation.

A case study performed by the authors has shown that synthetic workload generated out of session data from the reference workload causes utilization of all the measured sys-
tem resources comparable to the utilization during performance testing using the reference workload. The closeness of cumulative distributions of response time and number of concurrent sessions has confirmed that the system’s performance was nearly identical under both workloads. In addition, the results have shown that the workload that does not maintain correct interrequest dependencies places significantly less stress on a system under study which reinforces importance of preserving correct interrequest dependencies in load testing web applications.

3.2 Automatic detection of performance regressions

A number of statistical approaches were proposed recently to address the problem of automatic performance degradations detection. This section is aimed to overview these techniques and discuss their benefits, drawbacks and applicability for industrial use.

3.2.1 Detecting performance anomalies using application signatures

Cherkasova et al. in [12, 36] discuss detection of performance anomalies in large-scale transactional applications. The researchers claim that application performance in normal circumstances can be modeled using application performance signatures. Such signatures are meant to be the composite metrics that represent application performance and do not vary if the load on a system changes. They only change with performance.

In [36] the authors develop transaction signature metric that helps analyzing performance of transactions at the application server. To build this metric, the whole transaction latency is split into the portions representing time spent at application and database servers. Then, the transaction latency at the application server is augmented with CPU utilization of the same server. It is assumed that these two counters are directly proportional and their ratio is constant. If this ratio fluctuates significantly for different releases, this should imply performance difference between them. The latencies are retrieved from the execution log.

In [12] the authors introduce a regression-based transactional model that approximates CPU cost of application transactions for given hardware configuration. The algorithm used to build a model consists of three steps. First, the optimal segmentation is found. At this stage the time points where the cost model exhibits a change are detected. Then, the segments containing anomalies are filtered out. This is done to build an anomaly-free model reflecting normal behavior of the application. At the last step, the remained time segments get unified using a single regression model. The anomalies get reported. Introduction of this performance model helps to detect differences in CPU consumption model of transactional applications.

A definite benefit of the discussed approach is that it mitigates the effects of randomization usual for performance measurements based on generated data input. However, this leads to non-stationary loads and may have impact on interpretation of load testing results if the performance testing technique does not consider load variability. Also, a drawback of the proposed technique is that it can only reveal CPU-related bottlenecks in transaction latencies. If the application has I/O bottlenecks, they will not be detected. And to detect CPU-related bottlenecks for different types of applications, the respective signatures have
3.2. Automatic detection of performance regressions

to be created in advance, which is a complicated process. Thus, it is hardly possible to consider the suggested approach ready for use in the industry, but the idea of creating application performance signatures is definitely valuable and is successfully implemented in the following study.

3.2.2 Capturing correlations among performance counters with association rule mining techniques

Foo et al. in [18, 19] address the problem of automated performance regressions detection with a technique of mining performance testing repositories to capture correlations among performance counters and analyzing them using association rules. This approach requires a performance testing repository built during the baseline runs to generate performance signatures that are validated against the counters in a target run. Violations of these signatures are considered performance problems.

To implement the suggested approach, an analyst has to build the performance repository out of a number of runs that have satisfactory performance and gather the performance counters from a target run. Then, the obtained performance counters have to be normalized and classified. Normalization of the performance counters is needed to mitigate the side effects of raw data, e.g. clock skew or delay. Classification is an important step in preparing data for use in machine learning algorithms for the derivation of performance signatures.

The performance signatures are extracted from data using frequent itemsets and association rules. The authors suggest using the Apriori algorithm [3] to discover the frequent itemsets describing mutually dependent counters. This algorithm should automatically reveal the correlations between such performance counters as input rate, throughput and CPU utilization, or I/O-related database counters. The association rules are derived from the frequent itemsets. If application of these rules exhibits different behavior, they are reported to the analyst for further investigation.

The key benefits of this approach are the following. First of all, it allows to work efficiently with a large number of performance counters. While manual testing assumes an analyst selects a small number of counters for further investigation, machine learning algorithms used in the proposed approach may generate composite performance signatures out of many different correlated parameters thus providing with more detailed results.

Secondly, this approach does not necessary build a baseline out of a single prior run. Instead, a performance repository is used. This repository may be formed out of many prior runs thus providing more reliable results.

The discussed technique is rather easy to implement and understand. It provides the analysts a clear overview of performance degradations in a form of visual performance report. Thus, it can be recommended to apply this technique for detecting performance degradations in the enterprise software used in the industry. This thesis continues Foo’s work and further develops the ideas initially presented in [19].
3. LITERATURE STUDY

3.2.3 Identifying performance regressions using control charts

Nguyen et al. in [39] have proposed a simple and rather efficient way to detect performance regressions in web systems using a statistical process control technique called control charts. The control charts are a common statistical control tool which is used since 1920s to detect deviations in manufacturing processes. The authors of the discussed paper propose to use these charts for detecting regressions in the load patterns of the software products.

Working with the control charts usually involves two stages, namely calculating control limits using the baseline dataset and applying them to the target dataset. As a result, an analyst retrieves a metric called violation ratio which shows to which degree the target dataset is out-of-control if compared to the baseline.

This technique imposes some limitations on load testing scenario and behavior of the system. Thus, it is assumed that input data is stable within the test runs and the performance counters have uni-modal normal distribution. Usually, both of these requirements are not satisfied in performance testing. It is possible to mitigate them, but it requires additional measures and is not always successful.

In general, the control charts technique is easy to understand for developers who are not performance testing specialists. The metric it reports is clear and intuitive, the practitioners note they can easily communicate the performance testing results with the rest of the development team not diving into the details of the underlying statistical model. Thus, the suggested control technique can easily be adopted in the industry.

3.2.4 Other approaches to performance degradations detection

3.2.4.1 Configuration space exploration

Yilmaz et al. in [48, 49] approach the performance testing problem with an idea to reveal the performance degradations by exploring the application’s configuration space. The researchers claim that due to the time pressure the performance analysts are limited by testing the target application with a small number of configurations and have to extrapolate the obtained results for the whole configuration space. This may lead to ignoring many performance bottlenecks until the application is deployed. These bottlenecks arise because of different reasons. For instance, a customer may run the application with its configuration different from the testing setup because of using specific hardware platform.

The described approach has a number of drawbacks. First of all, it is based on a human decision at the initial step. Among all the configuration options the performance engineers should choose a subset of options they consider important for the system performance. All the further space reduction operates with this initial estimation. Secondly, the described process is rather complicated for understanding and resource-intensive. Skoll environment used as a technical platform for experimenting with the underlying quality assurance process is not freely available.
3.3. Summary

3.2.4.2 Execution log analysis

Another approach to detecting performance degradations involves the analysis of the application’s execution log. The major benefit of techniques using this approach is that they can be applied when performance is tested for an already-compiled software suite, usually provided by a third-party vendor.

Aguilera et al. in [4] describe a method for performance testing distributed black-box systems. Their approach relies on passive tracing of communication between the nodes in the distributed system and further analysis of the obtained traces. While analyzing the traces, the researchers detect causal relationships between the different requests and calculate processing latencies at computational nodes using the difference in timestamps between the related events in the log. The main challenge of this approach is to devise proper algorithms that would find causal relationships in the log. To solve this problem, two algorithms have been implemented. The nesting algorithm combines all the traces into a single global trace and examines its entries to find out how the calls are nested in it. This algorithm requires RPC-style communication between the nodes. The convolution algorithm finds causal relationships by considering the aggregations of messages. Each of per-edge traces is treated as a time signal and to find correlations between the messages, signal processing techniques are applied to them. Thus, this algorithm can be applied in situations where the traces are described in free form. However, as the nesting algorithm has more knowledge about the structure of the communication, it provides more concise representation of the tested systems.

Jiang et al. in [25] have developed a five-step process to automatically detect performance problems from load testing results and organize the findings in a visual report. The first step of the process is to convert raw data from the log into a set of execution events. Then, the execution sequences are extracted from this set and performance of these sequences is calculated based on the difference in timestamps between the first and the last events in it. At the third step, the execution scenarios are identified from the repeating sequences and the performances of these scenarios are summarized for previous and current runs. Then, the performance of the scenarios between the runs is compared using statistical tests. The scenarios with statistically different performance between the runs are selected for the report. The report itself is generated at the last step. The scenarios in it are ranked by the degree of performance deviation between the prior runs and a current one.

A serious drawback of the empirical approaches discussed in this section is that they rely on the structure of the execution log. While in many situations the available log is detailed enough to be used as a data source, sometimes it does not contain enough information and thus cannot be used in the analysis. Also, for the second approach it is required from the engineers to analyze the log manually in advance to find the parameters used to link the log files to the scenarios.

3.3 Summary

In section 3.1 the recent approaches used to generate realistic workload traces for workload generators were reviewed. In most of the covered studies their authors address the problem
by developing analytical models using stochastic processes to create virtual users that generate requests to the web pages in concordance with their preferences (user equivalents approach, the UEs). Another analytical approach does not treat web traffic as super-position of traffic generated by individual virtual users but rather attempts to generate aggregated traffic directly (aggregate traffic generation, the AGT). However, this approach as well as the attempts to create a hybrid approach are not widely used in the research community as they are more complex in comparison with the UEs and do not offer significant improvements over the solutions created using the user equivalents. Those approaches that do not model web traffic analytically but generate the traces based on the analysis of the execution log also tend to underperform UE-based approaches due to their simplicity and worse flexibility.

In section 3.2 a review of recent statistical approaches to automatic detection of performance regressions in evolving software systems was performed. Most of the techniques addressing the discussed problem were proposed very recently and to the best of our knowledge, neither of them is widely adopted for industrial use nowadays. While a number of reviewed approaches to automatic detection of performance regressions report promising results, a lot of work still has to be done to verify their applicability for different classes of applications and create production-ready software packages supporting them. Out of the reviewed techniques the methods involving association rule mining ([18, 19]) and control charts ([39]) seem the most promising and the best suitable for detecting performance regressions in a wide range of software applications. Both techniques are easy for understanding by software developers not having much knowledge about the performance testing processes. Introduction of a production-ready performance analysis system based on one of these approaches would be beneficial for many businesses that need in accurate evaluation of the performance characteristics of their software.

The other techniques reviewed in this chapter have severe drawbacks that limit their applicability and are less suitable for general use in the industry. Thus, configuration space exploration approach proposed by Yilmaz et al. in [48, 49] is very complex and is not applicable for testing web systems, as they are usually deployed with a single configuration, and exploring the other configurations will not help with avoiding performance issues in the configuration deployed at production servers. The main drawbacks of the empirical approaches by Aguilera et al. [4] and Jiang et al. [25] is their dependence from the detailed execution log and rather low accuracy. The performance signatures approach described in [12, 36] is similar to the approach proposed by Foo et al. it [19] but is less automated, hence it is more complex and elaborate.
Chapter 4

Performance testing repository structure and functionality

The main goal of the graduation project performed by the author was to apply and extend existing statistical methods for automatic detection of performance degradations in web systems. The performance degradations in the evolving software appear over time and to analyze them it is required to track performance of target applications during certain time periods. For correct detection of performance degradations and estimation of their severity, it is needed first to find a performance baseline, i.e. such values of the performance counters that are considered normal for the scope of completed performance test runs. These values can usually be determined out of one or several test runs marked as reference by the performance analysts. After the baseline is found, it can be used to understand whether the results of a new test run are in line with the previous runs, and if not, how severe the performance degradations are.

Performance test results analysis still remains mostly a manual and elaborate process. A usual way to the performance evaluation of web applications does not involve a thorough data mining-based analysis of the previously completed test runs. Instead, the analysis usually consists merely of the visual comparison of the performance charts between the most recent and preceding test runs. While this approach is sufficient for the applications that do not depend significantly on moderate changes in performance during the software evolution process or have long development cycles, for the rapidly evolving software that has to operate under high load it can be insufficient. The engineering teams developing such software may benefit from introducing the automatic approaches to detect performance degradations.

4.1 Related software

As it was covered in the literature study, not much work has been done on automating performance degradations detection process until recently. For this reason, there exist no performance repositories aimed to organize historical performance data for further statistical analysis.
To the best of our knowledge, the only well-known software allowing to store historical data from the completed performance test runs and compare the runs between themselves is Microsoft Visual Studio (Ultimate Edition)\(^1\). This is an advanced programming environment that includes load testing and performance analysis functionality. The Visual Studio allows performance analysts to build and execute complex performance test plans, gather performance counter values from the computational nodes and analyze the results of the test runs. It supports creating a Load Test Results Repository\(^2\) to import and export performance test results. However, it does not include any methods for automated performance degradations analysis. Despite it is possible to create useful performance reports containing comparisons of certain performance counters between the test runs using the Visual Studio, this software does not decide whether the application performance has decreased and how important the changes in performance are. The performance analysts have to do a lot of work analyzing the performance reports to draw the respective conclusions.

Performance test analysis functionality in the Visual Studio has many benefits and this software is successfully used by many software development companies. However, for a number of reasons it cannot be used as a framework for building the performance repository for the statistical analysis of performance test results. First of all, this is proprietary software with closed-source code, which makes it impossible to extend its functionality in the required way. Secondly, it only supports test data in the Visual Studio format; it is impossible to import the data that is gathered with other performance testing tools. Also, the Ultimate Edition of the Visual Studio, which is the only edition supporting performance test analysis, is extremely expensive and it does not make much sense to purchase it for its performance testing functionality only.

The idea of a performance repository used to store the performance test results and analyze them for building performance baselines and detecting performance degradations using statistical methods was initially introduced in a scientific paper by Foo et al. in \cite{19}. In this paper, the authors have described mining a performance testing repository using association rules to reveal the performance degradations in the web systems. However, a working version of the performance repository was not provided. Also, the authors did not address many important details of its functionality. Thus, taking into account the importance of having a performance testing repository for maintaining efficient performance engineering practices, this graduation project involved building an open-source performance testing repository having the properties described in the following section.

### 4.2 Repository characteristics

For the reasons discussed earlier in this document, it was decided to develop an open-source implementation of a performance testing repository as a part of the graduation project\(^3\). The aim of this software is to support management and analysis of performance test results. The

\(^1\)http://www.microsoft.com/visualstudio/eng  
\(^3\)The implementation of a repository is available at https://bitbucket.org/dzzh/mining.
4.3. Technological stack

performance testing repository was built to assist performance engineers with the following tasks.

- Get easy access to the results of different performance tests ever run in the company.
- Convert the results of performance test runs from different performance assessment tools into a common format for further comparisons.
- Unify the values of the performance counters obtained from different nodes within the computational environment.
- Make initial estimates about performance of newly added test runs in comparison with the baseline results.

A provided repository’s implementation satisfies the following non-functional requirements.

- Open-source. The repository code was written using publicly-available open-source components and is opened for public changes.
- Multiple data import formats. The repository supports import of performance counter values in a number of different formats. Its architecture supports easy addition of the new formats.
- Web interface. The repository can be deployed online to let users access it from different places and support collaboration.

4.3 Technological stack

The performance repository was built in the Python4 programming language using the Django5 web framework and MySQL6 database. jQuery7 and Twitter Bootstrap8 frameworks were used for front-end development. All the software used to create a performance repository is open-source, easily accessible and can be used without limitations in the industrial setting. The repository itself is a WSGI web application that can be deployed online using Apache9 or other Python-compatible web server.

---

4 http://python.org  
5 http://djangoproject.com  
6 http://mysql.com  
7 http://jquery.com  
8 http://twitter.github.io/bootstrap/  
9 http://apache.org
4. PERFORMANCE TESTING REPOSITORY STRUCTURE AND FUNCTIONALITY

4.4 Repository structure and terminology

The performance repository operates with several different entity types. At the highest level of entities’ hierarchy stands a performance environment. The performance environment is considered to be a set of computational nodes that are logically organized to perform a single task, e.g. support a distributed web application.

Each environment consists of one or more computational nodes. These nodes may serve different purposes, e.g. function as back-end, front-end or database servers. It is assumed that the performance tests executed against the environment influence all the nodes belonging to that environment and do not impact the nodes outside the environment.

It is also assumed that the performance engineers build performance tests and launch them to test behavior of the environment at a whole. Most commonly, such performance tests consist of a series of web requests to the load balancer or directly to the computational nodes. The performance of each test run may be assessed by means of various performance counters that can be gathered using different software either at environment level (e.g. latency of the certain requests) or at node level (e.g. CPU utilization of a back-end server or a number of database writes at a database server).

Each environment may contain a number of tests. A test consists of a number of test runs. It is assumed that each test contains the test runs that were launched with the same parameters against different versions of the target software. Separation of the performance tests from the test runs allows the performance engineers to run different performance tests on the same application and compare the results between the various runs with ease. The performance engineers may describe the test configuration in a test description.

The main entity of a performance testing repository is a test run. Each run can be classified as either belonging to a baseline or target set. If a performance analyst classifies a test run as a baseline run, the values of the counters gathered during this test run are used together with the other baseline test runs to calculate a single baseline for estimating performance of the target runs. If a run is classified as belonging to a target set, its results are not used for the baseline calculations. After the performance analyst evaluates a new performance test run and does not find severe violations, he may change its status from target to baseline. The software allows to limit a number of baseline test runs used for the baseline calculations not to consider the results of testing legacy software. Limiting the number of baseline test runs may be useful after releasing a new version of software with significantly changed performance characteristics.

Each test run may include several different files with the performance counter values gathered during the test run. Their conversion to a common format is discussed in the following section.

The performance counters are defined separately and can be associated with a test run. One test run may have many different counters that are retrieved from different data files.

4.5 Performance counters

The performance counters are time series representing the results of monitoring certain components of a target system. In performance engineering they are used as measures
of system performance. It can be said that all the performance engineering processes are
directed to keep the values of the performance counters in line with expectations.

The goal of a performance repository is to store the performance counter values from
different test runs to make them available for further analysis and comparisons. These data
are retrieved from the external files created by third-party monitoring applications and is
stored in the repository in two different formats.

First of all, the source data extracted from the files is stored as it is. To support this
type of storage, so-called raw performance counters are used. These counters consist of
records, with each record used to store one value of a single counter with its associated
timestamp and a label. The labels are used to distinguish the values coming from different
computational nodes for node-related counters or the values belonging to different requests
for environment-related counters. The raw counters are not used directly for data analysis,
these activities are performed on the consolidated performance counters that are created out
of the raw counters.

The only difference between the data in the source files and its representation in a form
of raw performance counters is the time frame. While the source files may contain arbitrarily
large numbers of data points that are gathered during different overlapping time frames,
the raw performance counters only keep a subset of data within the boundaries set by the
performance analysts. It is needed to set the time boundaries for the test run start and
end, because the accuracy of further analysis may suffer from inconsistent behavior of the
performance counters during ramp-up and cool-down periods when not all the threads in a
load generator are working.

Raw counter values are not used for data analysis directly because of two reasons. First
of all, raw data has to be resampled. Such resampling is needed as typical input to a sys-
tem may consist of many raw files gathered from different computational nodes using var-
ious monitoring software. Usually, the counters in these files are gathered with different
sampling rates and even the same counters at different computational nodes have different
timestamps due to the clock skew and stochastic structure of the processes execution in
modern non-real-time operating systems. Thus, to correctly compare the different counters,
all of them have to be down-sampled with the same sampling rate.

Secondly, the continuous structure of raw counter values does not allow to apply ef-
ficient data mining algorithms to derive meaningful information out of them. For these
algorithms to work, the source data has to be classified in advance. Also, different per-
formance counters have their values in different ranges and it is incorrect to compare the
counter values based on this numerical and not-related data. Classification of counter values
allows to mitigate this problem.

For these reasons, the repository supports the second type of storage. This storage
operates with the entities called the consolidated performance counters which are the main
data source for the data analysis module. Consolidated counters are generated out of the
raw counters using the consolidation process discussed below.
4. PERFORMANCE TESTING REPOSITORY STRUCTURE AND FUNCTIONALITY

4.5.1 Data consolidation process

The goal of data consolidation process is to transform unrelated raw counter values into classified data. To achieve this, the data has first to be downsampled. The length of a sampling period should be set manually by a performance analyst and has to be larger than the lowest sampling rate for any of the counters. Say, for a test run lasting for an hour and containing the counter values gathered each 15 seconds or more frequently it makes sense to define a sampling period of 30 or 60 seconds. After the sampling period is set, raw data is split into the periods of this length. Each performance counter in a test run will then have as many values in its time series, as many periods of this length are in a test run time frame. All the values of a counter within each period are used to find a single value to be used in a new performance counter time series. To find this value, different strategies can be applied. The simplest ones are to use average or median values. As averaging is generally less accurate, median values are used for downsampling in the provided implementation of a performance testing repository.

Contrary to the raw performance counters, for the consolidated values there is no need to use timestamps to align the data points on a time scale. Instead, an offset is used. The first data point in a consolidated time series is assigned zero offset, for each consequent data point the offset increments. If for some of the sampling periods it happens that no respective raw counter values are available, a consolidated value is set to null, but is saved to the storage anyway.

After the raw counter values are downsampled, the median values with the associated offsets are stored in a database. Downsampled data don’t get classified before being saved. The real classification is done on the fly right before the analysis. The reason behind this is that the classification depends not on the values in the same target test, but on the baseline values that change with the baseline runs set. For example, let’s say the values of some performance counter in a target run are normally distributed between 10 and 20, and at a certain point in time a median value in the respective sampling period equals to 11. Then, in a system with Low, Medium and High classification categories this value will most probably be classified as Low, if the algorithm will rely on the values in the same target run only. If this category is fixed and saved in the database, it will only be known that at this point in time the counter had a relatively low value. This makes sense on its own, but this information will be meaningless if the results of a target run are compared with some baseline runs. Let’s say that the baseline values of a respective counter are distributed between 5 and 13. In this case, 11 will most probably be classified as High, not Low. Thus, storing the median values instead of the categories is more flexible and leaves more information. The classification itself is performed at the analysis stage. Its details are discussed in the subsequent section.

If there exist several baseline test runs, to calculate baseline median and deviation their values are combined into a single array for each raw performance counter.

4.5.2 Classification

To convert the consolidated values of the performance counters into the categories that would be easy to analyze with data mining techniques, the performance testing repository
uses classification modules (classifiers). Four interchangeable classifiers are available in the repository code. They are described in this section.

All the classification algorithms available in the code of the performance repository rely on raw counter data to calculate the category boundaries. It would be easier to use downsampled data of the baseline tests for finding these aggregated values, but after experimenting with these data it turned out that downsampling introduces certain loss of precision that has to be taken into account. Hence, it was decided to rely on raw data for baseline calculations to make the classification more accurate. This means that for the classification both raw and discrete performance counters are needed.

### 4.5.2.1 Median-based classifiers

The first two classifiers were built to calculate the categories of consolidated performance counter values based on the medians and standard deviations of baseline raw counters. A taxonomy with three categories, namely Low, Medium and High, is used in them.

To find a category of a value in a performance test run, the classification algorithm first calculates median and standard deviation for the respective set of baseline raw counter values. Then, the downsampled value in a run is compared to the baseline median. If the first (fixed-stddev) classification method is chosen, the algorithm classifies the values that lie within the predefined number of baseline standard deviations from a median (1 by default) as Medium, otherwise as Low or High depending on whether the target value is smaller or larger than the baseline median.

If the second classification method (fixed-percentage) is in use, the analyst has to specify a fixed percentage of values that are considered low or high in the baseline set. The algorithm then calculates the boundaries of a corridor allowed for the medium values based on this information and classifies the target data points as Medium only if they make it in a corridor. Otherwise they are marked as Low or High the same way as in the previous method.

The experiments performed with these two counters have shown that they lack precision are are not well suitable for performance degradations detection. Thus, most of the baseline counter values get into Medium category, yet only a small fraction of them is classified differently. The target values are classified as Low or High only if severe performance violations exist in the run, otherwise most of the values are also classified as Medium. This makes it impossible to use these classifiers for detecting performance regressions that are not severe enough to be immediately detected during the manual analysis. To solve this problem, two more accurate classifiers were developed.

### 4.5.2.2 Quartile-based classifiers

These classifiers are more suitable for performance degradations detection as they operate with more categories and classify the values based on quartile points [45]. Both these classifiers use a taxonomy with such categories as Low, 1st, 2nd, 3rd and High. Low and High categories are used to mark outlier values, 1st category is used for values that reside in the lower quartile ($Q_1$), 3rd category is used for values that belong to the upper quartile ($Q_3$)
and 2nd category is used to classify the values lying in the interquartile range (i.e. between $Q_1$ and $Q_3$).

A simpler of two quartile-based classifiers is called quartile. It only uses the information about the quartile points to classify data values. To find outlier boundaries, it uses the smallest and largest values from the baseline. Thus, it is assumed that the baseline does not contain outlier values and they can only appear in the target. The second classifier is called quartile-stddev and uses median and standard deviation to calculate the boundaries for Low and High categories. At first, the classification algorithm used in this classifier calculates the median and standard deviation of the baseline values. Those values that do not get into the interval $\text{median} \pm 1 \text{ stddev}$ are considered outliers and are classified as Low and High the same way as they are classified in fixed-stddev classifier. Then, the remained values (those that fall into Medium category for fixed-stddev classifier) are classified using quartile-based classification the same way it is performed in quartile classifier.

The need to develop quartile-stddev classifier emerged because of the drawbacks of the classifiers developed beforehand. As it was mentioned earlier, median-based classifiers lacked precision in detecting relatively small performance degradations, as most baseline values were falling into a single category. A drawback of quartile classifier is that it assumes that the baseline does not have outlier values, that turned out to be a wrong assumption. quartile-stddev classifier properly distinguishes regular values from the outlier and provides three categories for the regular values thus having sufficient precision for performance engineering tasks.

The structure of a performance repository allows to easily add new classification algorithms that can better suit the needs of the performance analysts. The classification problem has many approaches and some of the available classification algorithms (for instance, those using the distribution properties of a baseline set) may show better results. However, the experiments performed during this research project and described in case studies (see Chapters 7 and 8) indicate that quartile-stddev classifier works well in performance degradations detection tasks.

4.6 Basic analysis tools

The performance repository supports a simple technique for estimating the performance of a newly added test run. This technique is not as advanced as the methods implemented in the analysis module of the project, but it may also be helpful for estimating performance of the target runs.

If the performance of a new test run has decreased significantly, it can be easily detected either from the performance charts or by finding violation ratios for the available raw performance counters. The violation ratio here is defined as the total number of outliers (the values classified as Low or High) divided by the total number of samples.

To help the analysts to easily detect broken test runs, each test run page contains a table listing all the available performance counters with such aggregate statistical metrics as
minimum, maximum, average, median and standard deviation as well as their comparison with the respective metrics calculated over a baseline set. Also, the tables contain the violation ratios (separately for Low and High values). The table is sorted by the High violation column in descending order, which helps to overview the most severe problems in a run.

For the situations when the violation ratio is not very high but still suspicious, it may be useful to analyze the data by visually comparing the counter behavior in the target run with the baseline runs. To do this, the repository provides two different charts, namely a boxplot and a time-series plot. Both these charts are available for raw and discrete counters. It is recommended to use the charts based on the discrete counters for the analysis, but the raw charts can sometimes also be useful to check whether data consolidation leaded to significant loss of precision, which is an unlikely but possible case.

4.7 Workflow

This section describes what it looks like to work with the provided implementation of a performance testing repository from the prospective of a performance analyst.

At first, the user has to define a testing environment. This step consists of setting its name and describing the computational nodes that form the environment and their roles in it. After this step, the user may have a look at the performance counters that are available in the system and add more counters if there is such a need. Then, the user should create a new test and specify its parameters in the description. This is needed for the situations when there exist many tests with slightly different parameters, i.e. different number of running threads or different state of caches. Also, the user has to set a length of a sampling period that will be the same for all the test runs.

When the test description and sampling period are set, the user has to launch his performance tests and gather the values of the performance counters. When these values are measured, the user should create a test run and set its start and end time as well as to specify whether the test run is used to form a baseline or has to be used as a target run.

After setting the test run parameters, the user may add data files containing the values of performance counters to it. After a test file is added into a system, the analyst chooses its format and selects the performance counters that
can be found in the file. After all the preparations are done, the data extraction algorithm retrieves raw performance counters out of the test file and consolidates them.

When raw and consolidated performance counters are created, they can be used by the analysis module. Also, the analyst may observe the statistical metrics and time-series charts at test run details pages to quickly detect the broken test runs. If certain counters were tracked incorrectly or their values significantly differ from the expected ones, they can be disabled for some time to let the analysis module compare the runs without considering these counters.

The analyst adds as many performance test runs as it is needed. Each target run can later be compared with the previously defined baseline. This baseline is formed out of the runs that have satisfactory performance, as estimated by the analyst.

Some interfaces of a performance repository are shown at Figures 4.1 – 4.5.
Figure 4.4: Test run details.
4. PERFORMANCE TESTING REPOSITORY STRUCTURE AND FUNCTIONALITY

(a) Format selection

(b) Counters selection

Figure 4.5: Data files processing.
Chapter 5

Application of association rule mining for performance testing

The main goal of a performance testing repository discussed in the previous chapter is to store aggregated values of the performance counters. This should allow performance analysts to track changes in performance of target applications during their lifecycles. Despite the fact that raw performance data can be useful on its own and certain conclusions about the quality of a software package under study can be made after applying the simple analysis methods discussed earlier, there exist situations when performance degradations are not obvious and difficult to detect with simple heuristics. To reveal the performance degradations in these cases, certain data mining algorithms can be used. In this chapter the applicability of association rule mining algorithms to performance testing datasets is discussed.

5.1 Classification of data mining algorithms

As it was pointed out earlier, the analysis of performance test results remains to a large extent manual and highly inefficient process, though its tasks may have be automated. Many data analysis algorithms were developed recently and some of them may be efficient for the analysis of performance testing data stored in a repository. Deriving knowledge from patterns occurring in the large volumes of data is known as knowledge discovery in databases (KDD) and to find the algorithms suitable for mining performance testing data, this topic has been studied attentively. KDD is an important and broad research field that has much attention from the research community and covers a rather large set of tasks nowadays. Fayyad et al. in [17] divide the data mining tasks used in KDD process into six different classes, namely classification, regression, clustering, summarization, dependency modeling and change and deviation detection. Many of these classes cover the problems similar to ones to be solved while analyzing performance tests.

The main problem of mining performance testing datasets in the scope of the discussed project can be stated as finding the degree of similarity between the datasets containing performance observations. The performance data that is stored in the repository either belongs to a baseline dataset or to a target dataset. To conclude whether a target run was successful
or not, it is needed to compare the values of the performance counters gathered during the
target run with the values of the respective baseline runs. If the values and their distribu-
tions are similar, the target run may be considered having satisfactory performance. If the
baseline and target datasets differ significantly, this means that the target run has different
performance and this difference has to be reported for further investigation.

The degree of similarity between two datasets can be found in several different ways.
First of all, it is possible to apply the classification algorithms to mark each of the trans-
actions in a target data set as successful or failed and make conclusions about the run at a
whole based on the percentage of failed transactions. Secondly, change and deviation de-
tection algorithms can be used to discover the most important changes in a target set. This
information can later be used to determine whether this dataset differs significantly from
its baseline or not. The third possible way of comparing the datasets is with dependency
modeling algorithms. These algorithms build a dependency model that describes significant
dependencies between the variables. To find the degree of similarity between the datasets
using dependency modeling, it is needed to derive a dependency model for a baseline set and
apply it to the target dataset. If the target dataset fits the model, this means that it is similar
to the baseline. Otherwise, baseline and target datasets have to be considered dissimilar.

5.2 Association rule mining

Due to the large amount of work needed to be done it is not feasible to implement and
compare all the above-mentioned approaches within a scope of a single graduation project.
Hence, this study addresses only the applicability of dependency modeling algorithms on
mining performance testing data. Later in this document dependency modeling is referred
to as association rule mining. This term was introduced by Agrawal, Imielinski and Swami
in [2] and is commonly used for the dependency modeling algorithms nowadays.

Association rule mining techniques were initially designed for the market basket anal-
ysis tasks. By early 1990s, improvements in barcode and data storage technologies made it
possible for large retail chains to keep records of every combination of items ever purchased
by the customers and update these records in near real-time. Processing of customers’ trans-
actions may help to find important patterns in their behavior and use them to improve the
effectiveness of marketing campaigns, merchandise and sales planning. These activities
eventually lead to higher profits for the companies that employ data analysis techniques to
find knowledge in their data. For this reason, a lot of attention was paid to market basket
analysis and association rule mining over the last two decades. Several important algorithms
addressing the problem were developed and adopted for various scenarios. Apart from be-
ning used for market basket analysis, these algorithms are successfully applied in network
security and computational biology as well as in time-series analysis now [23].

5.2.1 Formal definition

The association rule mining problem can be stated as follows [2]. There exist a set of
items $I = \{a_1, a_2, ..., a_n\}$ and a set of transactions $T = \{T_1, T_2, ..., T_m\}$. Each transaction
contains an itemset from $I$, i.e. $T_i \subseteq I$. An itemset containing $k$ items is called $k$-itemset. An
association rule is defined as an expression $X \Rightarrow Y$ where $X$ and $Y$ are itemsets from $I$. This expression means that if $X \in T_i$, then probably $Y \in T_i$. The rule $X \Rightarrow Y$ has support $s$ if $s\%$ of transactions in $T$ contain $X \cup Y$. The rule holds with confidence $c$ if $c\%$ of transactions in $T$ that contain $X$ also contain $Y$.

Having a transaction database $T^1$, the problem of mining association rules is to find all the rules having support and confidence higher than the user-defined thresholds. These thresholds are called minimum support (minsup) and minimum confidence (minconf). The itemsets having at least minimum support are called large or frequent.

### 5.2.2 Problem decomposition

Association rule mining problem is usually decomposed into two subproblems. First of all, it is needed to find all the frequent itemsets exceeding the minimum support. Secondly, when these itemsets are found, they are used to generate association rules satisfying the minimum confidence constraint. The second subproblem has a straightforward solution, when the rules are generated by iteratively moving the items from the rule antecedent to its consequent and checking the confidences of the generated rules for each frequent itemset. For this reason, the association rule mining problem can be reduced to finding all the frequent itemsets satisfying the minimum support constraint.

### 5.2.3 The Apriori algorithm and its variations

One of the most important algorithms for association rule mining is Apriori described by Agrawal and Srikant in [3]. Since its introduction, a lot of papers discussing Apriori variations and adaptations for different use cases were presented.

Apriori is a level-wise breadth-first algorithm that employs candidate sets generation and utilizes the downward closure property of itemset support. This algorithm works in the following way [3]. During the first pass, it calculates supports of every single item occurring in a database. Those items that have support larger than minsup are considered large 1-itemsets. All the subsequent passes consist of two phases. During the first phase, large itemsets $L_{k-1}$ found at the previous pass are used to generate candidate itemsets $C_k$. During the second phase, the support of the candidate itemsets is calculated and only those itemsets that have support larger than minsup are added to $L_k$. The algorithm continues until there are no new itemsets having support that is large enough. At the end, all large itemsets $L_k$ are combined to form the final result.

The downward closure property of the Apriori algorithm means that any subset of a large itemset must be large. Because of this property, it is only sufficient to generate candidate itemsets at pass $k$ by joining frequent itemsets obtained at pass $k-1$ which is less computationally intensive than bruteforceing all existing $k$-itemsets.

Limiting the number of candidate $k$-itemsets and the relatively fast pruning procedure responsible for removing the candidate $k$-itemsets not having sufficient support make Apri-

---

1For association rule mining to work, underlying datasets may be presented in any machine-readable format. In this study it is assumed that the transactions are logically organized in a form of a single-table SQL database.
ori an effective association rule mining algorithm for market basket analysis. Apriori has satisfactory performance even if processing large datasets containing tens of thousands items and millions of transactions.

However, the Apriori algorithm has a number of drawbacks. One of the most important ones is the need to scan the whole database at each pass. To overcome this drawback, several algorithms reducing the number of database scans were developed. One of such algorithms is AprioriTid, described together with Apriori in [3]. In original Apriori, the support of each candidate itemset is calculated out of raw data contained in the transactions. AprioriTid associates the transactions with the itemsets contained in them and calculates supports for the candidate itemsets based on these data. AprioriTid is more effective than Apriori for mining relatively small datasets that contain a number of frequent itemsets of medium to long size. Apriori is more efficient when the frequent itemsets existing in the database are short\(^2\). AprioriHybrid [3] is a combined algorithm that switches between Apriori and AprioriTid implementations in different passes and usually has better performance than Apriori and AprioriTid on their own.

Another drawback of Apriori not allowing to effectively utilize this algorithm for certain data mining tasks is that Apriori often generates way too many short association rules that are not always interesting. If the longest frequent itemset existing in the database is rather long (ten items or more), the total number of association rules will be extremely large, because all the combinations of frequent items from the longest itemset are frequent itemsets themselves, and each of these itemsets can be used to generate several different association rules. This makes Apriori and its variations rather inefficient for detecting relatively long patterns as Apriori is a top-down algorithm that generates the shorter itemsets prior to the longer ones. AprioriTid and some other Apriori variations may be more efficient in long patterns detection, but their applicability for this task remains rather limited anyway.

There were many attempts to improve performance of Apriori-based algorithms. The most notable techniques include reducing the number of database passes, database sampling, imposing additional constrains on generated patterns, organizing items into taxonomies and the algorithm parallelization [23, 28]. Despite these optimizations are rather efficient when working with market basket-like data, they do not overcome architectural limitations of the Apriori algorithm and thus do not help much with mining long rules.

5.2.4 The FP-growth algorithm

Taking into account the above-mentioned drawbacks of Apriori-like algorithms, Han et al. have developed an entirely different approach to association rule mining that does not use candidate itemsets generation. It was introduced in [21] and is known as frequent-pattern growth (FP-growth). Contrary to Apriori and its variations, the FP-growth is a depth-first algorithm that scans the database only twice. It stores the needed intermediary information about the data in a condensed and small data structure called FP-tree. The FP-growth is not the first developed depth-first association rule mining algorithm, but it is the most well-known one.

\(^2\)It is assumed here that short itemsets do not contain more than five items.
The FP-growth has three important features that distinguish it from Apriori-like algorithm [20, 22]. First of all, it maintains groupings of original data in the FP-tree. Secondly, the FP-growth uses divide-and-conquer technique to partition the dataset and the set of patterns to be examined. Thirdly, it avoids costly generation of candidate itemsets.

The FP-growth algorithm for finding frequent itemsets consists of two steps [22]. In the first step, a source transaction database is compressed to an FP-tree. In the second step, this structure is used to detect the itemsets.

The first step of this algorithm contains two database scans. At first, the database is scanned to derive a list of frequent items with their supports. Then, the items are sorted in decreasing support order. An empty tree with a null root is created. After these preliminary steps, the algorithm scans the database for the second time and for each transaction it creates a path in the tree going from the item with the highest support to the least frequent item with the lowest support. For each node in the tree, the algorithm keeps item name, count of item occurrences in the respective path and a node-link. If two transactions share the same path, either completely or partially (i.e. have common prefix), the algorithm increases count value for each of the shared items.

Simultaneously with building the tree, the algorithm creates an item-header table where each item is pointed to its first occurrence in the tree. Nodes with the same item name are organized into linked lists via node-links.

The resulting FP-tree is very compact for most databases, but still contains all the information about the transactions that is needed for association rule mining. Because of the node-links, it is possible to retrieve all the transactions containing a certain item very fast.

After the FP-tree is built, the algorithm derives the frequent itemsets from it using the following procedure. For a database with \( N \) frequent items, the whole space of potential frequent itemsets is divided into \( N \) parts. The first part contains only the itemsets having item \( a_1 \), the second part contains the itemsets having \( a_2 \) but not having \( a_1 \) and so on up to the \( N^{th} \) part containing only the itemsets with \( a_N \) and no other items. Each of these parts can be traversed the following way.

Using the item-header table and the node-links, the algorithm retrieves from the FP-tree all the transactions in which \( a_i \) participates. The prefixes of the paths where \( a_i \) occurs form a sub-database which is called \( a_i \)'s conditional database. This database is used as a source for a new and smaller FP-tree. This process continues recursively until the built conditional FP-tree is small enough for direct traversal. All combinations of its components are the valid frequent itemsets for the sub-database containing the transactions with \( a_i \) element. To get the final result, it is needed to merge all the frequent itemsets obtained during the algorithm execution into one set. During the derivation of sub-databases, there is no need to include the already traversed paths for the following items.

### 5.3 Performance testing challenge

Foo et al. use the Apriori algorithm for automatic detection of performance degradations in web systems [19]. However, taking into account the drawbacks of Apriori that were discussed earlier in this chapter, it is doubtful that this algorithm is efficient for data mining.
5. APPLICATION OF ASSOCIATION RULE MINING FOR PERFORMANCE TESTING

Performance testing data in industrial setting. Apriori is suitable for mining large datasets for short association rules, but the organization of performance data is usually different and due to some of its properties it is hard to manage by the Apriori algorithm.

5.3.1 Organization of performance data

Contrary to market basket data where transactions have a random structure, the structure of transactions in a performance testing database, as it is implemented in a performance repository, is rather formal. The market basket transactions may be of different length and usually contain almost entirely random combinations of items from a large item set. A transaction in a testing database contains a list of performance counters that are tracked for an application and a category for each of the counters. Each transaction contains all the available counters but their categories differ. The length of the transactions is the same. An example of a transaction database containing classified values of performance counters is presented in Table 5.1.

<table>
<thead>
<tr>
<th>TID</th>
<th>CPU util.</th>
<th>DB reads</th>
<th>DB writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>2</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>3</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>4</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table 5.1: An example of a performance testing database.

5.3.2 Impact of data organization to mining algorithms

Organization of performance testing data allows to make certain assumptions concerning the association rules that are contained in performance datasets. First of all, the transactions in the datasets are usually rather long. More performance counters are tracked for the applications, the longer transactions are created in a system. This means that the database may contain very long association rules that are extremely difficult to calculate if following top-down level-wise breadth-first traversal of a search space used in algorithms from the Apriori family. In other words, it can be said that contrary to market basket datasets generated by the retail chains, performance testing datasets are non-sparse. This means that most association rule mining algorithms should not perform well in analyzing them, as such algorithms are mainly designed to solve different types of problems [46]. When decreasing minimum support, the number of existing frequent itemsets in non-sparse datasets grows much faster than in sparse datasets [50].

Also, the performance of the association rule mining algorithms depends significantly from the chosen approach to classify the raw data. If a selected taxonomy contains a small number of categories, then many baseline transactions will be very similar and this will lead to appearance of one or more extremely long itemsets. Such itemsets negatively affect the algorithm performance as they contain a huge amount of smaller itemsets that have to be
processed. On the other hand, if the classification algorithm operates with many different categories, the resulting frequent itemsets should be smaller and have smaller support. The problem with this approach to the classification is that certain groupings of items can be lost. Several implementations of a classification algorithm were discussed in the previous chapter and a *quartile-stddev* classifier was selected as the basic classification algorithm\(^3\). This classifier works reasonably well for performance testing datasets which has been proven by the case studies discussed later in this document.

Another important observation is that due to the relatively small number of aggregated transactions contained in a test run, minimum support value has to be rather high — starting from several percents and moving up. Otherwise the results will suffer from noise patterns that are not observed in most part of data and are introduced mainly because of fluctuations not related to behavior of the software under study.

Due to the structure of data, the items contained in a performance database can be treated in different ways. During the frequent itemsets generation, each combination of a performance counter and a category associated with it is considered as a separate item. E.g. counter *CPU utilization* with value *Low* and the same counter with value *Medium* are treated as two entirely different items. This approach to data processing allows to use association rule mining the same way this technique is applied in market basket analysis, as all the items will be considered not related in any way. The drawback of this approach is that it conceals additional information contained in the data, namely in its structure. It is known that each transaction should contain exactly one copy of each item with a value from a taxonomy and use of this restriction should help during the analysis of the generated association rules. Thus, despite the fact that information about the structure of performance testing data is not needed during the frequent itemsets and association rules generation, it has to be kept with the items at these steps. To do this, the counter name and its category have to be encoded using a two-way hashing function before the generation and after it the relations have to be decoded back. The details of this process are discussed in the following section.

### 5.3.3 Items hashing

Before applying the association rule mining algorithms discussed in the following chapter, the classified items presented in the transactions need to be converted to integers using a simple two-way hashing function. After the association rules are derived, the original counter-category relations are restored. These relations are not used during the association rules generation but are needed at the later analysis steps. Thus, they have to be kept with data during the whole analysis process. Storing these data as integers is a performance optimization aimed to speed up items’ comparisons.

To keep additional information about the performance data structure in the converted items, it was needed to find a hashing function that would be aware of it and allow to recover this information during decoding. For this reason, the following hashing function was chosen to encode data: \( e = \text{counter} \ast \text{N} + \text{category} \). In this function, *item* is an id of a counter in a database, *category* is an id of a category in the taxonomy used for the counter

\(^3\)See Section 4.5.2.
and $N$ is a multiplier exceeding number of categories (10 was used as a multiplier in the experiments for this study).

To decode data, $e$ has to be divided by $N$. After the division, counter id is the quotient and category id is the remainder. After applying this hashing function, the resulting values are ordered first by the respective counter ids, then by category ids.

### 5.3.4 Datasets comparison challenge

As it was stated earlier in this chapter, the main problem that has to be solved during the analysis of performance testing datasets is finding the degree of similarity between them. To decide whether a target run resembles baseline runs with association rule mining, it is needed to find patterns in both datasets and compare them. For the performance testing use case, the most important task during the comparison is to find the patterns that appear in the baseline dataset, but are missing in the target, or appear in both datasets but have different characteristics. Those patterns that appear in the target run but are missing in the baseline runs are less interesting, as they help to answer whether the baseline resembles the target, not vice versa.

Existence of two different datasets imposes a certain challenge, as usually the problems to be solved with the association rules involve only one dataset. After the patterns are found in it, they are reported to the analysts and data mining ends. If there are two datasets, the patterns have to be detected in both of them and compared afterwards. Apart from being important for performance testing, this problem appears e.g. in market basket analysis when there is a need to analyze how the customers’ behavior changes over time or at different locations. However, to the best of our knowledge, not much work was done previously on finding differences in patterns appearing in transaction databases. Thus, Tansel and Ayan refer to this problem in [43] but do not suggest an algorithm for comparing large sets of frequent itemsets or association rules. Foo et al. in [19] discuss a way to detect performance degradations using association rules generated with the Apriori algorithm. In their approach, the researchers derive association rules for the baseline dataset, apply them to target dataset and make conclusions on the validity of the rules based by degree of deviation between the correlation confidence for baseline and target runs.

In the next chapter a new algorithm for mining performance testing datasets is presented. This algorithm compares maximal frequent itemsets to quickly find the similarities and differences between baseline and target runs in a performance test. After the most significant differences between the datasets are found, they are used for association rules generation. These rules are further analyzed to form a list of violating performance counters that are presented to the analyst.
Chapter 6

Dasha: a performance testing mining algorithm

In this chapter a new data mining algorithm called Dasha is presented. This algorithm compares the patterns existing in transaction databases with association rule mining techniques. In Dasha, the datasets are initially compared using frequent itemsets that are derived from them. Only after the itemsets are compared and their difference is obtained, this difference is used to construct association rules for the further analysis. This two-level comparison approach based on traversal of lexicographic tree structures allows to compare large performance testing datasets in reasonable time and is suitable for usage in industrial setting.\(^1\)

6.1 Dasha overview

Taking into account considerations listed in the previous chapter, the Dasha algorithm was designed with the emphasis on performance. As it was pointed out earlier, the number of association rules extracted from the performance databases can be extremely large. Most of these rules are not interesting and are contained in longer rules. Their evaluation leads to significant time consumption even for rather small databases. To mitigate this problem, data analysis in Dasha is initially performed not on the association rules, but on the frequent itemsets.

In the proposed algorithm a tree structure to store the information about the frequent itemsets and their supports is used. This tree serves as a caching layer for itemsets-related information and implements a data structure initially described by Rymon as a set-enumeration tree in [41]. Prior to our study, set-enumeration trees have already been used for association rule mining by a number of researchers (see e.g. [1, 7, 10]). However, in the preceding studies such trees were mainly used to store information about the transactions contained in the source datasets for further generation of frequent itemsets. Agarwal et al. in [1] use set-enumeration trees to store itemset-related data, but their implementation significantly

\(^1\)The implementation of the Dasha algorithm as a web application including the performance repository is available at https://bitbucket.org/dzzh/mining. A stand-alone algorithm’s implementation is available at https://bitbucket.org/dzzh/python-dasha.
differs from their approach presented in this paper. A contribution of our approach is an idea to construct a set-enumeration tree from the already retrieved frequent itemsets and use the resulting data structure for quick traversal of the itemsets, retrieval of their supports and fast comparison of two sets of frequent itemsets.

Later in this paper the set-enumeration tree structure used for storing frequent itemsets is referred to as the support tree, because it stores supports for the combinations of items existing in the datasets under study.

The Dasha algorithm to detect performance degradations consists of the following steps.

1. Build the support trees for baseline and target datasets.
2. Compare the support trees using level-wise top-down approach.
3. Derive the association rules from the different frequent itemsets.
4. Analyze the association rules and make conclusions.

In the following sections these steps are described in more details and the properties of the resulting algorithm are explored.

6.2 Sample datasets

To illustrate the flow of the Dasha algorithm and the data structures used in it, throughout this chapter the algorithm’s execution is demonstrated on a simple example. In this section the source data used for it is described.

Assume a performance analyst wants to find difference in performance between two test runs. The first run is performed on a stable version of the software and has satisfactory performance. It is considered to be a baseline. The second run is performed on a new version of the software and its performance has to be evaluated. Both test runs consist of six observations, during each observation four performance counters are tracked. The data is classified and stored in a performance repository. Thus, each counter value is associated with a category, either Low (1), Medium (2) or High (3). The counter values are hashed to improve the algorithm’s performance. As a result, each counter observation in the datasets has value between 11 and 43, where tens mean the ids of the performance counters, ones mean categories. E.g. 12 means Medium value for counter c1, 43 means High value for counter c4. The performance datasets contents after these optimizations are presented in Table 6.1.

As this example is trivial, the conclusions about differences in data can be made manually. First of all, it is obvious that counters c1 and c3 are stable in both datasets. In every observation they have Medium value that is encoded as 12 and 32 respectively. Counter c2 has High value in two observations in the baseline dataset and in three observations in the target dataset. It has Medium value in four observations in the baseline dataset and in three

---

2 See Section 4.5.2 for more details about the classification algorithms.
3 See Section 5.3.3 for more details about the hashing function.
observations in the target dataset. Counter c4 behaves similarly to c2 in the baseline but has different behavior in the target dataset. It has Low value in one observation, Medium value in three observations and High value in two observations. Depending on the algorithm’s settings (i.e. minsup and mindiff) these differences can either be reported as violations or not.

6.3 Support tree structure and generation

Before discussing the algorithm’s implementation, the support tree that is an important data structure used in Dasha has to be described.

This tree stores the information about the frequent itemsets existing in the studied database and their supports. It is assumed that the items within a frequent itemset do not repeat and there is total ordering \(<_O\) of the items in the database. If item \(i\) occurs before item \(j\) in the ordering, this is denoted by \(i <_O j\).

The support tree closely resembles the FP-tree used for frequent itemsets generation in the FP-growth algorithm described in [21][4], but differs from it in the following aspects.

• The FP-tree is used to organize a list of transactions. The support tree is used to organize a list of frequent itemsets.

• The support tree does not have a concept of an item-header table.

In other aspects, the support tree resembles the FP-tree and can be built following the procedure similar to building the FP-tree. Total ordering in the support tree is also implemented similarly to the ordering used in the FP-tree. Namely, the items in an itemset to be added to the support tree are first sorted by their frequency in decreasing order, then naturally in increasing order. Later in this document this ordering is referred to as FP-ordering.

More formally, the support tree can be defined as follows\(^5\).

\(^4\)A short introduction into the FP-growth algorithm and the FP-tree data structure is provided in section 5.2.4.

\(^5\)Tree definition and algorithm’s description are based on the FP-tree description as presented in [21].
Definition 1 (Support tree). A support tree is a data structure defined below.

1. Support tree is a tree structure. It consists of a root node containing null as an item and a set of item-prefix subtrees as the children of the root.

2. Each node in the item-prefix subtree contains an underlying item, support value, link to a node parent and a set of links to node children.

3. Support value of a node is used to store the support of an itemset that appears in the respective dataset and includes the node and all its parents up to the root.

4. For two nodes \( i \) and \( j \) in the support tree it holds that if \( i.parent == j \), then \( j.item <_O i.item \) and \( i.support \leq j.support \).

Based on this definition, a support tree can be constructed using the following algorithm.

Input: a list of itemsets.
Output: an instance of a support tree.

Initialize a support tree with a root \( R \) associated with null item. For each frequent itemset in the input list do the following.

1. Sort the items appearing in the frequent itemset in the FP-order using \( <_O \) relation. Let the list of items in a current itemset be \([p | P] \), where \( p \) is the first element and \( P \) is the remaining list.

2. Add the items into the tree using \( R.add([p | P]) \) procedure that works as follows. If \( R \) has a child \( n \) such that \( n.item == p.item \), remember this node; otherwise create a new node \( n \) and link its parent to \( R \). Then, if \( P \) is non-empty, call \( n.add(P) \); otherwise, set \( n.support \) to itemset support.

During the tree construction process, the itemset supports are assigned to the last nodes of the respecting paths in the tree. If the source list includes all the frequent itemsets that exist in the database, the resulting support tree will not have nodes with no support associated with them, except for the root.

6.4 Dasha implementation

In this section the steps used in the Dasha algorithm are discussed in more details.

6.4.1 Support trees generation

Input: transaction database, minsup, mindiff.
Output: support tree.
6.4. Dasha implementation

6.4.1.1 Frequent itemsets generation and support trees construction

The first step of the Dasha algorithm to compare the patterns in transaction databases is to construct the support trees for both baseline and target datasets.

As the support tree construction is based on frequent itemsets processing, they have to be generated from the database in advance. To complete this task, any existing frequent itemsets mining algorithm can be used. In this study it was decided to use an implementation of the FP-growth algorithm [21]. This algorithm was chosen because of the structure of performance testing datasets that was described in the previous chapter. This structure imposes the existence of large frequent itemsets in the database and the FP-growth should be more efficient for generating them than any of Apriori-based algorithms. Also, the FP-trees that are generated by the FP-growth algorithm are used at the later steps in Dasha for calculating itemset supports.

After a list of frequent itemsets with their supports is retrieved from the database, it is used to build a support tree as described in section 6.3.

6.4.1.2 Decreasing minimum support value for the target tree

Before generating a support tree for the target dataset, it is needed to decrease its minimum support to include in the resulting support tree the itemsets that have a little lower support than the ones in a baseline dataset. This change is important for a later step when the support trees are compared. If not decreasing target’s minimum support, there will be situations when an itemset \( A \) has support slightly higher than \( \text{minsup} \) in the baseline dataset and slightly lower than \( \text{minsup} \) in the target dataset. In this case this itemset should appear in the baseline support tree but it will be missing from the target support tree and during the comparison this will be reported as a false positive violation. However, if the target dataset would contain the itemsets with support lower than \( \text{minsup} \), itemset \( A \) will be presented in it and the comparison of these itemsets will be performed based on the difference between their support values, that can be considered insignificant. In this situation the false violation will not be reported.

It is important to properly select the decreased value for target’s minimum support. If this value is chosen to be too close to the \( \text{minsup} \) threshold specified by the user, it will not solve the problem. If target’s \( \text{minsup} \) is chosen to be too low, it will lead to the drastic drop of the algorithm’s performance, as when \( \text{minsup} \) decreases, the number of frequent itemsets increases and it takes longer to generate them. The following formula was used in this study to calculate \( \text{minsup} \) value for the target dataset.

\[
\text{minsup}_{\text{target}} = \text{minsup} \times (1 - \text{mindiff})
\]

Here, \( \text{minsup} \) is minimum support threshold measured in the number of transactions and \( \text{mindiff} \) is minimum difference threshold measured from 0 to 1. These values are defined by user. The meaning of \( \text{mindiff} \) is explained in section 6.4.2. If the number of transactions in baseline and target datasets differ, the formula has to be adjusted to reflect this difference.
6. DASHA: A PERFORMANCE TESTING MINING ALGORITHM

6.4.1.3 Implementation

The pseudocode implementation of the support trees generation algorithm is presented in Listing 6.1.

Listing 6.1: Support trees construction.

```python
def support_trees(bsl_dataset, tgt_dataset, minsup, mindiff):
    # Decrease minimum support for target dataset
    tgt_minsup = minsup * (1 - mindiff)
    # Construct fp-trees
    bsl_fp_tree, tgt_fp_tree = fp_trees(bsl_dataset, tgt_dataset)
    # Generate frequent itemsets from fp-trees
    bsl_itemsets = freq_itemsets(bsl_fp_tree, minsup)
    tgt_itemsets = freq_itemsets(tgt_fp_tree, tgt_minsup)
    # Construct support trees
    bsl_support_tree = support_tree(bsl_itemsets)
    tgt_support_tree = support_tree(tgt_itemsets)
    return bsl_support_tree, tgt_support_tree

def fp_trees(bsl_dataset, tgt_dataset):
    # Implementation omitted, refer to FP-tree papers.
    bsl_fp_tree = fp_tree(bsl_dataset)
    tgt_fp_tree = fp_tree(tgt_dataset)
    return bsl_fp_tree, tgt_fp_tree

# Construction of frequent itemsets out of FP-tree.
# Implementation omitted, can be found in FP-tree papers.
# Returns list of frequent itemsets.
def freq_itemsets(fp_tree, minsup):
    return itemsets

# Support tree generation
def support_tree(itemsets):
    # The tree is created following Definition 1 (Section 6.3)
    tree = SupportTree()
    for itemset in itemsets:
        tree.add(itemset)
```

6.4.1.4 Example

The resulting support trees generated for the sample datasets with the Dasha algorithm are presented at Figure 6.1. A set of paths in each support tree equals to the set of frequent itemsets for each of the datasets. The trees were generated with \( \text{minsup} = 2 \) and \( \text{mindiff} = 10\% \). Each node in the resulting tree contains a value from the dataset and its support in parenthesis. The support is calculated for the itemset containing the node itself and all its parents up to the root.

In this example the target support is not adjusted as both \( \text{minsup} \) and \( \text{mindiff} \) values are rather small. With such settings decreasing target support is not helpful.

48
From the support trees it can easily be seen that the itemsets \{12, 22\}, \{12, 23\}, \{42\}, \{43\} and some others have different supports and thus represent potential violations. Whether these differences will be reported as violations or not depends on the algorithm’s settings (\textit{minsup}, \textit{mindiff}).

Shaded nodes in the baseline support tree illustrate the roots of the subtrees that differ from target. They are explained in the following section.

### 6.4.2 Support trees comparison

\textit{Input:} Support trees and FP-trees for baseline and target, \textit{mindiff}.  
\textit{Output:} a list of violating itemsets.

#### 6.4.2.1 The comparison algorithm’s description

After the baseline and target support trees are constructed, they have to be compared to find the difference in patterns appearing in the respective datasets.

To find the difference between the support trees, a top-down breadth-first approach is used. The comparison starts with the children of the root node in the baseline support tree. These nodes are added to the comparison queue and until the queue is empty the nodes are retrieved from it one by one and processed.

Once a baseline tree node is retrieved from a comparison queue, the algorithm looks for the corresponding node in the target tree. The support tree is constructed in such a way that every node in it represents an itemset including the node itself and all its parents up to the root. The support value of a node equals to the support of this itemset. Hence, to find a node \( A_T \) in a target tree that corresponds to a node \( A_B \) in a baseline tree, it is needed to move up the baseline tree to reconstruct an ordered itemset \( A \) and find this itemset in the target tree by moving down from the root.

If a lookup procedure is not able to find \( A_T \) node, this is considered a violation. In this case, even if \( A_B \) node has any children, the lookup process for the subtree having \( A_B \) as a root terminates, because all the itemsets identified with \( A_B \)’s children contain \( A_B \)’s item, and if a target tree does not contain \( A_T \), it neither contains any itemsets including its item, which is the same as \( A_B \)’s item.

If \( A_T \) is found, the algorithm compares support values of \( A_B \) and \( A_T \). Before the comparison, if the number of transactions in the baseline and target datasets differs, the support value in the target tree has to be properly scaled to reflect this difference.

To compare the support values for the support tree nodes, minimum difference value is used as a threshold. If \( \frac{|support_{A_B} - support_{A_T}|}{support_{A_B}} \) exceeds \textit{mindiff}, it is assumed that the values are significantly different and this is a violation. In this case, the algorithm reacts the same way as if \( A_T \) is missing. Any \( A_B \)'s children are not processed further.

If the difference between \( A_B \) and \( A_T \) is less that \textit{mindiff}, it is assumed that pattern \( A \) matches in the support trees. In this case, all \( A_B \)'s children are added to the queue.

After the queue is empty, the algorithm reports a list of violating itemsets detected when processing the items in the queue. These itemsets are grouped by their parent non-violating
Figure 6.1: Support trees for the sample datasets.
6.4. Dasha implementation

itemsets (itemset \{a,b\} is called a parent of FP-ordered itemset \{a,b,c\} here and later in this study).

6.4.2.2 Implementation

Pseudocode of the support trees comparison algorithm can be found in Listing 6.2.

Listing 6.2: Support trees comparison.

```python
# Compare the trees and return the violating nodes
def compare_trees(bsl_tree, tgt_tree, mindiff):
    # Initialize comparison queue with root children
    queue = queue()
    for c in bsl_tree.root.children:
        queue.put(c)

    result = list()
    while queue:
        # Get a node from queue
        bsl_node = queue.pop()
        # Find correspondence in tgt (implementation omitted)
        tgt_node = find_node_in_tgt_tree(bsl_node)
        # Correspondence not found, report violation
        if not tgt_node:
            result.add(bsl_node.path)
        # Correspondence found
        else:
            sup_diff = (bsl_node_sup - tgt_node_sup) / bsl_node_sup
            # If nodes differ significantly, report violation
            if sup_diff > mindiff:
                result.add(bsl_node.path)
            # Otherwise add children of baseline node to queue
            else:
                queue.putall(bsl_node.children)

    # After the queue is empty, return violating paths
    return result
```

6.4.2.3 Example

At Figure 6.1 there are the support trees that are built from the sample baseline and target datasets presented in section 6.2. The shaded nodes in the baseline trees illustrate the difference between the trees that is found if the algorithm is run with \( \text{minsup} = 2 \) and \( \text{mindiff} = 10\% \). If the difference is found for a node, the algorithm does not go further to analyze its children.

The results of comparing the support trees for the sample datasets are presented in Listing 6.3.
6. DASHA: A PERFORMANCE TESTING MINING ALGORITHM

Listing 6.3: Comparison of the support trees for sample datasets.

| Shared path: | | (supports 6, 6) |
|-------------|-------------------|
| node 22     | (supports 4, 3)   |
| node 23     | (supports 2, 3)   |
| node 42     | (supports 4, 3)   |

<table>
<thead>
<tr>
<th>Shared path:</th>
<th>[12] (supports 6, 6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node 22</td>
<td>(supports 4, 3)</td>
</tr>
<tr>
<td>node 23</td>
<td>(supports 2, 3)</td>
</tr>
<tr>
<td>node 42</td>
<td>(supports 4, 3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared path:</th>
<th>[32] (supports 6, 6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node 22</td>
<td>(supports 4, 3)</td>
</tr>
<tr>
<td>node 23</td>
<td>(supports 2, 3)</td>
</tr>
<tr>
<td>node 42</td>
<td>(supports 4, 3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared path:</th>
<th>[12, 32] (supports 6, 6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node 22</td>
<td>(supports 4, 3)</td>
</tr>
<tr>
<td>node 23</td>
<td>(supports 2, 3)</td>
</tr>
<tr>
<td>node 42</td>
<td>(supports 4, 3)</td>
</tr>
</tbody>
</table>

6.4.3 Generation of association rules and their analysis

Input: a list of violating itemsets, FP-trees, mindiff.
Output: a list of association rules grouped by antecedent.

6.4.3.1 Association rules generation

After a list of violating itemsets is retrieved, it is used for association rules generation. The itemsets presented in the input list significantly differ from their parents and this allows to conclude that the association rules generated out of the violating itemsets should reflect this difference. For each violating itemset presented in the input list and containing \( n \) items, the Dasha algorithm generates \( n \) association rules having \( n - 1 \) items in the antecedent and one distinct item in the consequent. For each of the generated rules, the algorithm calculates its support and confidence. For rule \([a, b] \rightarrow c\), support is calculated as \(\text{support}_{[a,b,c]}\), confidence is calculated as \(\frac{\text{support}_{[a,b]}}{\text{support}_{[a,b,c]}}\).

An important challenge to be addressed by the algorithm is fast calculation of rule confidences. When the support of a rule can be found in the respective support tree, the support for the rule antecedent is not necessarily presented there. For example, if the algorithm finds a violating itemset \([a, b, c]\), it should generate three association rules out of it, namely

\[
\begin{align*}
[a, b] & \rightarrow c, \\
[a, c] & \rightarrow b, \\
[b, c] & \rightarrow a.
\end{align*}
\]
6.4. Dasha implementation

Obviously, depending on the source data the itemsets \{a, c\} and \{b, c\} may be infrequent and thus missing from the respective support tree. A naïve approach to calculate their supports would be to count their occurrences in the source datasets. However, this would drastically decrease the algorithm’s performance. Instead, the Dasha algorithm retrieves these data from the corresponding FP-trees. To do it, the algorithm takes the least frequent item from the itemset, finds all its appearances in the FP-tree using its node-links and moves up to the root starting from these nodes. If the algorithm passes all the itemset nodes on its way to the root, it concludes that the respective path is valid. Then it summarizes the count values for the least frequent nodes in the valid paths. By the FP-tree design, the resulting value equals to the itemset support in the respective dataset.

It is needed to mention that if the itemset is presented in the respective support tree, its support should be retrieved from there, not from the FP-tree, as the FP-tree lookups described in the previous paragraph are much more computationally intensive than search in a support tree. The FP-tree lookups should only be used if an itemset is not found in the support tree.

A list of association rules generated by the Dasha algorithm for the sample datasets is presented in Listing 6.6. The association rules generated at this step are grouped by their antecedent and are subject to additional optimizations that are discussed in the following section.

6.4.3.2 Association rules analysis and selection

The resulting list of association rules reflects the most important differences between the baseline and target datasets. Level-wise top-down comparison of frequent itemsets implemented with set-enumeration trees has allowed to limit the number of the association rules that are returned. However, the number of these rules can be decreased even more. The algorithm generates many rules with the same consequents and different antecedents. To detect a violation for the node represented by the association rule consequent, the amount of the returned data may be excessive. To address this issue, the algorithm chooses the most representative rule per different consequent instead of returning all the association rules generated at the previous step. The selection criteria is average difference between baseline and target supports for the categories of a given counter in a rule (called rule severity). More formally, it is measured as \( \frac{\sum_{r,b,s, t | r.b = \sup - r.t \sup}}{r.b \sup + r.t \sup} \), where \( r \) is an association rule, \( c \) – rule consequents for the violating performance counter. If severity metric is almost the same for the two rules, the rule with larger baseline support is selected for further reporting. For example, if the algorithm generates three association rules for consequent \( d \), namely

\[
[a] \rightarrow d \text{ (severity 0.15, baseline support 0.4)},
[a, b] \rightarrow d \text{ (severity 0.2, baseline support 0.3)},
[a, b, c] \rightarrow d \text{ (severity 0.2, baseline support 0.2)},
\]

the second rule will be chosen for reporting as it has higher severity than the first rule and higher baseline support than the third rule. In section 6.4.3.4 the result of applying the same heuristic to the association rules generated from the sample datasets is shown.
After the most descriptive association rule is chosen for each of the violating performance counters, it makes sense to also present the distribution of categories for this counter. This can be helpful in making conclusions about the violation – whether the counter values in the target dataset increased or decreased in comparison with the baseline. This can be done using the FP-tree traversals similarly to calculating itemset supports as described in section 6.4.3.1. The traversal will return all the categories of a given rule consequent and their supports. When returning these data, some categories with low supports in baseline and target can be dropped to emphasize the most important categories.

Listing 6.7 shows the output of the Dasha algorithm for the sample datasets after the performed optimizations. This is the final output of the algorithm that is used by the report generation engine described in the following chapter.

### 6.4.3.3 Implementation

The pseudocode implementation of the association rules generation algorithm is presented in Listing 6.4. The implementation of the rules selection algorithm is presented in Listing 6.5.

#### Listing 6.4: Association rules generation.

```python
def build_rules(diff, mindiff, bsl_fp_tree, tgt_fp_tree):
    rules = list()
    for itemset in diff:
        for i = 0; i < itemset.length; i++:
            antecedent = copy(itemset)
            consequent = antecedent.pop(i)
            rule = rule(antecedent, consequent, bsl_fp_tree, tgt_fp_tree, mindiff)
            rules.add(rule)
    return rules

def rule(ant, con, bsl_fp_tree, tgt_fp_tree, mindiff):
    # For better performance, FP-tree lookups should only be called if itemset is not found in support tree.
    # For simplicity, search tree support is omitted.
    bsl_ant_sup = bsl_fp_tree.support_for(ant)
    bsl_con_sup = bsl_fp_tree.support_for(ant + con)
    tgt_ant_sup = tgt_fp_tree.support_for(ant)
    tgt_con_sup = tgt_fp_tree.support_for(ant + con)

    # No need to gen. rules if antecedents supports close
    if supports_close(bsl_ant_sup, tgt_ant_sup, mindiff):
        return None
    rule.antecedent = ant
    rule.consequent = con
    rule.bsl_support = bsl_con_sup
```

54
6.4. Dasha implementation

```python
rule.bsl_confidence = bsl_ant_sup / bsl_con_sup
rule.tgt_support = tgt_con_sup
rule.tgt_confidence = tgt_ant_sup / tgt_con_sup
return rule

def fp_tree.support_for(itemset):
    itemset.fp_sort()
    paths = node_links(itemset.last)
    support = 0
    for path in paths:
        copied_itemset = copy(itemset)
        for item in path:
            if item in copied_itemset:
                copied_itemset.pop(item)
        if copied_itemset is empty:
            support += path.last.count
    return support
```

Listing 6.5: Association rules analysis.

```python
# Additional processing of rules for their better presentation
def select_best_rules(rules, bsl_fp_tree, tgt_fp_tree):
    # Step 1: leave one rule for each different consequent
    best_rules = list()
    # Straightforward implementation omitted
    consequents = rules.get_all_differ ent_consequents()
    for c in consequents:
        # Straightforward implementation omitted
        best_rules.add rules.get_rule_with_max_severity(c)

    # Step 2: Add distribution of categories support
    result = list()
    for rule in best_rules:
        # Straightforward implementation omitted
        distr = get_categories_distr (rule, bsl_fp_tree,
                                      tgt_fp_tree)
        result.add(distr)
    return result
```

6.4.3.4 Example

Association rules generated for the sample datasets are presented in Listing 6.6.

Listing 6.6: Association rules for sample datasets.

```plaintext
Antecedent: [] (supports 6, 6)
consequent 22: supports (4, 3), confidences (0.67, 0.50)
consequent 23: supports (2, 3), confidences (0.33, 0.50)
consequent 42: supports (4, 3), confidences (0.67, 0.50)
```
After performing the optimizations described in section 6.4.3.2, the output of the Dasha algorithm looks as presented in Listing 6.7. The optimizations performed to the association rules as they are shown in Listing 6.6 include the following.

- Rules with duplicating consequents are removed. The resulting output contains only the rules with different consequents.
- For each violating counter the resulting output contains all possible categories that exist in the source datasets (see rule \([12, 32] \rightarrow 41\), which is missing in Listing 6.6 due to its low support and rule \([12, 32] \rightarrow 43\) which is missing from Listing 6.6 as it is considered matching in the datasets).

**Listing 6.7: Final output of the Dasha algorithm for sample datasets.**

```
adash_algorithm ( bsl_dataset , tgt_dataset , minsup , mindiff ):
```

6.4.4 Putting it all together

Source code of the Dasha algorithm can be found in Listing 6.8. It contains the functions described in the previous listings earlier in this chapter.

**Listing 6.8: Dasha algorithm to compare patterns in transaction databases.**

```python
def dasha_algorithm ( bsl_dataset , tgt_dataset , minsup , mindiff ):
    # See Listing 1.
    bsl_fp_tree , tgt_fp_tree = fp_trees ( bsl_dataset , tgt_dataset )
    bsl_support_tree , tgt_support_tree =
```
6.5 Summary

Dasha is a new data mining algorithm for fast detection of differences in transaction databases using association rule mining techniques. This algorithm was designed to compare the results of performance test runs, but can also be applied in other domains. For example, it is expected that the presented algorithm will be effective in comparative market basket analysis for detecting changes in customer behavior between different locations or over time. However, this assumption has to be justified with a case study that was not performed in the scope of this project.

```plaintext
6.5. Summary

support_trees(bsl_fp_tree, tgt_fp_tree, minsup)

# See Listing 2.
diff = compare_trees(bsl_support_tree, tgt_support_tree, mindiff)

# See Listing 4.
rules = build_rules(diff, mindiff,
                    bsl_fp_tree, tgt_fp_tree)

# See Listing 5.
return select_best_rules(rules, bsl_fp_tree, tgt_fp_tree)
```
Chapter 7

Case study: Dell DVD Store

Dell DVD Store\(^1\) (DS2) is an open-source web benchmark that is often used by academic researchers and software developers in industry to performance test web software and server hardware. This is a rather simple web application that is intended to simulate a web store allowing its customers to browse and purchase DVDs. DS2 consists of a backend database layer, a web application layer and driver programs. The database layer contains information about the registered users, available DVDs and fulfilled orders. During the setup process, it can be populated with automatically generated data of arbitrary size. Web application layer allows to browse the store and make purchases using a web browser. Driver program simulates the load to an application following the patterns common for web store customers.

7.1 Testing setup

7.1.1 Case study goals and hypotheses

**Goal 1** To apply the automatic performance detection approach presented in Chapter 6 to a relatively simple example.

**Goal 2** To study the effects of changing the algorithm’s settings on the returned results.

**Goal 3** To illustrate the workflow of a performance engineer during the analysis of the performance tests using the automatic performance regressions detection tool.

**Hypothesis 1** No violations should be detected between two performance test runs that are executed against the same version of the application under study in same conditions.

**Hypothesis 2** If a new version of the software under study contains a performance degradation, violations of the tracked performance counters will be raised after comparing the performance tests results with those captured while testing a baseline version of the software.

\(^1\)http://linux.dell.com/dvdstore/
7. CASE STUDY: DELL DVD STORE

7.1.2 Hardware and software

The hardware setup for this case study consisted of three computers. Dell DS2 was installed on a dedicated Linux server with an automatically generated database of 20 GB size. The database size was chosen to be several times larger than the amount of available RAM. This led to quite large read and write times both at database and disk system levels and resulted in clear performance testing results. A load generator was run from a laptop located in the same network as the target machine. The communication between two machines was set up using Wi-Fi connection. The values of the performance counters gathered at the target machine were stored in OpenTSDB time-series database\(^2\) instance that was deployed in a virtual machine in the cloud.

Dell DVD Store includes several different database backends and web layer implementations. For this case study, the application was configured to work with MySQL database and PHP web layer deployed at Apache web server.

To have more control over the load generator and access the latencies of the issued requests, instead of using a default load driver supplied with DS2 benchmark, a performance test plan was created with Apache JMeter\(^3\) tool. The flow of this test plan resembled one of the default load driver and consisted of various user activities including login, registration, browsing and purchasing. All the performance test runs discussed in the chapter were run with 10 threads. Half of the users issuing requests to DS2 were already registered and had to log in, the other half had to register first. Each user performed five different searches on average and each second user issued a request to buy some DVDs in the store after the searches.

7.1.3 Tracked performance counters

To analyze performance of a target application, 17 performance counters were tracked and analyzed in the research prototype consisting of a performance testing repository\(^4\) and an analysis module based on the Dasha algorithm\(^5\). These counters are overviewed in Table 7.1. Request latencies were gathered at a client machine with a load generator. Contrary to the values of the other counters, these values were gathered upon receiving a specific request. The other counters were gathered at a target machine with sampling periods of 15 or 60 seconds.

7.1.4 Research prototype configuration

All the tests in this case study were performed using quartile-stddev classifier with outlier boundaries set to baseline median \pm 1 \text{ standard deviation}\(^6\), if not specified otherwise.

---

\(^2\)\url{http://opentsdb.net}
\(^3\)\url{http://jmeter.apache.org}
\(^4\)See Chapter 4 for performance testing repository description.
\(^5\)See Chapter 6 for the Dasha algorithm’s description.
\(^6\)See Section 4.5.2 for more details on the used classification algorithms.
7.1. Testing setup

<table>
<thead>
<tr>
<th>Category</th>
<th>Counter</th>
<th>Sampling period, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>User load</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>System load</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>I/O wait</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Num interrupts</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Num context switches</td>
<td>15</td>
</tr>
<tr>
<td>I/O system</td>
<td>Read requests</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>Write requests</td>
<td>60</td>
</tr>
<tr>
<td>MySQL</td>
<td>Bytes received</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Bytes sent</td>
<td>15</td>
</tr>
<tr>
<td>Request latencies</td>
<td>Browse actor</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Browse category</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Browse title</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Login</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Register</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Purchase</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Checkout</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 7.1: Performance counters tracked during DS2 analysis.

The analysis was performed with minsup changing between 10% and 20% and mindiff changing between 30% and 50%.

7.1.5 Additional analysis metrics

The Dasha algorithm presented in Chapter 6 analyzes the differences between the two datasets and reports the most important violations that are found during the analysis. This algorithm does not assess the severity and nature of the reported violations. However, it is obvious that these violations may be of different importance. Let us consider two situations that would illustrate this difference between the violations that can be reported by the Dasha algorithm.

As a first example, let us assume that CPU system load counter was tracked in two performance test runs and during their analysis the algorithm reports a violation for this counter that has 20% support in the baseline dataset and 18% support in the target dataset. In the baseline, 60% of the counter values are in 2nd category and 40% are in 3rd category. In the target, 50% of values are in 2nd category and 50% are in 3rd category.

As a second example, let us assume that during the same runs the algorithm also tracked MySQL received bytes counter and after the analysis it reports a violation for this counter that has 100% support in the baseline dataset and 90% in the target dataset. In the baseline, 80% of values are in 2nd category, 10% are in 1st category and 10% are in 3rd category. In the target, 10% of values are in 3rd category and 90% are in High category.

\(^7\)Minsup and mindiff are explained in section 5.2.1 and section 6.4.2 respectively.
It is obvious that the second violation is more severe and more attention has to be paid to it. While the observed minor differences in CPU system load counter values could have been caused by data fluctuations appearing because of the stochastic processes in the hardware and software environments, the differences in MySQL received bytes counter behavior most probably represent a significant performance regression. This conclusion can be made because the violations for the second counter are observed throughout the whole duration of the second test run and distribution of values between the baseline and target runs differs dramatically: while the counter values in the baseline are grouped near the median, the target values are mostly outliers that significantly exceed the baseline values.

As it can be seen from the provided example, the violations reported by the algorithm may be of different importance and to be more effective, the application implementing it should additionally inform the performance analyst about those violations that require more attention and have to be investigated first of all.

To reflect the differences in importance of the reported violations, the presentation algorithm uses three metrics, namely support, severity and hybrid. Support equals to the support of a violated rule antecedent in the baseline dataset and represents the fraction of a tracked time period during which the violation is observed. For the examples discussed earlier, the value of this metric equals 20% and 100% respectively.

Severity metric is designed to be a measure of difference in categories distribution. This metric is used in the Dasha algorithm to find the most important association rule per given consequent\(^8\). It is measured as \(\frac{\sum_{i=1}^{c} bsl_{sup} - c_{i} tgt_{sup}}{bsl_{sup} + tgt_{sup}}\), where \(r\) is a violating association rule, \(c\) – rule consequents. For the examples discussed earlier, the value of this metric equals to approximately 10% and 90%.

Hybrid is a composite metric that equals to severity + support/5. The violations reported by the Dasha algorithm get sorted according to this metric before being shown to the analyst. Hybrid metric was designed to reflect the changes both in support and severity and address the fact that for each of the rules support value always changes between 100% and 0% while the severity metric usually has smaller range and thus has to have higher weight. The difference in metrics weights was determined empirically.

7.2 Experiments

7.2.1 Experiments description

During the described case study, four performance test runs were completed and compared using different analysis parameters. Each test run was executed for 60 minutes. Initially, two performance tests were run one after another to form a baseline for the analysis of the following runs. These tests were run with no alterations of DS2 source code. The second of these two test runs was analyzed against the first one. The other test runs were analyzed against a baseline formed out of the two first test runs.

Before launching the third test run, the code responsible for the registration procedure was altered by adding a delay to it. Because of this delay, the latency of the respective

\(^8\)See Section 6.4.3.2 for more details.
7.2. Experiments

request increased by approximately 25%. After comparing this test run with the baseline it was expected that Register latency would be reported as a violation. Also, the introduced idle period should have had an impact on the values of some other counters gathered at a target machine because of slightly decreased load. The fourth test run had a larger delay added to the same place. For this test run, the average latency of registration request was increased by approximately 35%. The reasons behind introducing this test run are described in the following section.

7.2.2 Analysis of results

7.2.2.1 Comparison of baseline runs

After the first two test runs with no DS2 code alterations were executed, they were added to the system and compared against each other to make sure the algorithm does not discover violations in them and to find proper threshold values for minsup and mindiff parameters. The first run was considered a baseline run, the second one was considered a target run. When executing the analysis with minsup = 20% and mindiff = 50%, no performance violations were detected between the runs. After the analysis was run with minsup = 15% and mindiff = 50%, two violations were detected. However, these violations were raised with very low values of severity (0.1 and 0.2 respectively). The reported counters were then inspected manually using the time-charts generated by the research prototype. The investigation did not reveal significant differences in the values. For this reason, it can be concluded that the violations were reported because of minor data fluctuations usual for the performance testing datasets and were not related to the performance regressions that could have been introduced in the second run.

When the analysis was run with minsup = 10% and mindiff = 50%, eight violations in total were reported. The values of hybrid metric in them were generally higher than in the previous experiment, but either support or severity were low. Manual investigation of the newly reported violations did not reveal significant differences between the datasets.

When the analysis was run with minsup = 20% and mindiff = 40%, four minor violations were reported, and when mindiff was set to 30%, eight violations were reported. Decreasing mindiff to 20% lead to reporting the same eight violations with slightly changed support and severity values. Experiments with simultaneously decreasing minsup and mindiff did not lead to additional interesting observations.

The analysis results discussed in this section clearly indicate that the performance testing datasets may contain significant amount of noise that has to be taken into account during the analysis. To remove this noise, either minsup and mindiff parameters have to be set to rather high values — between 10% and 20% for minsup and between 30% and 50% for mindiff — or the violations with low support or severity metric values have to be removed from the analysis results before showing them to the analyst. If minsup and mindiff are set to lower values, the analysis algorithm will detect and report even minor differences between the datasets. This behavior can be useful for certain applications, but is harmful for performance analysis because of the noise contained in the analyzed data.
It can thus be said that Hypothesis 1 is valid proven that the algorithm’s settings are high enough to mitigate the noise from data.

7.2.2.2 Performance degradation of registration request

To assess the ability of the research prototype to detect performance regressions in performance testing datasets, a delay was added to the registration request as described in section 7.2. The results of a performance test run with the introduced regression were analyzed against two prior test runs considered as a single baseline.

The first analysis was conducted with \( \text{minsup} = 20\% \) and \( \text{mindiff} = 50\% \). These values were empirically determined when comparing the baseline runs as sufficient thresholds for mitigating noise in data. During the analysis with these parameters, eight performance violations were detected, with six of them having severity > 0.2. Out of all counters belonging to Latency category, only Register latency counter was flagged as violation; the other reported counters were CPU- and database-related.

Register latency counter was flagged with severity = 0.43, support = 0.62. Two MySQL-related counters were considered more important than this counter due to higher values of hybrid metric, the other counters were reported as less important.

7.2.2.3 Effects of changing the classifier configuration

While the reported counters themselves were in line with the expectations, it was not clear why MySQL-related counters were flagged with higher priority than the target counter Register latency. To investigate it further, time-series charts for the target counter and MySQL sent bytes with severity = 0.53, support = 0.62 were examined. These charts are provided in Figure 7.1.

From the boxplots it is clear that the difference between values of Register latency counter is greater than the difference between MySQL sent bytes values. This difference can also be seen in the time-series charts. However, if inspecting the categories into which the values fall (the category boundaries are shown at time-series charts with the dashed lines), it can be seen that most of the values for Register latency counter fall into the same 2nd category as baseline values while the values for MySQL sent bytes mostly get into lower categories. The reason for that is the way the values are distributed by categories in the classifier. Sometimes the analysis may fail to detect the violations (especially minor ones) because the values of a violating counter will still fall into the same category as the baseline values. However, in this situation most probably the violation won’t remain undetected because it will be raised for the related counter as it can be seen here.

The distribution of categories for the counter values in this experiment is shown in Figure 7.2. From the provided charts it can be seen that for MySQL sent bytes counter it resembles normal distribution with mean in 2nd category for the baseline and in 1st category for the target. For all the categories, the average difference in confidence between baseline and target is high, which leads to high severity value. For Register latency counter, the average difference in confidence is smaller and this leads to smaller severity.
Figure 7.1: Time-series charts and boxplots for experiment 7.2.2.2.

(a) MySQL sent bytes

(b) Register latency
7. Case study: Dell DVD Store

Figure 7.2: Categories distribution for experiment 7.2.2.2.

(a) MySQL sent bytes

(b) Register latency
To mitigate the discussed problem, the configuration of a classifier can be adjusted. In the next experiment, the outlier boundaries were moved from ±1 standard deviation to ±0.8. After this change, the degradation of Register latency was marked as the most important with severity = 0.55, support = 1.0. However, this change may lead to less precision for more volatile counters. It can be said that the parameters of the chosen classifier have to be found empirically based on the set of available performance counters. It could also be useful to allow the classifier to have different settings for the different counters. However, this is not investigated further in this study and left for future work.

### 7.2.2.4 Effects of changing the analysis configuration

To understand the relation between the analysis settings and the results returned by the performance degradations detection algorithm, the analysis of a run with degraded performance was repeated for several times with different parameters. Before these experiments, settings of the classifier were set to default values.

When the analysis was run with minsup = 0.15, mindiff = 0.5, all the important violation counters remained in place, three additional minor violations were reported. With minsup = 0.1, mindiff = 0.5, four minor violation were reported. After increasing minsup gradually to 0.3 and 0.35 the algorithm was still able to detect the five most important violations. With minsup = 0.4, one of the important violations (MySQL received bytes) was not reported.

Several experiments were run with minsup set to 0.2 to find the impact of changing mindiff. Gradually decreasing it from 0.5 to 0.2, it was still possible to detect all the most important violations. However, the lower was mindiff, the more minor violations were reported. Then, mindiff value was gradually increased to 0.9. At this setting, Register latency counter was not flagged as violating. However, at all the earlier steps with mindiff = \{0.6, 0.7, 0.8\} the five most important violating counters were reported correctly. The higher was mindiff, the higher was severity and the lower was support for Register latency counter.

From the performed experiments it became clear that analysis settings minsup = 0.2, mindiff = 0.5 are the most suitable for detecting performance regressions in the described environment because with this settings the algorithm can efficiently filter out noise from the data and report the important violations.

### 7.2.2.5 Another registration regression experiment

As it was mentioned in section 7.2, performance testing data was also gathered for the test run with performance of registration request decreased by approximately 35%. This test run was executed to see how the violations are reported when the values of the violating counter do not fall into the same category as the baseline values.

This test was run with minsup = 0.2 and mindiff = 0.5. For this configuration, 11 violations were reported, all of them having medium or high severity. Register latency counter, however, was reported as the most important, with severity = 0.97, support = 1.0. The next important counter was reported with severity = 0.67, support = 0.62. All five important violations that were reported when analyzing the previous test run with smaller
delay introduced into the request processing code, were reported in this analysis as the most important. This behavior is perfectly correlated with the expectations.

Thus, during the experiment **Hypothesis 2** was validated. After a regression was introduced into the new release of the target software, the research prototype flagged several performance counters as violations. It has to be noticed, however, that because of precision loss during the consolidation and classification as well as because of the noise contained in performance testing data, minor violations can be not detected by the implemented research prototype.

### 7.3 Report generation

In this section the presentation of performance tests analysis results is discussed. The presentation of data mining results used in the presented approach is similar to one suggested by Foo et al. in [19]. As an example here a sample output from comparing two performance test runs on Dell DVD Store benchmark is used. This section does not aim to thoroughly explain the used approach, as all the theoretical details behind the algorithm for detecting performance degradations are discussed in the previous chapters.

The analysis starts from test details page. It is shown in Figure 7.3. This page contains a list of test runs that are associated with the current test. For each test run in the list, its status (baseline or target) and execution time are shown. Also, it is possible to open full test run details by pressing a link with the test run name. This link leads to a performance repository page associated with the test run.

![Test details page](image)

Figure 7.3: Test details page.

The analysis is started by clicking “analyze” link. The selected target test run will then be compared against all the runs defined as a baseline. The results of the analysis are shown in Figure 7.4. This page contains a list of violating counters sorted by the value of *hybrid* metric. The algorithm tries to mark the most important violations with red and yellow colors.
7.4 Summary

The experiments performed by the author in this case study and while working on the research prototype implementation indicate that the Dasha algorithm presented in Chapter 6 can effectively compare the transaction databases and report the most notable differences using association rule mining techniques. The application of the algorithm for detecting performance regressions has shown that in order to be the most efficient, the algorithm’s parameters should be adjusted to remove noise from data. Also, it is important to correctly set up a classifier distributing the continuous values of the tracked performance counters into discrete categories.

The experiments performed during this case study indicate that the algorithm is the most efficient when minsup is set to 0.2, mindiff is set to 0.5. Decreasing minsup leads to polluting the results with the noise contained in the data; increasing it may lead to missing important violations. Changes in mindiff have less impact on the returned results but if this setting is set to extremely low or high values, this may lead to incorrect results as well.

All the goals set before this case study were reached. The implemented research prototype was able to locate performance degradations introduced into Dell DS2 code. When comparing the baseline test runs against each other, no significant violations were reported.

Report generation process was described.

Both the hypotheses were validated. It is important, however, to correctly configure the research prototype before the analysis to mitigate noise from data and do not lose important information contained in it.
Figure 7.4: Analysis results page.
Chapter 8

Case study: TomTom PND Web Engineering

In the previous chapter it was shown on a simple example how a research prototype implementing the Dasha algorithm can be used to automatically detect performance degradations between different test runs. The best combination of the algorithm’s settings to remove noise from data and efficiently report performance violations was determined empirically.

In this chapter applicability of the research prototype in a real industrial setting is discussed. During his graduation project, the author worked at TomTom International BV on performance testing of a large e-commerce web application. As a part of his project, a set of performance tests covering main application use cases was developed. These tests were added to a continuous integration server for automated execution to test new application releases. The results of performance tests were analyzed using the research prototype presented earlier in this document. Description of these tests and findings made during their analysis are presented in this chapter. Because of a non-disclosure agreement, certain details about the system under study and its operation are omitted from this document.

8.1 Web application under study

8.1.1 Application selection

As it was covered in section 2.2, the main responsibility of TomTom PND Web Engineering department is developing and maintaining a set of web applications intended to support the company’s clients who possess TomTom portable navigation devices and smartphone programs. These web applications are logically organized into a single ecosystem and communicate with each other by means of web requests. When deployed to production servers, the applications experience high load and it is vitally important for the company to ensure their correct operation in these conditions.

Because of a large number of system’s responsibilities, its general complexity (both in terms of code and infrastructure, meaning support of several different testing environments, authentication and other security issues, etc.) and limited time frame dedicated for the case study, it was impossible to performance test the system as a whole. Instead, the tests were
8. CASE STUDY: TOMTOM PND WEB ENGINEERING

designed and performed against the Philadelphia web application, which is an important part of a system. This backend component has a number of important responsibilities and communicates both with the internal database and most of other system components. The Philadelphia database contains information about the digital products available for purchase in TomTom online store and customer-facing frontend applications fetch it via calls to the Philadelphia. Also, it implements a number of web services related to the management of digital content installed at the clients’ devices.

Recently, this component was optimized for performance. After these optimizations, response times for most web requests served by it have decreased and became more stable and predictable than they were before the optimizations. However, certain requests processed by Philadelphia still remained unstable because of being dependent on the other components having less optimized code. It was requested by Philadelphia developers to prepare a set of performance tests and a scenario to execute them easily for assessing performance of the latest changes to the codebase as well as to ensure that no significant performance degradations are introduced between the application releases.

8.1.2 Request groups

Web requests served by Philadelphia can be logically divided into two groups. The requests from a first group interact only with the internal MySQL\(^1\) database that contains static data and is rarely updated. They read data from the database and transform it to a format requested by the user. However, these processing steps can be rather complicated and require a lot of computational power. This scenario allows to efficiently cache data at different levels of its processing and ensure stable response times under the required load. Despite the database is deployed separately from an application server, database queries do not introduce significant fluctuation of the request latencies, because MySQL is very well optimized and many calls to it are cached. Network latency also does not change much because all the servers are located nearby and are connected with high-bandwidth connections with excessive throughput.

The second group of requests depends on the other components of the system. Each request in this group triggers a number of additional requests to obtain necessary data from different system parts and cannot proceed until the responses to these requests are returned and processed. This makes performance testing a non-trivial task, because the resulting latencies obtained after generating the load and applying it to the system do not only contain information about the performance of a target component, but also include the latencies of the internal system components. Thus latency of these requests becomes a composite metric that is difficult to use for estimating performance of a target system component on its own.

For performance testing purposes it is needed to know both end-to-end latency that is a fraction of time between issuing a request to the target component (in this case, Philadelphia) and receiving a response and latency of a target component that is end-to-end latency minus the latencies of the external components. End-to-end latency is important for the end users as it represents a delay the user observes when issuing a request and awaiting for a

\(^{1}\)http://mysql.com
response. Target component latency is important for the developers of this component, as it shows whether the main source of delays is in the target code or not.

It is clear that end-to-end latency and target latency are the same for requests from the first group, as they do not depend upon external components. Their performance testing can be done using Apache JMeter or a similar workload generator. For the requests from the second group the situation is different. It is impossible to get target latencies for the involved components of a system under study during performance testing with a workload generator only. There are two ways to get this information.

One of the possible approaches is to use a low-overhead profiler available in a number of contemporary application performance management suites, like AppDynamics\textsuperscript{2}. Such profilers can be used even at highly loaded servers and are able to display accurate timings of the executed requests at target components. A drawback of this approach is that these tools are rather expensive, hence they cannot be applied in all situations. Also, the existing products in this market are not yet flexible enough to provide raw data from their measurements, its extraction requires significant effort, if possible at all.

Another way to measure the latencies at the target component of a system under study is to mock the external components with stubs. During the described case study, this approach was used. If using stubs, every request to an external system gets intercepted by mocking code that immediately returns fake data generated according to the specified templates. One drawback of this approach is a need to program stubs in advance and make sure that the fake objects resemble real data. Another drawback is that mocking can only give approximate results. This happens because of difference in data and processor time distribution for the case when some of the external components are deployed together with the target component. However, as at TomTom environments all the system components are deployed separately, this issue was not taken into account and performance testing of these requests was done using already available mocks for the external parts of a system.

8.2 Testing setup

8.2.1 Goals and hypotheses

Goal 1 To study the behavior of the implemented research prototype on testing changes in performance of a large enterprise web application between two its releases.

Goal 2 To study the effects of changing the algorithm’s settings on the returned results.

Hypothesis 1 The results of automated analysis using the research prototype will correlate with the results of manual analysis using time-series charts and aggregated values of the tracked performance counters.

\textsuperscript{2}http://appdynamics.com
8.2.2 Hardware and software

Similarly to the Dell DS2 case study described in the previous chapter, hardware setup for this case study consisted of three computers with the first of them hosting the target application, the second one being used as a load generator and the third one running OpenTSDB server for gathering hardware-related performance counters. The same performance counters as described in Table 7.1 were captured in this study. Naturally, instead of DS2 request latencies, latencies of requests to the Philadelphia system were measured. The tests were executed with 50 threads instead of 10 threads used during Dell DS2 testing. In all the test runs \( \text{minsup} \) was set to 0.2, \( \text{mindiff} \) was set to 0.5, if not indicated otherwise.

All the performance tests written to analyze behavior of Philadelphia under different conditions were organized in a separate project and put under version control in the company’s repository. The performance tests themselves were created using Apache JMeter\(^3\).

To automate tests execution, a scenario for their launch was prepared using Apache Maven\(^4\). This tool is used to organize build processes for the evolving software projects. Its rich core functionality and a wide collection of available plugins allows to manage execution even of very complex build scenarios needed for large industrial projects.

For letting Maven run JMeter tests and analyze their results, two plugins were set up, namely \texttt{jmeter-maven-plugin}\(^5\) and \texttt{jmeter-analysis-maven-plugin}\(^6\). The first plugin allows to specify a directory where the plugins are kept and define properties that can later be read from JMeter scripts. Also, it runs the JMeter executable in the specified build phase. The second plugin helps to generate reports out of raw XMLs with JMeter performance testing results. To allow performance testing against different testing environments, separate profiles were created for each of them.

To speed up information retrievals, caching is used pretty heavily in the Philadelphia. Hence, performance testing results may significantly differ between two test runs executed against the same software version if the state of caches is different prior to the tests’ execution. To mitigate this problem, cache clearing requests were added at the initial step of each testing script.

One of requirements for the performance testing project was to set up automatic execution of performance tests on schedule. To satisfy this requirement, a respective job was added for pre-production testing environment to Jenkins continuous integration server\(^7\) used in the company. As the performance testing project created for this case study was a simple Maven wrapper over JMeter test files, it was rather easy to integrate it with default Jenkins installation. To support analysis of JMeter results and displaying performance trend charts from Jenkins, \textit{Performance} plugin\(^8\) was added to it. This plugin cannot show details of the performance test runs or even distinguish different calls in JMeter tests, thus its applicability is rather limited. However, it is suitable for the initial estimation of application performance.

\(^3\)\url{http://jmeter.apache.org}  
\(^4\)\url{http://maven.apache.org}  
\(^5\)\url{https://github.com/Ronnie76er/jmeter-maven-plugin}  
\(^6\)\url{https://github.com/afranken/jmeter-analysis-maven-plugin}  
\(^7\)\url{http://jenkins-ci.org}  
\(^8\)\url{https://wiki.jenkins-ci.org/display/JENKINS/Performance+Plugin}
8.3 Experiments

8.3.1 Experiments description

Performance tests created to analyze Philadelphia performance were built to cover the requests from the both groups discussed above. However, no significant differences in test results behavior were found between the two groups, thus only the tests covering service $S$ that belongs to the first group of requests are discussed here. This service does not communicate with any of the external components. When it receives a request from a client, its processing steps to generate a response consist only from communicating with MySQL database and doing the necessary internal calculations and data transformations.

When Philadelphia is deployed at production servers or at one of the available testing environments, it is set up to cache the calls to this service for better performance. In the tests described in this chapter, service $S$ was tested with the caches being disabled and enabled in different testing configurations. These tests are analyzed separately.

Two Philadelphia releases were used in this case study. Release $R_{\text{new}}$ was the most recent one by the time the performance tests were run. It was considered a target release. Version $R_{\text{old}}$ was released approximately two months before $R_{\text{new}}$ and was treated as a baseline release.

All the tests performed in this case study were set up to run for approximately an hour. This duration of a run was chosen to gather a sufficient number of observations for the performance counters. At least five test runs were executed per each test configuration to ensure their stable behavior and discard the runs with significant performance fluctuations.

8.3.2 Analysis of results

8.3.2.1 Discarding broken runs and mitigating effects of data fluctuations

Before comparing the runs executed on $R_{\text{old}}$ version against $R_{\text{new}}$ version, the runs testing the same software version were compared against each other to make sure no significant violations are found between them.

Analysis of baseline runs has shown that because of minor data fluctuations some counters are classified differently between the similar test runs and thus have different categories distribution even if comparing two executions that were run one right after another. Due to the stochastic structure of processes in non-real-time operating systems, background tasks and slightly different behavior of a load generator the aggregate performance counter metrics may differ between two runs by 0.5–1%. This difference is usually difficult to see if just comparing the time-series for the respective runs. However, this may become a problem for a classifier, as values for many performance counters do not fluctuate significantly and are usually distributed very close to the median value. If in a target test run a median differs from the baseline by as little as 0.5%, this difference becomes sufficient for the algorithm to classify the values of the respective counter differently.

To mitigate this problem, it is needed to form a baseline out of several different test runs. In this case, the classifier would operate with more data and the categories would be a little wider.
Also, it turned out that some test runs may contain counters that behave very differently from the same counters captured during the other runs. This issue may be caused by the background processes at a target machine or slightly changing performance of a load generator or network. To prevent polluting the results with these obviously wrong data, broken datasets were discarded before the analysis.

### 8.3.2.2 Effect of slow calls to latency counters classification

Testing Philadelphia application with enabled caches has uncovered an issue that was not observed during the earlier experiments. It turned out that the latencies are sometimes not consolidated correctly because of the slow calls. Most of the calls to the systems that use caching are completed extremely fast. At the same time, there almost always are cache misses, and the requests not fetching data from caches can take orders of magnitude longer to complete, when compared to the cached requests. It is especially easy to notice during the **cold start**, when the system is started with all the caches being empty and many requests fetch data using heavy calculations because of a high percentage of cache misses. However, even when the caches are mostly filled, there still can be requests that do not hit them. Even if the percentage of cache misses is low, the respective requests may still significantly affect the aggregated latencies, as their values are much higher than those for optimized requests.

As an example of such behavior, let us consider calls to service $S$ during one of the test runs. If its results are cached, it takes around 250 ms to complete a request to it. However, if cache misses happen during its execution, the requests may take between 2000 and 11000 ms. Because of these slow calls, median latency at this run was reported as 380 ms, average latency was around 1500 ms. Raw and consolidated values for this performance counter are shown in Figure 8.1.

From Figure 8.1(a) it is clear that despite the fact that most of the requests are rather fast (median is at 380 at box-and-whisker plot), for many requests it takes several seconds to complete.

In Figure 8.1(b) it is shown how these slow requests affect distribution of values into categories. It was expected that the values after the cold start would get into 2nd category, but most of them were classified as belonging to 3rd category because both the classifier and consolidated values were affected by the slow requests. To overcome this issue it can be recommended to use different classifiers for different performance counters and design a classifier taking this behavior into account for the latency counters. This is more flexible than having one classifier for all the counters, no matter what they are. However, this improvement was not implemented in the scope of this project.

### 8.3.2.3 Analyzing service $S$ with disabled caching

This experiment can be considered artificial, as caching is always enabled at the production servers. It was however interesting to see how its results correlate with the results obtained when testing the system with enabled caches.

The results of performance testing have indicated that most analyzed counters behave similarly in baseline and target runs. Thus, the median values for all but two performance
8.3. Experiments

counters in the target runs were fluctuating within 1% from the respective baseline values. The differences were spotted in behavior of CPU context switches and CPU System counters. The former counter indicates how often the CPU switches between different processes or threads within a process. In all target runs the median values for this counter were approximately 6–7% higher than in the baseline runs. The association rules reporting this violation were fired with severity = 0.56 – 0.62 and support = 67 – 81% during the analysis, which allows to consider this violation rather important. Median values for CPU System counter in the target runs were approximately 2 – 4% lower than in the baseline. This violation was detected with severity = 0.43 – 0.51 and support = 43 – 81%. Most probably, these two violations are related, as if a number of context switches increases, CPU spends more time on loading and unloading data to/from registers that is reflected in CPU System time. It is expected that CPU System performance was also lower in the other target runs, but the algorithm was not able to distinguish CPU System performance degradation from the noise.

Philadelphia developers have assumed that a cause of this degradation is an increased number of objects that are shared between different threads. Between \( R_{old} \) and \( R_{new} \) releases there were changes to the code base that could have led to such increase.
8.3.2.4 Analyzing service \( S \) with enabled caching

Manual analysis using the aggregated values reported by the performance repository has shown that the most significant difference in performance between \( R_{\text{old}} \) and \( R_{\text{new}} \) is in behavior of CPU system counter, as its median value has decreased by approximately 3–4\% in target releases. Average latency of one of two tracked requests \( R_1 \) has improved by approximately 2\% in comparison with the baseline runs. The other counters were fluctuating within 1\% of baseline values.

Analysis of the target runs with the research prototype has shown that it is able to correctly detect these changes in performance. Thus, in all six valid target runs CPU System counter was marked as a violation with average severity \( = 0.28 \) and support \( = 100\% \) in 5 out of 6 cases. If considering violations raised within all six runs, this counter was the most important violation.

Latency counter \( R_1 \) was also reported in all of the target runs, though it had lower severity and support values. It was reported with average severity \( = 0.18 \) and support \( = 64\% \) in 5 out of 6 cases.

In a half of the target runs, CPU user and CPU context switches were also reported as important violations. Severity for CPU user was fluctuating between 0.51–0.61, support was fluctuating between 29–55\%. For CPU context switches, severity was 0.54–0.67, support was 26–41\%. Further investigation has shown that in the respective runs these counters were deviating from the baseline more than in the others, thus they were not reported in the rest of the target runs.

A number of minor violations was reported in each target run. An average number of reported violations was 6.7.

8.3.2.5 Effects of changing analysis configuration

When studying the applicability of a developed research prototype for performance analysis tasks on Dell DS2 example, the optimal combination of the association rule mining algorithm’s settings was determined empirically\(^9\). In this configuration, \( \text{minsup} \) was set to 0.2, \( \text{mindiff} \) was set to 0.5. The former tests in this case study were completed with this configuration. In this section the performance testing results after altering the association rule mining algorithm’s settings are studied. This analysis was performed to justify the results obtained during Dell DS2 case study.

All the analyses discussed in this section were performed against the results of performance testing \( S \) service with enabled caching. The results presented in section 8.3.2.4 are considered reference for this section.

After increasing \( \text{minsup} \) to 0.3, four out of six target test runs were reported as not having any violations. In two remained test runs, the number of detected violations has decreased, their severity and support have been lowered.

When \( \text{minsup} \) was increased to 0.4, the number of violations in two remained test runs has decreased even more. In neither of test runs the most violating counter CPU System was reported.

\(^9\)See Section 7.2.2.4.
After \textit{minsup} was changed to 0.1, more minor violations were reported in a number of target runs. Violations of \textit{CPU System} counter were considered in general less important.

It can be said that the optimal value of \textit{minsup} equal to 0.2 that was determined empirically in section 7.2.2 turned out to be the most suitable for this set of tests as well. If \textit{minsup} is set to a larger value, some important violations may be not shown in the results. If this setting is set to be lower than optimal, the data may become polluted with the noise. Same trends were detected after analyzing the results with \textit{minsup} equal to 0.25 and 0.15. However, if changing \textit{minsup} with a lower step, the difference between the results is less apparent.

After setting \textit{minsup} back to 0.2 and increasing \textit{mindiff} to 0.6, the results generally became more clear as in three out of target six test runs the number of reported violations has decreased. In two of them, only \textit{CPU System} and \textit{R1} were reported.

When \textit{mindiff} was increased further to 0.7, \textit{CPU System} counter was still reported in all six target runs. Moreover, in five out of six runs, either only this counter or its combination with \textit{R1} were reported. It can be said, however, that this value is too high to be recommended to use, as the other CPU-related violations that were missed from a report with this algorithm’s configuration, were also important.

Decreasing \textit{mindiff} to 0.4 led to the appearance of several additional minor violations in a number of target test runs and a slight change in severity and support metrics for a few counters. Decreasing it further to 0.3 led to even more new violations. The existing violations were reported with slightly lower values of severity and support. These results also correlate with the results obtained when studying the effect of changing the algorithm’s settings in section 7.2.2. It can be said that changing \textit{mindiff} has lower effect on the reported violations than changing \textit{minsup}, and \textit{mindiff} values in a range between 0.4–0.6 are optimal for the performance analysis tasks.

### 8.4 Summary

A research prototype implemented in the scope of this project was able to detect performance regressions in a large industrial web application. Adjusting the algorithm’s settings allows to change the threshold for the importance of reported violations.

In the previous chapter, the optimal combination of the algorithm’s settings was determined. It was claimed that using this configuration the algorithm is able to distinguish performance degradations from the noise contained in the performance testing datasets and report all the important violations. Analysis of changing the algorithm’s settings on Tom-Tom performance testing datasets has justified this claim.

While analyzing certain target runs there may appear violations that are not typical for the rest of the runs. This happens because of noise in data and can be prevented by either changing the algorithm’s settings or combining the results of several target run into one.

If the system under study uses caching, then the performance counters tracking the behavior of requests’ latency may be consolidated incorrectly due to the effect of slow requests. This issue can be avoided by categorizing values of such counters using a different classification algorithm.
Chapter 9

The algorithm’s performance study

9.1 Discussion

Since association rule mining technique introduction in early 1990s, its performance has been a concern for the researchers. Virtually all scientific papers, either presenting entirely new or slightly improved association rule mining algorithms, contain a performance study comparing the effectiveness of the newly introduced approach with the prior research in the field; many of these papers suggest the improvements that can positively affect the performance of presented solutions in general or for some specific datasets; every once in a while independent studies comparing performance of these algorithms are being published (see e.g. [24, 42, 50]). Such studies are rather important, as the effectiveness of the association rule mining algorithms is the only thing that differentiates them from each other, if looking from the user’s prospective. All the frequent itemsets generators (e.g. Apriori [2], FP-growth [21], TreeProjection [1], etc.) eventually return the same results after processing the same input data, if implemented correctly. Having this in mind, when selecting a proper association rule mining algorithm an analyst would most probably choose the one that works faster for his tasks.

Performance of such algorithms to a large extent depends from the size and structure of input data. It can be affected by such parameters as the algorithm’s settings (minsup and minconf), average and maximal itemset length, the total number of different items, items distribution in the dataset, etc. For this reason, it is important to thoroughly study the performance of new association rule mining algorithms and define the boundaries of their applicability prior to recommending them for usage in real data analysis tasks. However, this is not always done in a proper way. Thus, Zheng et al. in [50] have studied performance of five well-known association rule mining algorithms on a number of real datasets from different domains. Their results significantly differ from the results of testing those algorithms on artificial datasets generated with the IBM Almaden tool, as they are presented in the respective publications.

This chapter is devoted to studying performance of the Dasha algorithm presented in Chapter 6. Dasha is not a classical association rule mining algorithm in a sense that its output is not a full set of frequent itemsets (or association rules) existing in the underly-
ing data. It is rather a tool for comparative association rules analysis that uses a frequent itemsets generator for intermediate calculations and returns the most important differences between the itemsets in source and target datasets. Thus, its performance cannot be compared directly to the performances of well-known frequent itemsets generators. Also, it is not compared here to the other comparative analysis algorithms based on association rule generation techniques, as to the best of our knowledge there exist no such algorithms apart from the work of Foo et al. [19] describing a performance degradation detection algorithm based on Apriori generator. However, it cannot be used as a reference for this performance study, as Apriori is extremely inefficient for mining performance testing datasets and there is no discussion concerning the algorithm’s performance in Foo’s paper.

Because of the above-mentioned reasons, instead of comparing Dasha performance with the performance of frequent itemsets generators or an Apriori-based performance degradation detection algorithm, performance of the Dasha algorithm is studied on its own in this chapter. The main goals of this study are to provide the analysts interested in using the Dasha algorithm with the initial estimates of its performance on different types of datasets, locate the potential performance bottlenecks and suggest the ways to avoid them.

9.2 Testing setup

To estimate performance of the Dasha algorithm, three experiments on different datasets were completed. The first experiment consisted of testing small artificial datasets that were designed to be the most difficult for processing by the algorithm and thus were representing the worst-case scenario. Analysis of the performance tests executed on these datasets was needed to estimate how fast the performance of the algorithm degrades if the volume of input data increases exponentially.

The second experiment was run on an artificial dataset generated with IBM Almaden tool. Such artificial datasets are often used to performance test association rule mining algorithms, as they resemble market basket data by their structure. The algorithm’s performance degradation rate tracked during this experiment was compared with the worst-case scenario; the effects of changing the algorithm’s settings on the resulting performance were studied.

The third experiment was executed on a real dataset resembling the structure of performance testing data. The performance testing datasets that are described in Chapters 7 and 8 were not used as inputs during this experiment as they do not contain enough transactions and items and are thus processed too fast to be valid performance indicators. Also, contrary to the prior tests, the dataset used in the third experiment contained a lot of differences in patterns between its subsets which allowed to demonstrate the effectiveness of the Dasha algorithm in such setting.

All the performance tests discussed in this chapter were run on a 64-bit Linux computer with a quad-core processor operating at 3.2 GHz and 16 GB of random access memory (RAM). The tests were executed sequentially in a single thread; multi-core structure of a

---

1See Section 5.3 for justification.
2It is implied here that input data is represented not only by the transaction database itself, but also by the frequent itemsets extracted from it.
central processor allowed to dedicate full processing power of a single core to analysis tasks. If not mentioned otherwise, available amount of RAM was sufficient for the testing process and swap files were not used to store temporary data. The tests were executed against a stand-alone version of the Dasha algorithm written in Python programming language\(^3\).

Before running the tests, the algorithm’s implementation was split into three different steps, namely generation, comparison and optimization, and decorated with profiling instructions. The first step included processing raw data files and building FP-trees and support trees\(^4\). It can be said that this step is close by its computational complexity to frequent itemsets generation, as after generation is done, the algorithm contains all the frequent itemsets organized as support trees in RAM. However, additional costs to organize the itemsets into the support trees may take up to 30% of processing time spent at this step in the worst case.

After the initial tests it was decided to track time spent on raw files processing and FP-trees generation separately from support trees generation, because the former operations take nearly constant time to execute on the same pair of data files independently from the algorithm’s settings \(\text{minsup}\) and \(\text{mindiff}\). Changing these settings only affects performance of support trees generation, thus the respective timings are separated from the rest of processing time spent at this step.

Comparison step of the Dasha algorithm consists from comparison of the support trees\(^5\) and association rules generation\(^6\), as these two procedures are closely related.

Optimization step contains association rules analysis and selection\(^7\). After this step, the analysis is done and the most important violations are reported to the analyst for the further investigation.

9.3 Experiments

9.3.1 One-liner datasets

The first experiment was executed using a simple example that represents the worst-case scenario for generation step and nearly-worst-case scenario for comparison step. As follows from the support tree design, the worst-case situation for its processing is when all the itemsets existing in data are frequent and the differences between them occur at the lowest levels of the trees. In this case the algorithm will have to traverse and compare the support trees in full.

9.3.1.1 Experiment goals and hypotheses

Goal 1.1 To model the worst-case scenario for the algorithm’s performance and thus find its lowest performance boundary.

\(^{3}\)Implementation available at https://bitbucket.org/dzzh/python-dasha.

\(^{4}\)See Section 6.4.1.1.

\(^{5}\)See Section 6.4.2.1.

\(^{6}\)See Section 6.4.3.1.

\(^{7}\)See Section 6.4.3.2.
9. THE ALGORITHM’S PERFORMANCE STUDY

**Goal 1.2** To estimate the distribution of processing time spent during the different steps of the algorithm’s execution and its memory consumption.

**Goal 1.3** To identify the performance bottlenecks.

**Hypothesis 1** The algorithm’s performance may decrease super-exponentially with exponential growth of input data size.

### 9.3.1.2 Experiment description

The worst-case scenario can be modeled with a simple and short example. Assume the source dataset\(^8\) contains a single transaction where the items are represented by a number of letters from English alphabet placed in increasing lexicographic order, e.g. \{a, b, c, d, e, f, g, h, i, j\}. The target dataset contains a single transaction that is similar to the transaction in the source but is lacking the last letter presented there, i.e. being \{a, b, c, d, e, f, g, h\} for the given source dataset. If the algorithm is run against these two datasets with \(\text{minsup} = 1\), \(\text{mindiff} = 1\%\), all possible combinations of items existing in the datasets\(^9\) are considered frequent. Hence, they are all added to the support trees and then the trees are compared level-wise until the leaves are reached. All the leaves in the source in this case contain the item that is missing from the target, thus they are considered violations and reported for the optimization. At optimization step, a single rule is chosen out of the violations following the procedure covered in section 6.4.3.2. By simultaneously increasing the length of source and target transactions, it is possible to easily manipulate the number of created frequent itemsets and analyze how the algorithm’s performance changes when this number grows exponentially.

For the optimization step this test case is rather simple, as only one item appears in the resulting association rules. As it will be shown on the other examples later in this chapter, when several different items are reported as violations, performance of this step worsens.

The experiment was performed for \(n\) changing from 10 to 22. Its results are presented in Table 9.1 and plotted in Figure 9.1. Length and itemsets values are taken from the source dataset, for the target dataset \(\text{length}_t = \text{length}_s - 1\), \(\text{itemsets}_t = (\text{itemsets}_s + 1)/2 - 1\). Time to generate FP-trees for these tests was negligible, thus not measured separately from supports tree generation time.

Here and below the timings are shown in seconds rounded to the nearest tenth, memory consumption is shown in megabytes.

### 9.3.1.3 Analysis

From the provided results of the experiment it is clear that for the worst-case scenarios like the one tested with one-liner datasets, the algorithm’s performance decreases faster than datasets complexity increases. While the number of operations needed to generate frequent

---

\(^8\)In Chapter 6 source dataset was called baseline as the algorithm was explained from a performance testing prospective. Here, the algorithm is tested for different domains and source is used as a more generic term.

\(^9\)Each dataset generated this way contains \(2^n - 1\) combinations in total, where \(n\) is transaction length.
9.3. Experiments

itemsets and compare them grows with the same rate as the volume of input data, supplementary algorithm operations such as storing candidate violating rules and optimizing them introduce additional overhead. This validates **Hypothesis 1**.

When source and target datasets are similar, i.e. the differences between them only occur at the lowest levels of the respective support trees, comparison step is much more computationally intensive than generation and optimization steps.

Further investigation of comparison step performance has shown that between 30% and 50% of processing time in it is spent on calculating supports and confidences for the

<table>
<thead>
<tr>
<th>length</th>
<th>itemsets</th>
<th>generation</th>
<th>comparison</th>
<th>optimization</th>
<th>total</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1023</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
<td>0.2</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>2047</td>
<td>0.2</td>
<td>0.3</td>
<td>0.0</td>
<td>0.5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>4095</td>
<td>0.4</td>
<td>0.7</td>
<td>0.0</td>
<td>1.0</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>8191</td>
<td>0.7</td>
<td>1.4</td>
<td>0.1</td>
<td>2.2</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>16383</td>
<td>1.5</td>
<td>3.2</td>
<td>0.1</td>
<td>4.8</td>
<td>24</td>
</tr>
<tr>
<td>15</td>
<td>32767</td>
<td>3.1</td>
<td>7.0</td>
<td>0.2</td>
<td>10.3</td>
<td>40</td>
</tr>
<tr>
<td>16</td>
<td>65535</td>
<td>6.4</td>
<td>15.2</td>
<td>0.4</td>
<td>22.0</td>
<td>76</td>
</tr>
<tr>
<td>17</td>
<td>131071</td>
<td>13.1</td>
<td>33.0</td>
<td>0.8</td>
<td>47.0</td>
<td>143</td>
</tr>
<tr>
<td>18</td>
<td>262143</td>
<td>27.2</td>
<td>75.2</td>
<td>1.7</td>
<td>104.2</td>
<td>286</td>
</tr>
<tr>
<td>19</td>
<td>524287</td>
<td>54.1</td>
<td>164.4</td>
<td>3.3</td>
<td>221.9</td>
<td>555</td>
</tr>
<tr>
<td>20</td>
<td>1048575</td>
<td>109.1</td>
<td>377.6</td>
<td>6.7</td>
<td>493.4</td>
<td>1125</td>
</tr>
<tr>
<td>21</td>
<td>2097151</td>
<td>222.5</td>
<td>922.3</td>
<td>13.7</td>
<td>1158.4</td>
<td>2205</td>
</tr>
<tr>
<td>22</td>
<td>4194303</td>
<td>453.1</td>
<td>2690.1</td>
<td>27.4</td>
<td>3170.1</td>
<td>4380</td>
</tr>
</tbody>
</table>

Table 9.1: The Dasha algorithm’s performance on one-liner datasets.
missing itemsets in a target tree\textsuperscript{10}. The rest of time is spent on the nodes comparison. For the datasets with different structure it can be expected that percentage of time spent on calculating confidences can be even higher, because this operation is rather computationally intensive and is executed whenever an itemset is not found in the target tree. On the other hand, if there are not that many different itemsets in the support trees, this percentage drops. Additional performance tests executed on two equal one-liner datasets of different lengths have shown that this time decreases to zero while the time to compare the support trees does not change in comparison with previously run tests.

Previously in this section it was mentioned that these artificial datasets represent a nearly-worst-case situation for comparison step. Performance of this step depends both from a number of node comparisons and a number of association rules generated out of found differences. Thus, the worst-case scenario for this step should contain the maximal number of comparisons and the maximal number of generated rules, meaning one rule per comparison. It is relatively easy to find a pair of datasets with the maximal number of comparisons: one-liner datasets under study are a perfect example of such a pair. Also, it is easy to find a pair of datasets where a number of generated rules equals to the number of comparisons: an example of such a pair will be two completely different datasets, e.g. \{a,b\} and \{c,d\}. Only two comparisons will be performed for this pair of datasets and two rules will be generated. However, it is simply not possible to find a pair of datasets where both the number of comparisons the number and rule generations will be maximal as the rules are only generated when the difference is found. Thus, depending on the structure of data, relation between the number of comparisons and rule generations may be very different. Either of these operations can consume most time spent on comparison step.

Another interesting variation of a one-liner datasets test is a situation when a missing item is located not in the end of a source dataset, but rather in front or in the middle, or when several items are missing. This situation is much more likely to occur in real datasets. In this case time spent on comparison step decreases significantly. For example, it takes around 922 seconds to compare the support trees and generate the association rules if the length of a source transaction equals to 21 (letters from a to u) and the missing element is the last one (u). However, if the missing element is located in the middle of a source dataset (k), it only takes around 214 seconds to complete the comparison step. If the missing element is the first one (a), comparison step takes 219 seconds. Processing time decreases because the difference in support trees is found earlier and the subtrees of the violating nodes are discarded. The more violations are found, the earlier comparison finishes. If several elements differ between the datasets, the algorithm’s performance increases even more. Thus, if (a,b,s) are missing from the target tree, comparison step takes only 68 seconds. This is an important property of the Dasha algorithm that should allow it to perform better on real-world datasets than on this artificial nearly-worst-case example.

It also turned out that even for the small datasets the algorithm may generate very large volumes of temporary data for the further analysis and this may become a performance bottleneck both in terms of processing time and consumed memory.

\textsuperscript{10}See Section 6.4.3.1.
9.3. Experiments

9.3.1.4 Results

All the goals set before the experiment were reached. The worst-case example for the algorithm’s performance was modeled and studied. It was shown that the algorithm’s performance may drop super-exponentially with the exponential growth of the frequent itemsets number. It was shown, that in the worst-case scenario comparison step is the most computationally intensive, though generation step may also take a lot of time to complete. Supports calculation for the missing itemsets and generation of a large number of frequent itemsets turned out to be performance bottlenecks. Hypothesis 1 was validated during the analysis.

9.3.2 T10.I4.D100K dataset

The next experiment was executed on a well-known T10.I4.D100K dataset generated with IBM Almaden tool\(^{11}\). This dataset is frequently used as a test input for the association rule mining algorithms.

9.3.2.1 Goals and hypotheses

**Goal 2.1** To estimate the algorithm’s performance on a widely used dataset resembling retail data.

**Goal 2.2** To estimate the effect of changing the algorithm’s settings on the resulting performance.

**Hypothesis 2** The algorithm’s performance should significantly depend from changing the \textit{minsup} value.

**Hypothesis 3** Changing \textit{mindiff} value should not have a significant impact on the resulting performance.

9.3.2.2 Experiment description

T10.I4.D100K dataset contains 100 000 transactions with average transaction length equal to 10 items. The items are taken from a set of 1000. To test performance of the Dasha algorithm with different values of \textit{minsup} and \textit{mindiff} settings, the dataset was split in half. The first half was considered a source dataset, the second half was considered a target dataset. Contrary to dense one-liner datasets discussed in the previous section, T10.I4.D100K is a sparse dataset similar to the retail datasets processed in market basket analysis.

The results of performance testing the Dasha algorithm for different values of \textit{minsup} with fixed \textit{mindiff} on T10.I4.D100K dataset are presented in Table 9.2 and plotted in Figure 9.2. For this dataset, time spent on FP-trees generation was equal to approximately 14.5 seconds for all combinations of the applied settings. This time is included in \textit{total} time but is not considered in \textit{generation} time to make the trend more representative. The number of

\(^{11}\text{The dataset was obtained from } \url{http://fimi.ua.ac.be/data/}, \text{IBM Almaden generator is no longer available.}\)
9. The Algorithm’s Performance Study

generated frequent itemsets is taken for the source half of the dataset. For the target half the number of frequent itemsets was slightly higher in all cases due to decreased minsup\textsuperscript{12}.

The results of performance testing the Dasha algorithm for changing mindiff with two different minsup values are shown in Table 9.3 and plotted in Figure 9.3. As for the fixed mindiff tests, FP-trees generation time was around 14.5 seconds in all cases. This time is not considered in generation step measurements.

<table>
<thead>
<tr>
<th>minsup</th>
<th>mindiff</th>
<th>itemsets</th>
<th>generation</th>
<th>comparison</th>
<th>optimization</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00%</td>
<td>5%</td>
<td>150</td>
<td>19.9</td>
<td>0.0</td>
<td>0.5</td>
<td>34.9</td>
</tr>
<tr>
<td>1.00%</td>
<td>5%</td>
<td>385</td>
<td>56.7</td>
<td>0.1</td>
<td>2.1</td>
<td>73.5</td>
</tr>
<tr>
<td>0.80%</td>
<td>5%</td>
<td>503</td>
<td>66.1</td>
<td>0.2</td>
<td>3.0</td>
<td>83.8</td>
</tr>
<tr>
<td>0.60%</td>
<td>5%</td>
<td>758</td>
<td>77.7</td>
<td>0.4</td>
<td>4.7</td>
<td>97.2</td>
</tr>
<tr>
<td>0.50%</td>
<td>5%</td>
<td>1079</td>
<td>87.7</td>
<td>0.7</td>
<td>6.8</td>
<td>109.6</td>
</tr>
<tr>
<td>0.40%</td>
<td>5%</td>
<td>2004</td>
<td>104.3</td>
<td>1.5</td>
<td>9.1</td>
<td>129.3</td>
</tr>
<tr>
<td>0.30%</td>
<td>5%</td>
<td>4312</td>
<td>137.2</td>
<td>3.7</td>
<td>12.9</td>
<td>168.1</td>
</tr>
<tr>
<td>0.20%</td>
<td>5%</td>
<td>13259</td>
<td>196.9</td>
<td>10.2</td>
<td>18.6</td>
<td>240.3</td>
</tr>
<tr>
<td>0.10%</td>
<td>5%</td>
<td>27968</td>
<td>264.8</td>
<td>24.5</td>
<td>24.4</td>
<td>328.2</td>
</tr>
<tr>
<td>0.05%</td>
<td>5%</td>
<td>54333</td>
<td>331.9</td>
<td>59.5</td>
<td>36.1</td>
<td>442</td>
</tr>
<tr>
<td>0.02%</td>
<td>5%</td>
<td>136813</td>
<td>448.2</td>
<td>211.8</td>
<td>84.5</td>
<td>759</td>
</tr>
<tr>
<td>0.01%</td>
<td>5%</td>
<td>575626</td>
<td>800.1</td>
<td>706.5</td>
<td>178.3</td>
<td>1699.7</td>
</tr>
</tbody>
</table>

Table 9.2: The Dasha algorithm’s performance on T10.I4.D100K with fixed mindiff.

Figure 9.2: The Dasha algorithm’s performance on T10.I4.D100K with fixed mindiff.

\textsuperscript{12}See Section 6.4.1.2 for more details on minsup adjustment for the target dataset.
9.3. Experiments

<table>
<thead>
<tr>
<th>minsup</th>
<th>mindiff</th>
<th>itemsets</th>
<th>generate</th>
<th>compare</th>
<th>optimize</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05%</td>
<td>50%</td>
<td>54333</td>
<td>370.7</td>
<td>27.9</td>
<td>10.6</td>
<td>423.8</td>
</tr>
<tr>
<td>0.05%</td>
<td>40%</td>
<td>54333</td>
<td>353.9</td>
<td>29.5</td>
<td>15.1</td>
<td>413.8</td>
</tr>
<tr>
<td>0.05%</td>
<td>30%</td>
<td>54333</td>
<td>345.6</td>
<td>37.2</td>
<td>20.6</td>
<td>417.8</td>
</tr>
<tr>
<td>0.05%</td>
<td>20%</td>
<td>54333</td>
<td>337.8</td>
<td>50.3</td>
<td>28.8</td>
<td>431.3</td>
</tr>
<tr>
<td>0.05%</td>
<td>10%</td>
<td>54333</td>
<td>333.5</td>
<td>64</td>
<td>37.8</td>
<td>449.7</td>
</tr>
<tr>
<td>0.05%</td>
<td>5%</td>
<td>54333</td>
<td>333.3</td>
<td>59.8</td>
<td>36.4</td>
<td>444.4</td>
</tr>
<tr>
<td>0.02%</td>
<td>50%</td>
<td>136813</td>
<td>557.7</td>
<td>127.5</td>
<td>38.6</td>
<td>738.3</td>
</tr>
<tr>
<td>0.02%</td>
<td>40%</td>
<td>136813</td>
<td>510.1</td>
<td>158</td>
<td>51</td>
<td>733.6</td>
</tr>
<tr>
<td>0.02%</td>
<td>30%</td>
<td>136813</td>
<td>482.5</td>
<td>192</td>
<td>64.7</td>
<td>753.7</td>
</tr>
<tr>
<td>0.02%</td>
<td>20%</td>
<td>136813</td>
<td>463.6</td>
<td>227.6</td>
<td>80.3</td>
<td>786.1</td>
</tr>
<tr>
<td>0.02%</td>
<td>10%</td>
<td>136813</td>
<td>448.1</td>
<td>242.5</td>
<td>95.3</td>
<td>800.3</td>
</tr>
<tr>
<td>0.02%</td>
<td>5%</td>
<td>136813</td>
<td>454.5</td>
<td>213.1</td>
<td>84.3</td>
<td>766.5</td>
</tr>
</tbody>
</table>

Table 9.3: The Dasha algorithm’s performance on T10.I4.D100K with fixed minsup.

Figure 9.3: The Dasha algorithm’s performance on T10.I4.D100K with fixed minsup.

9.3.2.3 General analysis

For performance testing such datasets, generation of frequent itemsets can be a rather challenging task, as time spent on generation step of the Dasha algorithm is at least twice as large than plain generation of such itemsets from the whole dataset using the FP-growth algorithm. In Dasha, frequent itemsets are generated for each half of a dataset and this procedure is almost as complex as generating the frequent itemsets from the whole dataset. Afterwards, the generated itemsets have to be organized in a form of support trees. This procedure can also be rather computationally intensive. From this it follows that the Dasha algorithm can only be applied to the datasets for which it is possible to extract the frequent
itemsets in reasonable time. One of the efficient measures to extract such itemsets from the large datasets containing long transactions is to increase minimum support and confidence thresholds. However, it leads to precision loss. If this is not affordable, then the algorithm cannot be applied. Thus, it was not possible to test the available implementation of the Dasha algorithm with small enough \textit{minsup} values on an artificial dataset T40.I10.D100K that contains 40 items in a transaction on average, because frequent itemsets generation procedure was taking too long. For T10.14.D100K dataset time spent on frequent itemsets generation was much less.

The results of tests executed on real performance testing datasets, that are discussed in the following chapters, indicate that the algorithm’s performance is sufficient for them. If a dataset contains around 20 metrics, and \textit{minsup} is reasonably high, the analysis is performed in a matter of several seconds. However, for wider performance testing datasets or when applying the algorithm in other domains, e.g. for comparative market basket analysis, its performance may be insufficient for the analyst’s tasks.

\textbf{9.3.2.4 Analysis for the fixed \textit{minsup} case}

It is clear from the experiment’s results that the lower \textit{minsup} is, the longer it takes to complete the analysis. This is a rather straightforward observation, as decreasing \textit{minsup} immediately leads to generation of a larger number of frequent itemsets in both source and target datasets. Thus, the results of the experiment validate \textbf{Hypothesis 2}.

Another important observation is that, contrary to the one-liner example, for these artificial datasets total analysis time increases with a slower rate than the number of frequent itemsets does. This happens because in this dataset the differences are detected not only at the lower levels of the support trees. When a difference is detected, the subtrees having violating nodes as their roots are discarded and are not compared further which decreases processing time.

Up to a certain moment \textit{generation} step is the most time-consuming. However, when it is needed to detect all the violations including the smallest ones in rather similar datasets (and the datasets used in this set of tests are similar as they are obtained by splitting a single dataset in half), \textit{comparison} step may become much more time-consuming. Further investigation has shown that around 50\% of time during this step is spent on calculating supports for the items missing from the target tree.

\textbf{9.3.2.5 Analysis for the fixed \textit{mindiff} case}

Contrary to the previous case, changing \textit{mindiff} has almost no effect on total analysis time. This happens because the most time-consuming analysis step in a provided example is \textit{generation} that is affected by \textit{mindiff} changes less than after changing \textit{minsup}. When \textit{mindiff} increases, the number of frequent itemsets in the target dataset increases as well, but does not change in the source dataset.

Also, the higher \textit{mindiff} is, the lower is the number of resulting violations. This happens because the higher is the threshold for comparing two itemsets, the more of them are considered similar, hence more comparisons are performed. When \textit{mindiff} increases, differ-
ences are found on lower levels of the respective support trees. Because of this, *comparison* and *optimization* steps tend to take longer when *mindiff* decreases. However, this change is hard to notice in the results for the total analysis time, as at the same time *generation* step is completed faster.

Thus, from the results of the experiment it can be concluded that Hypothesis 3 is valid.

### 9.3.2.6 Results

The results of the experiment indicate that despite the fact that the algorithm’s performance still decreases significantly when a number of frequent itemsets in data grows, the rate of such degradation is much less than in the worst-case scenario. Contrary to the earlier experiment, *generation* step was the most computationally intensive during this set of tests; computation time of *comparison* step nearly reached it only with a very low *minsup* value.

The algorithm’s performance to a large extent depends from the chosen *minsup* value. Changing *mindiff*, however, does not affect it a lot.

Both goals set before the experiment were reached. Hypothesis 2 and Hypothesis 3 were validated during the analysis of its results.

### 9.3.3 Mushroom dataset

The final experiment described in this chapter was run on Mushroom dataset\(^\text{13}\). This dataset resembles performance testing datasets by its structure, because each transaction in it contains the equal number of nominal items.

#### 9.3.3.1 Goals and hypotheses

**Goal 3.1** To study performance of the Dasha algorithm if comparing two datasets with high variance in the itemsets.

**Goal 3.2** To analyze the algorithm’s performance if processing a large number of frequent itemsets.

**Hypothesis 4** *Comparison* and *optimization* steps should take a small fraction of total execution time, no matter what the algorithm’s settings are and how many frequent itemsets are generated.

#### 9.3.3.2 Experiment description

Mushroom dataset contains values for 22 different characteristics of over 8000 different mushroom species. All the transactions in Mushroom dataset are of the same length and contain nominal values that look similarly to the values of performance counters after their consolidation and classification. Before running the experiment, the Mushroom dataset was

\(^{13}\)Reference dataset can be found at [http://archive.ics.uci.edu/ml/datasets/Mushroom](http://archive.ics.uci.edu/ml/datasets/Mushroom). In this study, a version from [http://fimi.ua.ac.be/data/](http://fimi.ua.ac.be/data/) was used, as attributes categories in it are converted to a numerical sequence that is more suitable for the analysis.
split in half the same way as T10.I4.D100K dataset discussed earlier. The first half was considered a source dataset, the second half was considered a target dataset.

At \( \text{minsup} = 6\% \), available amount of free RAM was not enough to fit all temporary data, swapping to disk was used to complete the analysis. Because of heavy swapping, no measurements with \( \text{minsup} < 6\% \) were made, as they would take significantly longer.

The results of performance testing the Dasha algorithm on Mushroom dataset are presented in Table 9.4. Total time and memory consumption for different \( \text{minsup} \) values are plotted in Figure 9.4.

In this experiment, several adjustments to the presentation of the results were made. First of all, only total analysis time is shown in Table 9.4. This change was made because almost all the processing time during the tests was spent on generating the support trees. Time spent on the other operations never exceeded 2 seconds in total and was thus considered negligible. Secondly, instead of tracking the number of frequent itemsets in the source dataset, the total number of frequent itemsets was tracked. This was done because the ratios of itemsets number in source and target datasets were very different for different \( \text{minsup} \) values. The total number of generated itemsets was considered a more reliable metric.

<table>
<thead>
<tr>
<th>( \text{minsup} )</th>
<th>( \text{mindiff} )</th>
<th>( \text{itemsets} )</th>
<th>( \text{total} )</th>
<th>( \text{memory} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>5</td>
<td>1063170</td>
<td>82.3</td>
<td>882</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>1192594</td>
<td>93.9</td>
<td>989</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>1241790</td>
<td>97.5</td>
<td>1005</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>1786498</td>
<td>137.2</td>
<td>1467</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>2606130</td>
<td>200.9</td>
<td>2058</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>6329706</td>
<td>491.2</td>
<td>4916</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>8524214</td>
<td>667.2</td>
<td>6627</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>19203334</td>
<td>1489.5</td>
<td>14911</td>
</tr>
</tbody>
</table>

Table 9.4: The Dasha algorithm’s performance on Mushroom dataset.

### 9.3.3.3 Analysis

In neither of performance tests executed during this experiment time spent during comparison and optimization steps exceeded a small fraction (<1%) of the support trees generation time. This validates **Hypothesis 4**.

Processing Mushroom dataset involved working with a large number of frequent itemsets that were consuming a lot of memory. While one-liner datasets processing also included generation of many frequent itemsets, they used less memory due to the shorter average length of an itemset\(^{14}\).

Similarly to T10.I4.D100K testing, when \( \text{minsup} \) decreases, the number of frequent itemsets and time to process them grows very fast. If \( \text{minsup} \) decreases with constant rate,\(^{14}\).

\(^{14}\)The attribute categories in Mushroom dataset are numeric, but they were treated as strings during these tests to artificially increase memory consumption.
9.4. Measures to improve performance

Figure 9.4: The Dasha algorithm’s performance on Mushroom dataset.

then the lower it is, the faster number of generated itemsets and execution time grow, as relative difference between two values of a setting becomes higher.

From the itemsets-memory plot shown in Figure 9.4 it is clear that when the algorithm lacks operating memory and has to partially store the frequent itemsets at disk storage, the algorithm’s performance starts to deteriorate faster. This observation once again shows that generation of frequent itemsets may become a severe performance bottleneck. The ways to avoid such bottlenecks are discussed in section 9.4.

9.3.3.4 Results

Both goals set before the experiment were reached. It was shown that when source and target datasets have significant difference in their frequent itemsets, the Dasha algorithm is able to compare the respective support trees extremely fast and almost all the time during the algorithm’s execution it is busy with extracting frequent itemsets from the transactions and organizing them into support trees. This observation may be used for optimizing the algorithm for better performance.

Hypothesis 4 was validated during the experiment.

9.4 Measures to improve performance

The Dasha algorithm presented in Chapter 6 is rather complex as it consists of three different steps with each step containing several different procedures. Three experiments, described earlier in this chapter, have indicated that performance of the algorithm depends significantly from the structure of input data. Performance bottlenecks may occur at different places and can drastically increase analysis time. Also, the algorithm contains a number of changeable settings that can also affect its performance.
A description of the algorithm as it is presented in Chapter 6 and its reference implementation tested in this chapter are rather straightforward and were implemented with an intent to demonstrate the approach rather than to optimize it for all the possible inputs, as such optimizations would significantly complicate the logic behind the algorithm and would make it much more difficult for understanding.

This section contains a description of measures that can be taken to improve performance of the reference algorithm implementation. It suggests both the general measures that should significantly improve the overall performance for nearly all possible inputs and specific measures to speed up the algorithm in the known bottlenecks.

9.4.1 General measures

**Changing the algorithm’s settings** The easiest to implement and the most obvious measure to improve Dasha performance on a certain dataset is changing the algorithm’s settings. Thus, when \( \text{minsup} \) increases, the algorithm’s execution time decreases, as higher \( \text{minsup} \) values lead to generation of smaller number of frequent itemsets. Increasing \( \text{mindiff} \) has less impact on the algorithm’s performance but can also be useful for certain datasets.

Changing the algorithm’s settings may significantly improve performance of all the steps. However, it is worth to notice that for performance testing datasets it is recommended to keep \( \text{minsup} \) and \( \text{mindiff} \) rather high anyway because of a need to mitigate noise\(^{15}\), thus this measure is recommended mainly for other domains and can be applied in the situations when it is allowed not to track minor changes in data. This measure may be very effective, but it leads to certain loss of precision, hence it should not be applied blindly.

**Parallelization** Another efficient measure that would help to significantly improve the algorithm’s performance is parallelizing its execution. Because of relying on simple tree-like data structures, the Dasha algorithm can be parallelized in an easy and effective way at all the steps of its execution.

Thus, building FP-tree and support tree for the source dataset can be done in parallel with building these trees for the target dataset at generation step, as these procedures do not depend from each other. Instead of the reference FP-growth algorithm’s implementation used for support trees generation in the described version of the Dasha algorithm, one of its parallel implementations can be used (see e.g. [11, 31, 40]).

During comparison step, the support tree built from the source dataset can be divided into an arbitrary number of subtrees that may be compared with the respective subtrees in the target dataset in parallel following divide-and-conquer approach. To compare two subtrees with their roots having the same path to the full support tree root, it is only needed to know this path and copy read-only instances of the respective FP-trees to the workers. Dedicated workers can be allocated to calculate supports for the itemsets missing from the target support tree, as this is a heavy operation that may slow down trees comparison process.

\(^{15}\) Adjusting the algorithm’s settings to remove noise is discussed in more details in Chapter 7.
9.4. Measures to improve performance

To speed up optimization step, candidate rules may be stored not in a single list, but rather in a hash table of lists indexed by their violating element. These structures may then be processed in parallel to find the most important violating rules faster.

9.4.2 Avoiding performance bottlenecks

9.4.2.1 Frequent itemsets generation

When the frequent itemsets are long, it may take a lot of time and space to generate the support trees out of them. Contrary to the FP-tree, the support tree may grow exponentially with frequent itemsets length because of its prefix structure. This means that for the datasets with very long itemsets having only minor differences in data, the algorithm may be rather slow, as it was shown for one-liner datasets in section 9.3.1. One of the measures to prevent it is to remove the items with extremely high supports from the datasets before generating the frequent itemsets for further processing.

Situations like this don’t usually happen in market basket analysis, but in performance analysis it is pretty common that a value of a certain metric is stable and occurs in all or almost all the transactions. If considering such metrics in the analysis, each of them will double the size of the resulting support tree but will not affect the algorithm’s precision. Items with \( \text{support} = 100\% \) in baseline and target datasets can be removed from the datasets without additional checks, for items with support less than 100\% but higher than the reasonable pre-defined threshold (95-97\%) additional comparison of behavior in baseline and target datasets can be performed before making a decision about their exclusion from the datasets. After the association rules are generated, these items should be added to their left-hand sides.

9.4.2.2 Building support trees

While testing the algorithm’s performance on Mushroom dataset in section 9.3.3, it turned out that even processing of rather small datasets may sometimes lead to significant memory consumption for building the support trees. While generation of frequent itemsets and support trees comparison may complete in reasonable time for such setup, the need to keep large amount of data in memory during the analysis may significantly increase analysis time due to the need to store these data on disks, not in RAM. To avoid this issue, it is possible to generate the support trees on demand rather then pre-calculate them in full before moving to \textit{comparison} step. Such partial generation of the support trees can be implemented in two different ways: for breadth-first or depth-first traversals of the trees.

**Partial support trees generation for breadth-first traversal** At the described implementation of the Dasha algorithm, the trees are compared in breadth-first order level-wise. In this situation it is possible to interrupt frequent itemsets generation in conditional FP-trees upon reaching certain depth. A link to the conditional FP-tree with a reference to a node where the process was interrupted should be stored in the support tree. After reaching the leaf nodes of the support tree the interrupted process may be resumed from the saved position if all the generated itemsets in the respective subtrees are considered similar.
Apart from saving memory, a performance improvement of this approach is that processing time will not be spent on generating the itemsets that will never be compared. If certain subtrees are considered different and are discarded, the algorithm does not need to generate the frequent itemsets contained in them.

**Partial support trees generation for depth-first traversal** Changing the support trees traversal algorithm to use depth-first order would not affect the resulting precision of the Dasha algorithm, but would allow to build the support trees branch by branch (with first-order root children becoming the roots of the respective branches), compare the branches and immediately delete them after the comparison is done.

To speed up the process of inserting an itemset into the support tree, previously inserted node can be remembered until the new itemset is being inserted. The FP-growth algorithm processes the frequent itemsets in depth-first order, thus trying first to insert a new itemset as a child of a previous node may be faster than searching for the right position from the root in all cases.

**Avoiding support trees generation** It is also possible not to generate the support trees at all. Instead, the frequent itemsets can be compared one by one. As it was stated in section 6.1, support trees serve as a caching layer, they do not store any information other than already contained in the FP-trees. To compare the datasets this way, the process should be the following. After retrieving a frequent itemset from the FP-tree, its correspondence in the target FP-tree should be found with the same procedure that is used for calculating supports for the missing itemsets\(^\text{16}\). If the difference is found, generation of the frequent itemsets in the current conditional FP-tree should terminate.

This approach should drastically reduce the amount of RAM needed for the algorithm to operate but will most probably work slower than with the support trees, as calculating itemset supports directly from the FP-trees is a rather computationally intensive operation. A separate case study is needed to validate this approach and estimate its performance on different datasets.

**9.4.2.3 Calculating supports for missing itemsets**

As it was mentioned earlier, calculating supports for the frequent itemsets missing from the support tree directly in the FP-tree for a target dataset is a heavy operation. In the Dasha algorithm it is used to find how significantly the violating nodes differ between the source and the target. This information is used for results optimization to return only the most important association rules. For situations where it is needed to return all the violating association rules, this step can be omitted to speed up the algorithm. In this case, however, support value for the association rule antecedent in the target tree will remain unknown, it will only be known that it significantly differs from the source value.

If comparison step takes much longer than generation step, it is possible to adjust the formula for decreasing target’s minsup\(^\text{17}\). If minsup decreases, more itemsets will be gener-

\(^{16}\)See section 6.4.3.1.

\(^{17}\)See section 6.4.1.2.
ated for the target dataset and less of them will be missing from the respective support tree.
If target’s minsup equals to 1, the algorithm will never find a missing itemset in the FP-tree, so for this situation this part of the algorithm’s logic can also be omitted. It is needed to mention, though, that decreasing target’s minsup leads to higher memory consumption and longer generation of target’s support tree.

As follows from the Dasha algorithm’s design, the discussed suggestions to improve the algorithm’s performance should significantly speed-up the algorithm for most of the datasets suitable for analysis with the association rule mining. However, they were not implemented and tested in the scope of this graduation project and are left for future work.

9.5 Summary

The results of performance tests, executed and analyzed while working on a project, indicate, that the Dasha algorithm, as all the other association rule mining algorithms, may have poor performance on certain datasets. However, in most situations it holds that if system performance is sufficient to generate frequent itemsets out of source data, it should also be sufficient for the datasets comparison algorithm presented in Chapter 6. The Dasha algorithm may be used in different domains to solve various comparative analysis tasks, and depending on the requirements to the final results, certain algorithm steps can be tuned to improve the resulting performance or fit the algorithm into the target domain, as the provided solution is rather general and allows for many different modifications.

Performance bottlenecks may appear at several places during generation and comparison steps. Optimization step is usually less computationally intensive, but may also take a lot of time while processing certain datasets.

Before starting to optimize the algorithm for specific analysis tasks, it is important to understand that Dasha has modular design and was initially developed for analyzing performance testing datasets that have a specific structure of input data\textsuperscript{18}. Structure of data in the other domains and the tasks that have to be solved by the analysts are different and may require changing implementations of some parts of the algorithm. This may significantly affect the overall algorithm’s performance in positive or negative way.

\textsuperscript{18}Read more about performance testing challenge in Section 5.3.
Chapter 10

Conclusions and future work

This is the final chapter of the thesis report that gives an overview of the project’s contributions and contains directions for future work. A reflection on the results of this study is also presented here.

10.1 Contributions

The main contribution of this study is an implementation and presentation of a novel data mining algorithm for comparative association rules analysis, which is called Dasha. This algorithm is a modular solution for comparing patterns in transaction databases and reporting the most notable differences between them. An approach to automatically detect performance degradations in evolving web applications was developed using Dasha as its underlying analysis engine.

To support performance analysis tasks, a web application for storing and analyzing the results of the performance tests was built. This application consists of two parts. The first of them is a performance testing repository, which is an online storage for load test results. It helps performance analysts to easily access the results of load tests ever run against their target software, visualizes behavior of the tracked performance counters and prepares data for the analysis module. The repository supports working with several different file formats and can be easily extended to support the performance engineers with more analysis tasks. A number of data classifiers were built and tested with the repository.

The second part of a web application is the analysis module that consists of the Dasha algorithm’s implementation set up for processing performance testing datasets and of user interface visualizing the results of performance analysis.

In addition to the web application, the Dasha algorithm was implemented as a stand-alone application. This implementation is more suitable for testing and is easier for understanding than the code of a web application discussed earlier.

The discussed approach to compare transaction databases is based on a set-enumeration tree containing supports for the frequent itemsets that exist in the underlying data. This data structure is introduced in this paper and is called a support tree. An algorithm to build
this tree from an FP-tree, compare two support tree instances and optimize the results of comparison was presented.

10.2 Discussion/Reflection

10.2.1 Researched answers

The main research question investigated in the course of this study was *How to identify performance degradations and their causes in enterprise web applications?* This question included four sub-questions. The answers to these sub-questions as well as the answer to the main question are provided in this section.

The first sub-question  The first sub-question posed before the beginning of this research project was *How to store the results of performance tests from different sources for the further analysis?* To answer it, a prototype of a performance testing repository was built and described in Chapter 4. The performance testing repository is a web application aimed to store the results of load tests run to estimate the performance of the target software. Apart from merely keeping these results in a centralized storage, a provided implementation of the repository also visualizes them and prepares the data for analysis using data consolidation techniques. The performance testing repository supports several different classification algorithms and supplies a performance engineer with basic analysis tools that help to immediately detect severe performance degradations.

The second sub-question  The second sub-question was *What useful information can be extracted from the performance test results and what data mining techniques can be used for its retrieval and analysis?* It can be answered the following way. First of all, performance test results contain values of the performance counters that are captured at different moments in time during the performance test runs. These values can be directly compared to the values from the other runs to make conclusions about the differences between them. Secondly, it can be said that apart from the counter values themselves, the test results also contain correlations between these values. These correlations can be extracted and analyzed using several different data mining techniques with one of them being association rules analysis. Application of association rule mining algorithms for performance analysis tasks was discussed in Chapter 5.

When analyzing the applicability of association rule mining for performance engineering tasks, several different algorithms implementing this technique were studied. It has been shown in this document, that the most well-known Apriori algorithm is not suitable for working with the performance testing data, because it is very inefficient in processing long transactions appearing in the respective datasets. The FP-growth algorithm, on the contrary, turned out to be much more efficient and was chosen as the underlying frequent itemsets generator for the implemented performance analysis algorithm. Also, it was claimed that applying association rule mining techniques to detecting performance degradations requires to compare the association rules generated out of different datasets between each other. While the problem of generating such rules is being studied by many researchers
for nearly two decades already and there exist several different approaches to solve it, no well-known and adopted ways to perform comparative association rules analysis have been designed yet. This study aims to provide an approach to do so.

The third sub-question  

The third sub-question was *How to detect performance degradations in a performance test run having the results of previous (successful) runs? Is association rule mining a suitable technique for this task?* If the difference in performance between two runs is apparent, it can easily be detected by comparing aggregated counter values from different runs between each other. For most of the counters, the results of such simple comparisons will show notable differences. If, however, the performance differences are minor, more advanced techniques are needed to detect the violations. Association rule mining is a suitable technique for this task, as it can detect even minor violations in data and report those combinations of performance counters, that indicate the regressions in the clearest way. A research prototype for automatic detection of performance regressions using association rule mining was developed during this project. It is described in Chapter 6. To validate the suggested approach, two case studies were conducted. Their results are presented in Chapters 7 and 8.

The Dasha algorithm, presented in Chapter 6, approaches the automated detection of performance regressions problem and contains four steps, namely building the support trees, comparing them, deriving the association rules from the obtained difference and optimizing the results. The support tree introduced in this paper is a *set-enumeration tree* serving as a caching layer for the frequent itemsets contained in data. All the steps of the suggested algorithm have been thoroughly explained in this study.

The fourth sub-question  

The fourth sub-question posed before this study was *How to represent the results of automatic performance testing analysis to provide the analysts with the detected performance degradations and their justification?* To answer it, the user interface of the analysis module was designed for the web application developed in the scope of this research project. This interface lists the most important violations of the performance counter values between the compared test runs and explains to the performance engineer how these values are distributed in the results and why these particular combinations of performance counters were raised as performance degradations. The interface is presented in Chapter 6.

The main question  

Having answered the sub-questions, it is possible now to answer the main research question. Performance degradations in evolving web applications can be detected using association rule mining techniques. This study contains a description of a novel algorithm to complete this task and an implementation of a web application using this algorithm for the analysis of the performance tests. Two case studies performed on artificial and real performance testing data validate the suitability of the presented approach. A performance study investigates the properties of the presented algorithm and imposes certain limitations on its applicability. While the presented approach does not indicate the causes
of performance regressions explicitly, the correlations reported by the Dasha algorithm may give some insights on why performance degrades in certain test runs.

10.2.2 Threats to validity

Two case studies completed during this research project and described in Chapters 7 and 8 indicate that using the presented approach it is possible to automatically detect performance regressions in evolving web applications with high precision and satisfactory performance. However, there are certain threats to validity that impose some limitations on the presented solution.

One of such threats is data fluctuations. As it turned out during the case studies, even if the same performance tests are run on the same workstation following the same procedure, their results usually slightly differ due to the stochastic nature of processes in non-real-time operating systems. In some situations this may lead to incorrect classification of values for some of the tracked performance counters and, respectively, to incorrect analysis results. To prevent this, it is needed to capture several baseline runs and compare the target run with the consolidated results for all these runs to obtain more stable results. It is also needed to capture several target runs and choose those of them that do not contain significant deviations from each other.

Another problem related to data fluctuations is that using the discussed approach it is possible to detect even minor violations between the datasets. At the same time, performance testing datasets usually contain a lot of noise that has to be separated from significant results. It was shown in Chapter 7 how this can be done by manipulating the project’s settings. However, for certain datasets it can be rather difficult to mitigate the noise. If the noise is not removed properly, then the final results would either contain some false positives, or would miss certain important violations.

The second threat is the algorithm’s performance. As it was discussed in Chapter 9, the presented solution may be rather slow for certain datasets. A number of measures were suggested to decrease the analysis time. As follows from the design of a presented approach, their implementation should significantly speed up the presented algorithm. However, these measures were not tested on real data, and even after their implementation the algorithm may still be inapplicable for certain datasets, especially those containing very long frequent itemsets.

10.3 Future work

During the research project described in this study, the presented approach to automatically detect performance regressions in evolving web applications has undergone a lot of changes. At all stages of its development, including design, implementation, testing and documenting, many ideas to improve the final result were generated. Some of them were implemented, some were discarded, some were postponed and ended up in the issue tracker. The developed approach to compare transaction databases and a web application implementing it turned out to be rather complex systems leaving a lot of room for improvements and modifications to better fit specific analysis tasks. It was impossible to implement all the ideas
that have appeared while working on the project, as the more improvements were made, the more new ideas were generated. Probably, this happens with most evolving software projects. This section lists a number of improvements that can be implemented to make the presented results better.

10.3.1 Performance testing repository

To improve the performance testing repository presented in Chapter 4, the following can be done. First of all, the repository may be extended with additional data input formats. So far, the application supports working with Apache JMeter files (both in \texttt{jtl} and \texttt{csv} formats) and with several other csv-based formats. Thus, data from RRDtool and OpenTSDB applications in csv format was used as the repository input during the testing activities. To get these data in csv format from the respective storages, custom converters were built. Extending the list of supported data formats would help the engineers executing performance tests using other software to analyze their results with the provided research prototype.

Secondly, it is possible to adjust the repository to track regression direction for the available performance counters. Currently, all the differences between the baseline and target datasets are considered violations. Instead, it is possible to only mark the located differences as violating if the application performance in the target dataset drops comparing to the baseline and do not report the situations when target performance improves. To achieve this, it is needed to know which of the counters’ categories are associated with higher values and indicate that the performance has worsened. This information has to be kept on per-counter basis as it may differ between certain metrics.

Speaking about the counters classification, it is needed to mention another potential improvement, namely using different classification formulae for different counters. Currently, the classification boundaries are calculated following the same algorithm for all the available counters. Thus, with default performance repository settings, all performance counters are classified using \texttt{quartile-stddev} classifier with outlier boundaries set to median $\pm 1$ standard deviation. However, the values of different counters may behave differently. Some of them are more stable and have all their values distributed near the median (in other words, they have low variance), some are less stable and contain many outliers (thus being characterized with high variance). Different algorithms have to be used to classify such counters correctly.

More effort can be put into making it easier to detect broken runs. To be sure that the testing results are reliable, it is needed to compare all the runs assessing the performance of an application release between each other prior to comparing them to the runs executed against a different release. This has to be done as some of the runs may report very different performance in comparison to the other runs. The reasons of this include misconfiguration of a target application and certain infrastructural issues. Current version of a performance testing repository does not have tools for fast and visual comparison of runs testing the same release, thus broken runs detection has to be done manually.
10. CONCLUSIONS AND FUTURE WORK

10.3.2 The Dasha algorithm

The most important task to improve the Dasha algorithm presented in Chapter 6 is to implement the performance-related measures that are discussed in section 9.4. The results of the performance study executed on the reference algorithm’s implementation have shown that this implementation has a number of bottlenecks and they may drastically decrease the overall performance of the presented solution for certain datasets. However, it is expected that the existing performance issues can be successfully mitigated.

It was also mentioned in Chapter 9 that it should be possible to omit the support trees generation and compare the patterns in transaction databases using the FP-trees only. It would be interesting to implement the suggested algorithm and study its behavior and performance in comparison with the algorithm that uses the support trees. It is expected that the algorithm not involving support trees construction will require more processing time but less operative memory. However, this hypothesis has to be validated.

It would be useful to test the Dasha algorithm on more datasets from different domains. From the algorithm’s design and a performance study described in this document it follows that the suggested approach is generic enough to be successfully applied for comparative analysis not only in performance engineering but also in the other fields, e.g. in market basket analysis to find differences in customers’ behavior between different locations or over time. This hypothesis has also to be validated with additional research on real datasets.

10.4 Conclusions

In this Master’s thesis document a novel approach to automatically detect performance regressions in evolving web applications was presented. The underlying research project included development of an open-source performance repository aimed to store the results of performance tests and design of a generic algorithm to compare patterns in transaction databases using association rule mining techniques. This algorithm was used to automatically analyze the differences between performance test results and make conclusions about the regressions contained in them. Two case studies and performance analysis of of the presented approach were completed to assess the applicability of the provided solution for automatic performance regressions detection.


[21] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In Proc. ACM SIGMOD’00, pages 1–12, New York, NY, USA, 2000. ACM.


