Analysis of high-volume traffic using Complex Event Processing and a Domain Specific Language

Improving Priceline.com’s inventory cache performance

Erik Zuidema
Analysis of high-volume traffic using Complex Event Processing and a Domain Specific Language

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Erik Zuidema
born in Nieuwkoop, The Netherlands
Analysis of high-volume traffic using Complex Event Processing and a Domain Specific Language

Author: Erik Zuidema
Student id: 1217798
Email: tu@ezuidema.nl

Abstract

Priceline.com is an online travel agency that provides travel services to customers. With the hotel reservation service, customers can search and book hotels. In the online travel environment it is physically and economically not possible to retrieve the hotel room rates in real-time for every customer request. To overcome this problem the hotel rates are cached, but due to the fact that the suppliers will not send notifications of price changes it is a challenging task to keep the cache up to date.

Analysis of the problem of how to improve the cache's performance, in terms of accuracy and coverage, led to the conclusion that there is no single risk-free overall solution, but that the cache should be enhanced in a step-wise manner minimizing the (financial) impact of a faulty enhancement. The Enhancement Cycle is defined as a number of stages that each step-wise enhancement will go through.

The Cache22 Esper system provides a solution for two of those stages. The system measures the cache performance in a correct manner and predicts the expiration time for a hotel rate. The system uses Complex Event Processing to detect the patterns needed to do the measurements. The system has been generalized and is extended with a Domain Specific Language for a low-effort application to other problems.

The system is running in production and processes around 45 million messages a day. It aggregates and compresses the measurement data without losing expressivity and its predictions improved the cache's accuracy with 7% to above 90%.

Thesis Committee:

Chair: prof. dr. ir. G.J. Houben, Faculty EEMCS, TUDelft
University supervisor: dr. ir. A.J.H. Hidders, Faculty EEMCS, TUDelft
Committee Member: dr. A.E. Zaidman, Faculty EEMCS, TUDelft
This thesis is submitted in partial fulfillment of the requirements for a Master’s Degree in Computer Science and is my final paper for the Delft University of Technology. I am grateful to have been given the privilege and opportunity to enjoy an education at a great university in a historic Dutch city like Delft. I learned a great deal and was able to combine my studies with extra curricular activities such as volunteering for the Study Association, where I served as treasurer, part-time jobs, and spending time with friends. It has been a wonderful period in my life. I would like to thank my parents and my girlfriend for their support and their faith in the choices I made.

Finishing my studies in the United States by doing my Master’s Project at Priceline.com was perfect. Moving to a foreign country and living there for an extended period of time was educational. It gave me a chance to experience American culture as well as its bureaucracy with its demands for paperwork that proved my existence. Priceline.com is a multicultural company and thanks to the different ethnicities in my team, I also got to experience their cultures. It’s the little things you appreciate the most, like the lunches with my colleagues and the team lunches at Indian restaurants. I would like to thank Priceline.com for giving me this extraordinary opportunity. My special thanks go out to the team: Ravi thanks for working so closely together and enabling me to do the things I needed to do with the Priceline infrastructure; Jim thank you for taking the time to extensively test the system in order to get it production ready; Murali and Arnab thanks for the overall explanation of the systems and your support. I profoundly thank Amit for being my principal within Priceline.com and for having such confidence and faith in me, evidenced by the fact that my solution is now running as a production system.

I would like to thank Kurt, Kiran, Jay, Arnab and many others for making me feel more than just a colleague, or an intern for that matter, you made me feel like a friend, and this feeling is mutual.

Jan Hidders, thank you for being my supervisor from TU Delft. You did a great job in guiding and assisting me despite the distance and the six-hour time difference between us. Your willingness to meet on Skype was not only helpful to me but also a great joy for the people around me thanks to the mix of the harsh Dutch G’s and the English words. Additionally, I would like to thank Geert-Jan Houben and Andy Zaidman for being part of my thesis committee and taking the time and effort to read this paper and provide helpful feedback.

At last, I do not even know how start to express my utter most thanks to the Francis-Hulst family. First of all, Paul thank you for bringing me in contact with Priceline.com and that you put in a good word for me. I am thankful for all the practical advice you gave me. Paul and Titia, it was a great support for me to be able to stay at your home in the first months and have the time to look for my own place. It was unbelievably kind of you to lend me a car for
the rest of my stay in America. Titia, I would like to thank you for all your
good care and making me feel like having a second home, even after I moved
out. Charles and Eleanor, thank you for the good times we spent together and
showing me around in New York. I also need to thank Cato for that.

Altogether, this project has been a valuable experience that I will never for-
get and I will be forever grateful to everybody who helped to make it possible.

Erik Zuidema
Delft, the Netherlands
June 28, 2012
Contents

List of Tables vii
List of Figures ix
List of Listings xi

I Introduction 1

1 General 3
   1.1 Priceline.com, Inc. 3
   1.2 Delft University of Technology 4
   1.3 Thesis Overview 4

2 Techniques and Frameworks 7
   2.1 Complex Event Processing 7
   2.2 ExtJS 24

II Priceline.com 29

3 The Company 31
   3.1 Introduction 32
   3.2 Brands 32
   3.3 Affiliates 32
   3.4 Rate Types and Models 33
   3.5 Services 34

4 Processes and Systems 37
   4.1 Overview 37
   4.2 Processes 38
   4.3 Systems 42

III Cache22 45

5 Proceedings 47
   5.1 Problem Context 47
   5.2 Problem Analysis 50
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Cache22 Esper</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>6.2</td>
<td>Why Esper?</td>
</tr>
<tr>
<td>6.3</td>
<td>Phase-I</td>
</tr>
<tr>
<td>6.4</td>
<td>Phase-II</td>
</tr>
<tr>
<td>6.5</td>
<td>Post-Processing</td>
</tr>
<tr>
<td>6.6</td>
<td>Testing</td>
</tr>
<tr>
<td>6.7</td>
<td>Results</td>
</tr>
<tr>
<td>7</td>
<td>Cache22 EsperHA DSL</td>
</tr>
<tr>
<td>7.1</td>
<td>Motivation</td>
</tr>
<tr>
<td>7.2</td>
<td>Design</td>
</tr>
<tr>
<td>7.3</td>
<td>Language Definition</td>
</tr>
<tr>
<td>7.4</td>
<td>Implementation</td>
</tr>
<tr>
<td>7.5</td>
<td>Evaluation</td>
</tr>
<tr>
<td>IV</td>
<td>Conclusion and Future Work</td>
</tr>
<tr>
<td>8</td>
<td>The Sampling System</td>
</tr>
<tr>
<td>8.1</td>
<td>Mechanism</td>
</tr>
<tr>
<td>8.2</td>
<td>Sampling Technique</td>
</tr>
<tr>
<td>8.3</td>
<td>Time-series</td>
</tr>
<tr>
<td>8.4</td>
<td>Design and implementation</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion and Future Work</td>
</tr>
<tr>
<td>9.1</td>
<td>Conclusion</td>
</tr>
<tr>
<td>9.2</td>
<td>Future Work</td>
</tr>
<tr>
<td>Appendices</td>
<td></td>
</tr>
<tr>
<td>Appendix</td>
<td>Esper Code Samples</td>
</tr>
<tr>
<td></td>
<td>ExtJS Code Examples</td>
</tr>
<tr>
<td></td>
<td>Cache22 Esper</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
</tr>
<tr>
<td>Acronyms</td>
<td></td>
</tr>
<tr>
<td>Glossary</td>
<td></td>
</tr>
</tbody>
</table>
List of Tables

2.1 Types of Event Properties ........................................... 14
1 Pattern Operator Precedence [8, p. 153] ............................ 128
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Measuring average temperature by using a database system.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Measuring average temperature by using a CEP system.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Esper’s component overview.</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Visual representation of a \texttt{win: length(5) view}.</td>
<td>19</td>
</tr>
<tr>
<td>2.5</td>
<td>Visual representation of a \texttt{win: length_batch(5) view}.</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>Visual representation of a \texttt{win: time(4 sec) view} \cite{9, p. 30}.</td>
<td>21</td>
</tr>
<tr>
<td>2.7</td>
<td>Visual representation of a \texttt{win: time_batch(4 sec) view} \cite{9, p. 31}.</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Google Maps™ showing rates from the Priceline Group sites.</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Priceline’s architecture overview.</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>The iPad app showing a big amount of hotel rates.</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>A visual representation of the time dependent itinerary properties.</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>The number of request for each combination of \texttt{Advance Purchase (AP) \leq 21} and \texttt{Length Of Stay (LOS) \leq 21}.</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>The Enhancement Cycle.</td>
<td>51</td>
</tr>
<tr>
<td>6.1</td>
<td>High-level component overview of the Cache22 Esper system.</td>
<td>64</td>
</tr>
<tr>
<td>6.2</td>
<td>The Feeders architecture.</td>
<td>67</td>
</tr>
<tr>
<td>6.3</td>
<td>The StatsDataReader architecture.</td>
<td>68</td>
</tr>
<tr>
<td>6.4</td>
<td>The Event architecture.</td>
<td>69</td>
</tr>
<tr>
<td>6.5</td>
<td>The Dispatcher architecture.</td>
<td>71</td>
</tr>
<tr>
<td>6.6</td>
<td>The Processor architecture.</td>
<td>72</td>
</tr>
<tr>
<td>6.7</td>
<td>The Subscriber architecture.</td>
<td>72</td>
</tr>
<tr>
<td>6.8</td>
<td>The OutputEngine architecture.</td>
<td>73</td>
</tr>
<tr>
<td>6.9</td>
<td>The TableHelper architecture.</td>
<td>73</td>
</tr>
<tr>
<td>6.10</td>
<td>The Accuracy problem.</td>
<td>74</td>
</tr>
<tr>
<td>6.11</td>
<td>The design for the Esper replacement in Phase-II.</td>
<td>78</td>
</tr>
<tr>
<td>6.12</td>
<td>The Regression process.</td>
<td>81</td>
</tr>
<tr>
<td>7.1</td>
<td>High-level component overview of the Cache22 EsperHA DSL system.</td>
<td>87</td>
</tr>
<tr>
<td>7.2</td>
<td>Generation Gap Pattern structure \cite{27}.</td>
<td>93</td>
</tr>
<tr>
<td>7.3</td>
<td>The folder structure of the two gen-folders variant \cite{21}.</td>
<td>94</td>
</tr>
<tr>
<td>7.4</td>
<td>The classes that are part of the Esper DSL Library.</td>
<td>95</td>
</tr>
<tr>
<td>7.5</td>
<td>The hierarchy of Readers and Transformers.</td>
<td>96</td>
</tr>
<tr>
<td>7.6</td>
<td>The hierarchy of Events.</td>
<td>97</td>
</tr>
<tr>
<td>7.7</td>
<td>The hierarchy of Bootstrap and Processors.</td>
<td>98</td>
</tr>
<tr>
<td>7.8</td>
<td>The hierarchy of Subscribers.</td>
<td>98</td>
</tr>
</tbody>
</table>
7.9 The hierarchy of the Output Transformers and Table Output 99

8.1 **Baseline** situation; \( RC = \text{Rate Change}, \ G = \text{GDS Shop} \) and \( C = \text{Cache} \) Shop 105

8.2 **Improved** situation; \( rG = \text{Robot GDS Shop}, \ G = \text{GDS Shop} \) and \( C = \text{Cache} \) Shop 106

8.3 False positive 108

8.4 False negative 108

8.5 Early shop 108

8.6 Late shop 108

8.7 Unnecessary shop 108

8.8 A visual representation of a hierarchic stratification 110

8.9 A visual representation of an adaptive polling frequency 111

8.10 The Sampling System design overview 113

8.11 The custom servlet providing the web-service for the Sampling System 114

8.12 The Sampling GUI to administer the system, built in ExtJS 115
List of listings

3. The multi-row object subscriber implementation example.
4. Map event code example.
5. Map event Event Processing Language (EPL) example.
6. Plain Old Java Object (POJO) event code example.
7. POJO event EPL example.
8. Event window using view.
10. Event window using length batch view.
12. Event window using time batch view [9], p. 31.
13. Creating and inserting into a named window.
14. A pattern using many different atoms and operators.
15. Application definition. See Listing 34.
17. View definition. See Listing 36.
20. The pattern to follow for a Cache event.
21. The GDS pattern.
22. The pattern to follow for a Cache event.
23. The Processor syntax with an example.
24. The Statement syntax with an example.
25. The Transformer syntax.
26. The Transformer example.
27. Client-side socket code [10], p. 18–19.
29. The example statement to illustrate the row-by-row delivery.
31. Map and Object array delivery update method signature.
32. Map and Object array delivery update method signature.
33. The index.html code.
34. The Application class instantiation.
35. The (Robot) Controller class instantiation.
36. The (Robot list) view class instantiation.
37. The (Robot) store class instantiation.
38. The (Robot) model class instantiation.
39. The creation of schema, which are internal events.
40. The creation of a the cache window.
<table>
<thead>
<tr>
<th>List number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>The on DbTriggerEvent statement</td>
<td>135</td>
</tr>
<tr>
<td>42</td>
<td>The on DbUpdate statement</td>
<td>135</td>
</tr>
<tr>
<td>43</td>
<td>The select DbUpdate statement</td>
<td>135</td>
</tr>
<tr>
<td>44</td>
<td>The delete expired cache statement</td>
<td>135</td>
</tr>
<tr>
<td>45</td>
<td>The cache clean up event</td>
<td>135</td>
</tr>
<tr>
<td>46</td>
<td>Cache22 EsperHA DSL Syntax definition 1/3</td>
<td>136</td>
</tr>
<tr>
<td>47</td>
<td>Cache22 EsperHA DSL Syntax definition 2/3</td>
<td>137</td>
</tr>
<tr>
<td>48</td>
<td>Cache22 EsperHA DSL Syntax definition 3/3</td>
<td>138</td>
</tr>
</tbody>
</table>
# Introduction

## 1 General

1.1 Priceline.com, Inc. .............................................. 3
1.2 Delft University of Technology .................................. 4
1.3 Thesis Overview .................................................. 4

## 2 Techniques and Frameworks ........................................ 7

2.1 Complex Event Processing ....................................... 7
2.2 ExtJS ............................................................ 24
CHAPTER 1

GENERAL

1.1 Priceline.com, Inc. ............................................. 3
1.2 Delft University of Technology .................................. 4
1.3 Thesis Overview ................................................ 4

This chapter introduces the two stakeholders of the Master’s Project. Priceline.com is the contractor of the project, introduced in Section 1.1 and this project is part of the Master’s Degree of Computer Science at the Delft University of Technology, which is introduced in Section 1.2. The information in Section 1.3 contains an overview of the thesis, and serves as a reading guide.

1.1 Priceline.com, Inc.

Priceline.com is an Online Travel Agency (OTA) based in Norwalk, Connecticut, the United States. The company is part of the Priceline Group; with subsidiaries like Booking.com based in Amsterdam, the Netherlands, and Agoda based in Bangkok, Thailand, which might be familiar to European and Asian readers.

Priceline.com contracted this project and it was executed at their offices in Norwalk. Priceline.com provides a number of travel services. This project was executed for the hotel reservation service. This service allows customers to search for and book hotels. Priceline.com provides the customer with rates from over 210,000 hotels.

Priceline.com’s project description can be summarized as follows:

Typically, in online travel environments it is not physically and economically possible to get the rates from suppliers in real-time for every customer request: a search or booking. To meet the high demand for search requests, the OTAs cache the inventory information on their end. Due to unavailability of notification functionality from suppliers during a price change event, it becomes a
challenging task to prevent the cache from going out of sync with the supplier systems.

Priceline.com was looking to develop a better way to keep this cached inventory information fresh and accurate. This thesis describes how the problem was approached and explains the solutions found.

1.2 Delft University of Technology

Delft University of Technology is a public university in the city of Delft, the Netherlands. The university is providing technical education for over 170 years, being founded in 1842; and officially changed its name to Delft University of Technology or TU Delft in 1986. The university had over 17,000 students in 2011, of which over 3,500 first year students. About 16% of the student population consists of international students.

The high quality of TU Delft’s research and teaching is renowned. The university ranked 49 in the Times Higher Education World Reputation Rankings 2011, being the third European technical university and the highest-ranking Dutch university.

The Electrical Engineering, Mathematics and Computer Science (EEMCS) faculty has around 1600 students. The faculty has six departments and this thesis is part of the Software Technology department. The thesis is a partial fulfillment of the requirements for the degree of Master of Science. The university uses the European Credit Transfer System (ECTS) points; there are 60 credit points or 1,680 hours of study in an academic year. The Master’s Degree takes two years and the Master’s Thesis or Master’s Project is the finalizing stage of the program. The Master’s project is 45 credit points or 1,260 hours. The length of the project allows students to demonstrate their ability to solve a research or engineering problem.

1.3 Thesis Overview

The thesis consists of five parts. The introduction, Part I, provides background information on the techniques and frameworks used during the project—see Chapter 2. For people unfamiliar with Esper, the section on Esper is an advised read, because it will aid in the overall understanding of the project. The section about ExtJS is of importance to those who want to better understand the Sampling GUI.

Part II contains information about Priceline.com. Chapter 3 describes the services Priceline.com offers and introduces the terminology used within the travel industry and Priceline.com. Chapter 4 provides the context for the project and explains the processes and systems involved. Part II can be skipped by readers who are familiar with the Priceline.com infrastructure.

Part III consists of three chapters that explain the project itself. Chapter 5 provides the problem specific contexts and the outcome of the problem analysis. The problem analysis led to the definition of an enhancement cycle;
being a number of stages which an improvement for the cache system would go through. Chapter 6 explains the Cache22 Esper system, which is a solution for two of those stages. Its purpose and why Esper was chosen is explained in the first two sections. The next three sections are about the design and implementation of the system, which went through two phases: one with and one without Esper. The final two sections show the results and the testing of the system. Chapter 7 explains how the system was generalized to make it applicable for other applications using a Domain Specific Language (DSL) to define the structure of the application in order to generate the adjusted code. The chapter will explain why the DSL version was made in Section 7.1 to follow with the design, Section 7.2 and the definition of the language, Section 7.3. What part of the code and why it is generated is explained in Section 7.4. The chapter ends with an evaluation of the system in Section 7.5.

Part IV contains the conclusion and future work. Chapter 8 is about the thoughts and partial implementation of a Sampling system to test robots who gather rates to keep the cache up to date. Chapter 9 contains a section on the conclusion and the future work.

The Appendix contains background information: code examples of both Esper and ExtJS, Esper language specifics and more details about the Cache22 Esper and the Cache22 EsperHA DSL systems. The bibliography, acronym list and glossary can be found at the end of the Appendix.
Chapter 2

Techniques and Frameworks

2.1 Complex Event Processing . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 ExtJS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

This chapter introduces the techniques used for this project. A general knowledge of the concepts behind the techniques is needed to understand the solutions presented in this thesis. Therefore, the content of this chapter provides background information that serves to familiarize the reader with the subject.

Section 2.1 explains Complex Event Processing (CEP), and how it is used by the Esper framework. Esper is used for the Cache22 Esper system—see Part II. The Sampling project, Chapter 8, uses ExtJS and is described in short in Section 2.2.

2.1 Complex Event Processing

Complex Event Processing (CEP) is a highly active research field and fulfills an important role in many application areas [3]. The purpose of CEP is to process large quantities of events into information that is useful and understandable. To be able to process the amount of events, a CEP system needs to be scalable and needs to be dynamic; dynamic in the sense that it can accept various event types. This is achieved by decoupling providers and receivers. Providers are different events sources, and the processed information is given to the receivers. They are decoupled, because a receiver only is interested in the processed information and not in how it is comprised. Vice versa, a provider does not know which kind of receivers exist and how the CEP system derives information from the events it provides [3, p. 241].

By specifying correlation rules the CEP system can detect relationships among events. At the moment relational events are detected they can be further processed: either aggregated or composed into new events. It is possible to feed events back into the CEP system; the CEP acts as its own provider—it is basically a loopback mechanism [3, p. 241].
Events from the provider have a strong relation to the semantics of the underlying technology, e.g. a temperature sensor, while the processed complex events sent to the receivers reflect the semantics of the application, e.g. today’s average temperature in Connecticut. Therefore, it allows independent reconfiguration at the technical and application level. Furthermore, the step-wise processing contributes to the scalability, because it reduces the amount of messages [3, p. 241].

In addition, CEP systems are expected to guarantee non-functional properties, such as, reliability, availability, performance, security, expressiveness of event languages, event derivation and usability [3, p. 241].

Figures 2.1 and 2.2 show a visual comparison of the way a CEP processes information in contrast to a traditional database system. The main difference is that information is considered as a stream of events—$S_1, S_2, \ldots, S_n$ in Figure 2.2—instead of rows that are (constantly) added to a table—$T_1, T_2, \ldots, T_n$ in Figure 2.1. These streams are denoted as the providers, and thus close to the technical level—in this example: temperatures from sensors all over Connecticut. Traditionally a query would be periodically executed to calculate the average temperatures from the tables—giving results $d_1, d_2, \ldots, d_n$ in Figure 2.1. In CEP, however, a query is composed to keep track of the average temperature and it outputs the result periodically. In the system, this query will be constantly running and only retains the information it needs. It gathers the information from the event streams that are passing through the system. After the period expires the query needs to output; it will send the result to any of the receivers that showed interest by subscribing to the query. This is a beneficial way of processing; none of the data that is not of interest is kept, because the CEP system will discard any data that it does not need for further processing. The data that is kept is only temporal, it will be removed from the system as soon it is delivered to all the subscribed receivers. Therefore, it reduces the amount of data and processing time, because the receivers get the exact information they need and do not have to perform lengthy queries on large tables containing extra information that they are not interested in.

![Figure 2.1: Measuring average temperature by using a database system.](image-url)
Esper

Esper is a semi-opensource CEP system made by EsperTech Inc.—Section 6.2 will explain why Esper was chosen. The standard version is open source—GNU General Public License (GPL) and their Enterprise and High-Availability editions are partly open source, but do require a license. The main features of the Enterprise edition are the input and output options and the graphical dashboard. The Esper High-Availability (EsperHA) edition provides a persistent state mechanism for the ability to recover from (system) crashes. Codehaus describes their product as follows:

Esper is a component for complex event processing (CEP), available for Java as Esper, and for .NET as NEsper. Esper and NEsper enable rapid development of applications that process large volumes of incoming messages or events. Esper and NEsper filter and analyze events in various ways, and respond to conditions of interest in real-time. (..) Esper offers a Domain Specific Language (DSL) for processing events. The Event Processing Language (EPL) is a declarative language for dealing with high frequency time-based event data. [8]

The NEsper version was not given any attention, this is a Java only project, but it is to be expected that it will only differ in some of its specifics and will be similar in its general workings.

The EPL mentioned above is similar to classical Structured Query Language (SQL) queries. The difference is that these queries are executed on streams instead of tables. Therefore, there are new aspects associated with the EPL—such as windows and patterns. Selecting, aggregating, filtering and
grouping function similar to SQL. In Esper the EPL is used to “express filtering, aggregation, and joins, possibly over sliding windows of multiple event streams. It also includes pattern semantics to express complex temporal causality among events (followed-by relationship)” [11].

Esper’s components adhere to the decoupled design of CEP as described in Section 2.1. Figure 2.3 shows the design and displays elements in two colors. The blue elements are part of the core Esper system. The core is indicated by the underlying blue rounded rectangle. The red elements are part of the Esper Enterprise Edition (EsperEE). At the bottom of Figure 2.3 a block called EsperHA is shown, this is the Esper High-Availability edition. This edition supplements the core with resilient statements—making them part of the persistent state, clustering and hot backup functionality.

The EsperEE has a number of features—only the GUI and Java Management Extensions (JMX) features were used for this thesis, namely [12]:

- Data Distribution Service, Esper Service layer, Data Push, Management of Logical and Physical Subscribers and Subscriptions
- Continuous Display: Composable, configurable and interactive displays of distributed real-time event streams; Charts, Gauges, Timelines, Grids
Techniques and Frameworks 2.1 Complex Event Processing

- **Rich Multi-Window GUI**: Rich web-based user interface for managing all aspects of multiple distributed engine instances
- **Distributed Cache support**: Interoperability with several commercial and open source distributed caching software
- **JMX**: Runtime management and monitoring over standard JMX connectors
- **JDBC support**: JDBC compliant client and server endpoints for interoperability

The GUI is not as rich as EsperTech advertises, but convenient for debugging and monitoring purposes. It gives a visual representation of what is going on in the CEP system. JMX is used to “remotely”—another process on the same machine in this case—and adds EPLs to the engine. These topics will be revisited when the system itself is explained in Part III.

The blue elements are part of the standard Esper edition. The decoupled design is evident with the Event Stream Connectors and Adapters at the left side of the component model and the Output Adapters on the right side. Inside the Esper Continuous Processing there are Event Processing Statements that perform reasoning and process the incoming events to produce new result events. These result events can be picked up by Subscribers—they must be subscribed to the statement in order to receive its result events, hence the name. A Named Window is a set of events that are (temporarily) kept. The attributes of the window define which events qualify to be kept and for how long. This can also be managed by other statements—e.g., the DELETE FROM WINDOW statement. Statements are active queries defined in the Event Query & Causality Pattern Language, otherwise known as EPL. The Esper Core Container consists of configuration and management functionality.

The Historical Data Access Layer is used to “replay” events, e.g., events gathered in a database which, at a later point, are loaded and sent to Esper. The Historical Data Access Layer is not used for this thesis, because there was limited need for it, and it is cumbersome to configure and run. The important element of Esper is the Event Stream Connectors and Adapters. If Esper is running embedded as part of a project the Esper’s API can be used to communicate. For example, to start, control, and send the events to Esper. In the case Esper is running as a standalone process input and output adapters are needed.

The adapters are Comma-Separated Values (CSV), HTTP, Socket, SpringJMS and the relational database adapter. The CSV adapter reads from a comma-separated values file, which is a simple text file where each line is a comma-separated list of values. Each line is transformed into an event and fed to Esper thus the CSV is performing a playback. The HTTP and SpringJMS use their respective protocols to send the event to Esper. The relational database adapter uses JDBC to connect and read from a database. Please consult Esper’s IO reference documentation [10] for more information on these adapters. The Socket adapter was used for this project, therefore it will be explained in more detail next.
2.1 Complex Event Processing Techniques and Frameworks

The Socket adapter is the adapter chosen to be used for this thesis—the reason why will be explained in Part III. The socket adapter makes the Esper core accessible by using Java Server sockets. The Server socket waits for a request to arrive from the network. When a connection established the Socket Adapter will start a threadpool that processes the incoming events. The events have to be serializable so they can be written to the ObjectOutputStream of the socket. The port number that the ServerSocket uses can be configured in Esper’s configuration. In essence, sending an event from a client process to the Esper engine is done in the following way by a developer: set up the adapter on the engine side, remember the configured port and connect to it with a socket on the client side. When the connection is established, the developer can send the events by writing them to the ObjectOutputStream. See Listings 28 and 27 in the Appendix for the code samples.

The output adapters can be seen as the inverse variant of their input adapters; instead of being able to receive events they will send resulting events. Consult the Esper IO reference manual [10] for more information on the kinds of output adapters and how they work. The Socket adapter does not have an output variant. For this project a hybrid between a subscriber and an output adapter was custom written—see Section 6.3.

Esper has two variations of subscribers; one is called listener and the other is called subscriber. The listener is not used in this project, because the listener variant produces additional event and processing information, which was irrelevant for the project. Producing this information causes overhead in comparison to the subscriber variant. The subscriber comes with the limitation that only one is allowed per statement, while multiple listeners can subscribe to a single statement.

No interfaces have to be implemented for a subscriber to receive events. The subscriber is a “direct binding of query results to a Java object. The object, a POJO, receives statement results via method invocation.” [9, p. 291]; which means that the Esper code will directly call the methods in your subscriber with the query results as arguments. In order to receive events a subscriber can ask for row-by-row delivery or multi-row delivery. Row-by-row delivery will deliver each result event by method invocations. With multi-row delivery all events are delivered together with one single method invocation. For both ways the subscriber class only needs to have an update method. The arguments of this method depend on the type of delivery and have to correspond with the result events [9, p. 291–292]. To clarify this, an example for a simple EPL is given:

```
select orderId, price, count(*) from OrderEvent
```

Listing 1: The row-by-row subscriber query example [9, p. 291–292].
Then in the subscriber the update method is as follows:

```java
public class RowByRowSubscriber {
...
    public void update(String orderId, double price, long count) {...}
...
}
```

Listing 2: The row-by-row subscriber implementation example [9, p. 291–292].

*Esper* offers a variety of ways to handle the delivery of events. The custom hybrid subscriber/output adapter uses the most generic way: the multi-row delivery as Map and Object array. The update method for `Object[][]` looks as follows:

```java
public class MultiRowObjectSubscriber {
...
    public void update(Object[][] insertStream, Object[][] removeStream) {
        ...
    }
...
}
```

Listing 3: The multi-row object subscriber implementation example.

The object array contains rows in its first dimension and contains the values in the order specified in the statement’s `select` clause in the second dimension [9, p. 294]. The Appendix contains more information about the other varieties of event delivery; these varieties are not applied, but could provide a better understanding of the delivery process. Event delivery using listeners is not covered in this thesis, but an explanation is provided in Esper’s reference documentation [9, p. 295–296].

This concludes the system explanation of Esper. The following sections will explain the concepts behind events, EPL patterns, and named windows.

**Events**

The two types of events: events and result events, are actually one and the same. The only reason why they carry different names is to identify in which stage in the processing the event reside. An event is a piece of (temporal) information that needs to be processed—a single reading from a temperature sensor for example. A result event is a processed event that is provided to the subscribers. The result event could be based on many other events and EPL’s determine how the resulting event is created. That the events and resulting events are one and the same becomes even clearer because the resulting event can be fed back into the CEP system, and thus act as a normal event. In the case of the temperature sensors; the average temperature per sensor can be determined by an EPL that filters outliers and tracks the average which it reports every hour. This reported average temperature can be fed back into the
2.1 Complex Event Processing Techniques and Frameworks

CEP system. Other statements do not have to know how to exclude outliers for each sensor if they want to measure the average temperature of a set of sensors; they simply use the result events of the single sensors as their normal input events. This way a CEP provides many levels of abstraction.

In Esper there are three default ways to represent an event. The three representations are the POJO event, Map event, and Extensible Markup Language (XML) event. The additional way, to build a plug-in using the event representation API, is not used for this project. Esper supports multiple event types, because existing applications may already have events in one of the representations. The event representations are interchangeable and interoperable [9, p. 3]; meaning that the statements stay the same when a representation changes and that a statement can use events of different representation.

The event representations have the following in common [9, p. 3–4]:

• All event representations support nested, indexed and mapped properties. There is no limitation to the nesting level.

• All event representations provide event type metadata. This includes type metadata for nested properties.

• All event representations allow transposing the event itself and parts of all of its property graph into new events. The term transposing refers to selecting the event itself or event properties that are themselves nestable property graphs, and then querying the event’s properties or nested property graphs in further statements.

• The Java object and Map representations allow supertypes.

Event properties are used to read the information from the event. Table 2.1 contains each property type with a description, its syntax and an example.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>A property that has a single value that may be retrieved.</td>
<td>name</td>
<td>sensorId</td>
</tr>
<tr>
<td>Indexed</td>
<td>An indexed property stores an ordered collection of objects (all of the same type) that can be individually accessed by an integer-valued, non-negative index (or subscript).</td>
<td>name[index]</td>
<td>sensorId[0]</td>
</tr>
<tr>
<td>Mapped</td>
<td>A mapped property stores a keyed collection of objects (all of the same type).</td>
<td>sensor('light')</td>
<td>name('key')</td>
</tr>
<tr>
<td>Nested</td>
<td>A nested property is a property that lives within another property of an event.</td>
<td>name.nestedname</td>
<td>sensor.value</td>
</tr>
</tbody>
</table>

Table 2.1: Types of Event Properties.
The XML representation is a com.w3c.dom.Node instance. Strings will be the return types of properties if no schema is provided. If a schema document is configured, Esper can present event type metadata and validate statements that use the event. Although XML is a useful and convenient way of representing events it was considered, but not used for this project. Please consult the reference documentation [9, p. 14–21] for more information.

The java.util.Map representation was used during the first phase of this project. To use this representation an event has to implement the java.util.Map interface. The Java map works with key value pairs. The property names are the keys and the value of the property is set as the value in the map bound to that key; normally the Java method get(<key>) would be used to retrieve the value. In the Map representation multiple supertypes can be used, but they also have to be a Map event type. All the properties of any of the supertypes will then be available in the type itself. One-to-many relationships are represented via arrays, these kind of properties within the Map may be an array of a primitives, an array of a Java object or an array of a Map. All subtypes will be matched by an EPL when its supertype is used. For example, when there are two subevents that both have Chair as supertype, such as LoungeChair and RockingChair events, the EPL looking for Chair events will match the LoungeChair and RockingChair events. The reference documentation [8, p. 11] states the following about the event properties:

Map event properties can be of any type. Map event properties that are Java application objects or that are of type java.util.Map (or arrays thereof) offer additional power:

- Properties that are Java application objects can be queried via the nested, indexed, mapped and dynamic property syntax as outlined earlier.
- Properties that are of type Map allow Maps to be nested arbitrarily deep and thus can be used to represent complex domain information. The nested, indexed, mapped and dynamic property syntax can be used to query Maps within Maps and arrays of Maps within Maps.

```java
// The Java event values
String cardNo = "233";
Account acc = Accounts.retrieve();

// The Esper Event creation
Map event = new HashMap();
event.put("cardNo", cardNo);
event.put("account", acc);

// Sending the event
epRuntime.sendEvent(event, "CardUsedEvent");
```

Listing 4: Map event code example.
Listing 4 shows the creation of an event that has a Java primitive and a Java object as properties. These properties can be used in an EPL such as the one in Listing 5 for example. The code uses two Java objects and creates a Map event by instantiating a HashMap. The properties are set by binding the Java objects to the property names using the put method of the HashMap. The event now has two properties: `cardNo` and `account`. The name for the event is given at the moment it is sent, as can be seen in the last line. Now the EPL can query on the CardUsedEvent. The EPL in the example queries for the account number and the card number within a window of 60 seconds. The Esper reference documentation [9] provides more information on how to use the supertypes and the more advanced event types.

```
select account.number, cardNo from CardUsedEvent.win:time(60 sec)
```

Listing 5: Map event EPL example.

The POJO event representation is used in the second and third phase of the project. The Plain-old java objects are instances of classes which expose their properties with getters. These getters have to adhere to the Java Bean specification—note that the classes do not need to be fully compliant JavaBeans. In essence events should be immutable, because they are a recording of past events—even though the engine will still accept mutable events [9, p. 7]. POJO’s are the preferred way for using classes as events, but by using the configuration a lot of different class types can still be accepted—e.g., legacy classes. The project uses POJO’s, but when legacy classes are needed please consult the Esper Reference [9, p. 332] on how to configure these classes.

Supertypes are supported in the same way as with the Map representation. The EPL can again use the superclass or interface to catch all the inheriting child classes, and in the code of the child class this is simply done by extending a superclass or implementing an interface. Properties within a POJO can have all the types from Table 2.1: simple, indexed, mapped, and nested. The four property types would reflect in a class as follows:

```java
// The POJO event class
public class NewEmployeeEvent {
    // simple type
    public String getFirstName();
    // indexed and nested type
    public Employee getSubordinate(int index);
    // mapped type
    public Address getAddress(String type);
    // indexed type returning iterable
    public Employee[] getAllSubordinates();
}
```

Listing 6: POJO event code example.
Techniques and Frameworks 2.1 Complex Event Processing

Techniques and Frameworks 2.1 Complex Event Processing

Listing 7: POJO event EPL example.

The different types can be used in an EPL as shown in Listing 7. In the EPL the POJO's simple property getFirstName corresponds to firstName, and it is simple because it returns a primitive. The indexed property getSubordinate(int index) is used in the EPL as subordinate[0] and subordinate[1], hence the name indexed, because it returns a different object based on the given index. The comments of the event source code state that getSubordinate(int index) is nested as well. It is nested because it returns an object that again is a POJO and has its own properties. In this case, a POJO representing an Employee with a property called name in this case. The alternative way to have an indexed property is using a method that returns an object that implements Iterable. In this case, the method name is written in the plural form, and does not accept any arguments. The EPL remains the same, it has still the method name with the index number in square brackets. There are some specific ways that the capitalization in method names translate to properties in the EPL [9, p. 8], but experience suggests that the best practice is to use descriptive full names using the CamelCase convention. It is also possible to use constants, enumerations, parameterized types, and even set properties from EPL. These work intuitively, for setting properties a setter is needed and the others just use a slightly different notation, see Esper's reference [9, p. 9–10] for more details.

EPL

CEP revolves around the decoupling of providers and subscribers. For a system to be dynamic it has to accept all kinds of events and be able to produce result events, therefore the way these events are transformed has to be highly configurable and dynamic as well. Esper uses statements that are basically the correlation rules mentioned in the first paragraph, which determine how events are transformed. More specifically—for Esper—statements are running queries written in what Esper calls: Event Processing Language (EPL).

EPL is similar to SQL and therefore has a couple of recognizable clauses: select, from, where, group by, having and order by. These clauses are used to derive, filter and aggregate results from event streams. Event streams replace tables, where the event itself replaces a row in the table. As with SQL event streams can be merged with joins. The other clauses insert into, delete, and update are also present in EPL. The insert into can be seen as forwarding the events into other streams—remember the loopback mechanism. The delete and update have the same effect as the SQL variant, but there is no table to update or delete a row from. They can, however, update or delete an event from a named window—which will be explained in more detail later on. Metaphorically speaking, events can be seen as cars and
event streams as traffic. The traffic passes through a tunnel, which a metaphor for the window. With views it is possible to imply limitations on the tunnel. The concept of a view is similar to that of a view in SQL. A view defines the data available for the query. For example the `win:length` view would imply that the tunnel only allows a given number of cars in the tunnel, whereas the `win:time` would require the cars to cross the tunnel within a given time. There are many different built-in views [9, p. 266–281], and even views that derive statistics [9, p. 281–287].

In addition to SQL, EPL has two more clauses and a new concept; the clauses are `pattern` and `output`, and the concept is a named window. The `output` clause is used to stabilize and control the output of result events, for example let the statement output every 60 seconds—see the reference for more information [9, p. 82–87]. The named window and patterns concepts will be explained in further detail next.

**Windows and Named Windows**

Windows are created directly by using the `from` with a `win` view—see Listing 8 where a window is created for the Withdrawal event. The `win` views are the subset of view types that keep a (sliding) window of events. The given expression for the window determines when to release events from the window. Views can be combined arbitrarily, so only the most commonly used views will be explained here. Other view combinations can be found in the Esper Reference [9, p. 266–287]. The commonly used variations are: `length`, `length_batch`, `time`, and `time_batch`.

```
select * from Withdrawal.win:time(4 sec)
```

Listing 8: Event window using view.

The length view versions keep events independent of time: they keep the events based on amount instead. Therefore the window is called length, because the length of the window is the maximum amount of events it can hold. The time windows are not based on length, but on time, so they keep events for a certain time. The window can thus vary greatly in size. Consider a window of 60 seconds and a stream with 1000 events per hour. These 1000 events all arrive at the same time with a frequency of once every hour. In this case the window of 60 seconds will have zero events for 59 minutes and for one minute it will contain 1000 events. The batch versions of both length and time have a similar effect. The batch will act as a tumbling window instead of a sliding window. The `length_batch`, for example, will add events until the number of events that fit in the window is reached. When this happens it outputs all as new events, after which it empties its window, so none of the events are retained after the output. The normal length window, on the other hand, will output the newly arriving and the removed event each time a new event arrives. These newly arriving and removed events are called the `istream` and `rstream` respectively. A `select` query will by default use the `istream`. 
Figure 2.4 shows a length window based on the query from Listing 9. The UpdateListener receives both the istream and the rstream as displayed on the right. The incoming events and the state of the window at that time are shown on the left of Figure 2.4. Time is progressing from top to bottom, with the top being the earliest point in time and the bottom the latest point in time. Due to the irregularity at which the events arrive the updated window is also drawn irregularly from top to bottom, this is because the window is not time dependent. There is still space in the window till the point at which event \( W_5(150) \) arrives, so this causes the listener to only receive each new event until that point. This change when event \( W_6(300) \) arrives, the window has reached its maximum capacity and thus removes \( W_1(500) \). This results in a new and an old event in the UpdateListener; \( W_6(300) \) istream and \( W_1(500) \) rstream. This continues when a new event \( W_7(125) \) would appear; \( W_7(125) \) istream and \( W_2(100) \) rstream.

\[
\text{select } * \text{ from Withdrawal.win:length(5)}
\]

Listing 9: Event window using length.

![Figure 2.4: Visual representation of a win:length(5) view.](image)
Figure 2.5 shows a length batch window based on the query from Listing 10. The behavior is similar to that of the length without batch window. Only the way the istream and rstream are populated is different, due to what happens when the window reaches its maximum length. With the batch it waits with outputting on the istream until the window is full and then resets it. When the window is full again it will output to the windows content to the istream again and the rstream will contain the content of the previous window.

```
select * from Withdrawal.win:length_batch(5)
```

Listing 10: Event window using length batch view.

![Figure 2.5: Visual representation of a win:length_batch(5) view.](image)

Figure 2.6 shows a time window based on the query from Listing 11. The layout of the picture is similar to that of the length window, but here the time windows are drawn vertically, because they are not independent of time. The windows are drawn next to each other to simulate the sliding effect. The statement and thus the window is activated at t and because it size is 4 seconds stretch till t + 4. At t + 4 the first event arrives and is put in front of the time window, which causes it to be outputted as part of the istream. Next, at t + 5, the situation is reflected by the window horizontally drawn next to the first
window; another event $W_2$ has arrived and has been outputted. This continues until $t+7$, here no event arrives, so no events are on the istream or rstream. At $t+8$ it has been 4 seconds after $W_1$ arrived, therefore $W_1$ leaves the windows and is outputted on the rstream.

```
select * from Withdrawal.win:time(4 sec)
```

Listing 11: Event window using time view [9, p. 30].

Figure 2.6: Visual representation of a \texttt{win:time(4 sec)} view [9, p. 30].

Figure 2.7 shows a time window based on the query from Listing 11. This figure has a similar structure of that of Figure 2.6. The mayor difference is that this is a batch, thus a tumbling window and not a sliding window. This shows from the fact that the windows are drawn next to each other, therefore spanning $t+4$ horizontally and vertically. The istream is populated every 4 seconds and contains all the events of the previous 4 seconds, so for $t$ to $t+4$ these are $W_1$ and $W_2$, the next 4 seconds only one event arrives so for window $t+4$ to $t+8$ the istream is $W_3$, but now there is also an rstream. The rstream is the previous window, thus $W_1$ and $W_2$. 

---

"Techniques and Frameworks" 2.1 Complex Event Processing

"Select * from Withdrawal.win:time(4 sec)"

Listing 11: Event window using time view [9 p. 30].

Figure 2.6: Visual representation of a \texttt{win:time(4 sec)} view [9 p. 30].

Figure 2.7 shows a time window based on the query from Listing 11. This figure has a similar structure of that of Figure 2.6. The mayor difference is that this is a batch, thus a tumbling window and not a sliding window. This shows from the fact that the windows are drawn next to each other, therefore spanning $t+4$ horizontally and vertically. The istream is populated every 4 seconds and contains all the events of the previous 4 seconds, so for $t$ to $t+4$ these are $W_1$ and $W_2$, the next 4 seconds only one event arrives so for window $t+4$ to $t+8$ the istream is $W_3$, but now there is also an rstream. The rstream is the previous window, thus $W_1$ and $W_2$.
select * from Withdrawal.win:time_batch(4 sec)

Listing 12: Event window using time batch view [9, p. 31].

Figure 2.7: Visual representation of a win:time_batch(4 sec) view [9, p. 31].

Named windows can be created using the create window clause. This will create a global data window holding events of the same type and can be used in multiple statements. The insert into and delete clauses control the events that enter and leave the window. Another way for events to leave the window is, of course, the expiry policy of the view. The benefit of a named window is that it can be used by multiple statements that all see the same window. Normally, when using a view of an event type, a window is generated per statement and dependent upon the events from providers. The named window can combine events, subset event properties, and also declare its own properties while being populated from other statements [9, p. 52].

Listing 13 shows a query that creates a window that has two columns or event properties; one is the OrderEvent and the second is a calculated derived price. The from OrderEvent is needed, because the window is based upon the OrderEvent and selects all its properties, hence the asterisk. In the second
query the named window is populated. It selects every OrderEvent and inserts them into the named window, and the second column is calculated.

```sql
-- Create the window
cREATE WINDOW OrdersWindow.win:time(30) AS SELECT *, price AS derivedPrice
FROM OrderEvent

-- Inserting events into the window
INSERT INTO OrdersWindow
SELECT *
FROM OrderEvent

SELECT MyFunc.func(price, percent) AS derivedPrice
FROM OrderEvent
```

Listing 13: Creating and inserting into a named window

For additional information on named windows please consult the Esper Reference documentation [9, p. 114–131].

Patterns

Patterns are a new concept which does not exist in SQL and they are not just a few new clauses. They can match events or multiple events using a pattern definition that is flexible and expressive. It allows for more complex combinations in comparison to what is possible with the non-pattern clauses. A definition composed out of expressions. Expressions can be nested arbitrarily deep using round parentheses.

A pattern expression is composed of pattern atoms and pattern operators [9, p. 150]:

1. Pattern atoms are the basic building blocks of patterns. Atoms are filter expressions, observers for time-based events and plug-in custom observers that observe external events not under the control of the engine.
2. Pattern operators control expression lifecycle and combine atoms logically or temporally.

The atoms can be filter expressions that specify an event to look for or time-based event observers, such as intervals and schedules, or even a custom observer [9 p. 150].

A variety of operators is available in Esper. They can be grouped into repetition controlling, logical, temporal and guarding operators. The logical operators are straightforward: and, or, and not. The temporal group operates on event order; the -> (followed-by) is such an operator and matches if a matching expression is followed by another expression that matches.

There are a number of guard expressions. All the guards are meant to limit the lifetime of a (sub)expression. The clauses are: timer:within, timer:withinmax, and while. Guards can also exist at atom level, here they act as time-based observers. These atoms are: timer:interval and timer:at.

The nesting of pattern expressions has an impact on the repetition controlling operators: every, every-distinct, [num] and until. The repetition controlling expression are impacted, because their semantics change with the
use of parentheses. The operator precedence can be found in the Appendix. Listing 14 shows one pattern query using a lot of operators and atoms. The Esper reference documentation [9, p. 189–198] has a detailed explanation of the operators and atoms used in the statements of this project, but most operators work similar like they do in programming languages and are self-explanatory. For the complete pattern syntax see the Esper Reference [9, p. 150–174].

```
every (not A -> [2](B and C)) until timer:interval(60 sec)
where timer:within(15 min)
```

Listing 14: A pattern using many different atoms and operators

The `every` operator will try to match every sequence of events that match the subexpression, while the `where timer:within(15 min)` limits the lifetime of the whole expression to 15 minutes, so after 15 minutes this statement will cease its operation. The whole expression within the parentheses has to match before the `every` operator restarts looking for the first `not A` event. Other possibilities for using `every` without parentheses and their semantic meaning can be found in the Appendix. When a `not A` event is found the `->` (followed-by) operator has to fulfill its second part, so it start looking for a `(B and C)` that is repeated twice, hence the `[2]`. The `until` possesses an additional requirement on the matching of two `(B and C)`’s. It requires that this happens within 60 seconds. The part after the `until` is called the end expression, it ends the repeat (]). In this case the end expression is an atom: the time-based `timer:interval(60 sec)` atom, which observes time and matches when it seen 60 seconds pass-by. The one sentence description of what this statement tries to do is: for the next 15 minutes it collects every repetition of two B and C events that occur within 60 seconds after an event passed that was not an A event.

2.2 ExtJS

ExtJS is a Javascript framework from Sencha that is used as part of the Sampling implementation. ExtJS is used to create the client side GUI. This GUI helps to modify the Sampling system’s settings and helps to perform the Sampling actions. ExtJS is a framework that is helpful to anyone who is not a web developer. Experienced web developers will find it hard to work with ExtJS because it does not adhere to any of the traditional concepts of web development. Plain html, css and javascript can no longer be used in the traditional manner within the ExtJS framework. The documentation is inconsistent which makes it hard to get the samples and your own code working. On top of that, debugging is tedious. There is no syntax checking, because it is all Javascript, but the framework does not give you warnings or errors, for example when you are missing values or bracket in their syntax. ExtJS simply executes the code, which will result in a non-descriptive Javascript error buried within the framework’s code.
The benefit of ExtJS is that it supports multiple browsers and has a lot of standard UI components. The framework has a clean component model and is based upon the Model-View-Controller (MVC) model. In their MVC, the models are the collection of fields and their data. The datapackage provides several ways to persist the data, which is called a Store. The View consists of components; like grids and panels. The Controllers contain the actual functional code for the application and has tasks like rendering views, instantiating models, and other application logic.

The next paragraphs will explain, in short, how to use ExtJS to create an application using the Sampling GUI as an example. To keep it short, only the core methods will be discussed here. See the Appendix for the full sample code for a more comprehensive understanding. The numbers in the captions of the Listings refer to the Listing within the Appendix containing the full code.

The application starts with one file that contains the instantiation of the Application class and it contains global settings, reference to models, views and controllers. The application class is instantiated as follows:

```javascript
Ext.application({
    name: 'AM',
    appFolder: 'app',
    ...
});
```

Listing 15: Application definition. See Listing 34.

Controllers contain the logic of the application and listen to events from the views. To use the controller it has to be added into the controllers section within the Application file, this will make sure that the controller is loaded. Each controller has its own file, like `app/controller/Robots.js` and contains:

```javascript
Ext.define('AM.controller.Robots', {
    extend: 'Ext.app.Controller',
    ...
});
```

Listing 16: Controller definition. See Listing 35.

To actually create pages for the GUI the V part of MVC comes to play. The V stands for view. A view is a reusable component that can exist of multiple components. When modifying the robots for the Cache22 Sampling project there are a couple of action possible; for example adding a robot, showing all robots, etc. When working with ExtJS in a structured way, each of these actions are controlled by the controller and for each action there is a different view. The view is added into the view section of the controller, so it can be rendered by the controller. A view is contained in its own file (app/view/robots/List.js) and is initiated as shown in Listing 17.

```javascript
Ext.define('AM.view.robots.List', {
    extend: 'Ext.container.Container',
    ...
});
```

Listing 17: View definition. See Listing 36.
2.2 ExtJS

Techniques and Frameworks

Listing 17: View definition. See Listing 36

This view can now be used within controllers, therefore it has to be added to the views section within the controller—see Listing 35. One way this can be done is by using xtype and the name of the widget—a view can be given an alias, and the view is called a widget when referring to the alias.

The data package must be used to load data into the grid that was created by the view. The data package contains several ways to handle data, but the simple Ext.data.Store suffices. A store is used to create, load, update and delete data on the client side. Depending on the model used within the store, the data can be synced with the server. The store can be as simple as a definition of fields and an array of data, but in general a model is used to specify the structure of the data used in the store. The definition of a store looks as follow:

Listing 18: Store definition. See Listing 37

The store also has to be known to the controller and therefore it has to be added to the store section within the controller. Hereafter, the controller is able to use the store to perform the actions defined in its logic on the data. For example, when the remove button is pressed in the view—which the controller renders at that moment—the controller listens to the clicked event and the event listener method deletes the selected item from the store. The store is linked to the grid, and therefore also deletes the row from the gridview. Defining the fields in the store is not the best practice when the structure of the data becomes more complex or when the structure is across multiple stores. In this case, a model can be used that can capture this structure and is able to reuse it across multiple stores. A model is defined in Listing 19.

To actually save the data to a server a Proxy can be used. The Proxy definition can be done in either the store or the model, depending how the data has to be processed. For the Cache22 Sampling project the proxies were defined in the model—see Listing 38 in the Appendix.
Ext.define('AM.model.Robot', {
    extend: 'Ext.data.Model',
    fields: ['robotId', 'identifier', 'description'],
    ...
CHAPTER 3

THE COMPANY

3.1 Introduction ......................................................... 32
3.2 Brands ................................................................. 32
3.3 Affiliates ................................................................. 32
3.4 Rate Types and Models .............................................. 33
3.5 Services ................................................................. 34

The Priceline Group is composed four primary brands. Priceline.com, Booking.com, Agoda and Rentalcars.com offer various travel services. The goal of this chapter is to provide the reader with background knowledge about the services and the terminology related to the Priceline Group. Although the information can be considered as background information, it will contribute to the understanding of the overall project and provides the reader with a better feel for the size of the project.

Information about the Priceline Group and different brands can be found in Section 3.1 and Section 3.2. In addition to the traffic generated on Priceline.com’s own websites, other travel sites affiliated with Priceline.com (Affiliates) also steer traffic or direct bookings to Priceline.com—see Section 3.3. In Section 3.4 the models and their corresponding rate types are explained. Followed by Section 3.5, which explains the travel services that are provided by the brands and which business model is used per brand or service. The information in the chapter is based upon the PCLN 2011 Annual Report [14].
3.1 Introduction

Priceline.com was founded in 1998 in the United States. Over the years Priceline.com expanded their business with Booking.com, Agoda and rentalcars.com. Together they are the four primary brands of the Priceline Group. Priceline.com is an online travel company, offering their customers hotel reservations—at 210,000 hotels worldwide, car rental reservations, airline tickets, vacation packages, destination services and cruises. They provide their services in Europe, North America, South America, the Asia-Pacific region, the Middle East and Africa.

The common stock is listed on the NASDAQ Global Select Market under the symbol “PCLN”. Although Priceline.com is a relatively young company, it has a higher market value than more established companies such as Dell and Yahoo. In 2011, the Priceline Group had a gross profit of approximately 3.1 billion.

3.2 Brands

The Priceline Group, as of January 31, 2012, employed approximately 5000 full-time employees. Approximately 1,000 employees are based in the United States and approximately 4,000 are based in Priceline.com’s international offices.

The Priceline Group contains four primary companies: Priceline.com, Booking.com, Agoda and rentalcars.com. Priceline.com is based in the United States and offers a variety of travel services through www.priceline.com. Booking.com is based in Amsterdam with local offices all over the world. Their worldwide online hotel reservation service at www.booking.com covers hotels ranging from small independent hotels to five star properties all over the world. Agoda is an online hotel reservation service with Asia as its geographical focus. Agoda’s reach in the Asia-Pacific region is unmatched with 30,000 hotels instantly available. Rentalcars.com books over 2 million rentals a year in more than 6,000 location across the world. All brands are part of the Priceline Group and therefore are able to offer the lowest prices at more than 210,000 hotels all across the world.

3.3 Affiliates

An affiliate is a company that uses Priceline.com’s infrastructure. Affiliates act as third-party distribution channels, selling Priceline.com travel products either as a service on a non-travel website, or through an independently branded travel website, or by simply accessing the inventory through Priceline.com’s XML solutions. An affiliate will receive a commission per booking.

There are additional sources that generate traffic for Priceline.com’s websites. One of the major sources is Google. For example when a search is made on Google Maps™, and the keyword hotel near New York is used, Google Maps™ will display the hotels near New York and their prices—see Figure 3.1. The
customer selects one of the companies and, in the case of the Priceline Group sites, the customer will be redirected to a single-page checkout, where the type of room can be selected to completed the reservation.

3.4 Rate Types and Models

The Priceline Group uses different models within its companies. There are three main models that also determine the rate type that is used. A rate is the price of an item, e.g. airline ticket.

**Opaque Rates**  In the opaque model not all the aspects of the travel service are visible to the consumer before making an offer, hence opaque. This model is used with the Name Your Own Price® service. With this model Priceline.com is the merchant of record, meaning that Priceline.com is the company that charges the customers credit card.

**Merchant Rates**  In the merchant model the rate is price-disclosed. The Priceline Group brands are still the merchant of record, thus the ones that charge the customers credit card. This model is, for example, used in the U.S. and in Asia. The rentalcars.com brand uses a variant of the merchant model, merchant semi-opaque, where only the brand of supplier is non-disclosed making it semi-opaque.
Agency Rates The agency model uses retail prices. The Priceline Group companies are not the merchant of record and where the prices of their services are determined by third parties. The agency model earns money from travel commissions, reservation booking fees, and customer processing fees.

3.5 Services

The Priceline Group owns and operates an online global travel service network that attracts consumers wishing to make travel reservation and connects them in an efficient and innovative manner with suppliers of high quality travel services around the world [14]. People visit the Priceline sites to search for hotels, airline tickets, rental cars, etc, because a lot of the industry's travel brands are represented. Therefore, it is an easy way to compare and book, while saving money. For small travel businesses it is a huge benefit to be able to reach such a vast amount of customers, and even for large chains the visibility and additional potential customers is reason enough for being represented by the sites. The Priceline Group offers a variety of services for the United States and international market.

Hotels

Over 210,000 hotels in over 160 countries are offered on the Priceline Group websites. The websites are served in 41 different languages. In the year 2011 a total of 141.6 million hotel room nights, which is over 385,000 room nights a day, were booked through the sites. There are two ways of booking a hotel: by using retail or Name Your Own Price®.

Retail Hotels Retail means that the price of the hotel room is disclosed. Using the retail service, customers can select the exact hotel they want to book and the price of the reservation is disclosed prior to the time of booking. Retail hotels are offered internationally; primarily through the Booking.com and Agoda brands. Booking.com operates under an agency model and Agoda primarily under a merchant model. In the United States reservations are made using a merchant model, and as well sourced through Booking.com.

Name Your Own Price® Hotels With Name Your Own Price® customers can bid. The customer selects the city and a neighborhood, the check-in and check-out dates and the class of service—e.g., 4 stars. Together with a valid credit card and a price, the selection is called an offer. For making an offer the customer must agree to stay at any of the participating hotel partners and accept that the reservation cannot be refunded or changed. The exact hotel is shown after the offer is accepted by Priceline. If the offer is not accepted the customer can modify its selection or wait 24 hours and make the same offer again. Most hotels in all major cities and metropolitan areas in the United States and Europe participate. A hotel participates by providing Priceline.com with discounted rates. These rates are private, and thus not accessible to the general
public. Name Your Own Price® can save up to 60% of retail rates in the same area and class of service.

**Rental Cars**

The Priceline Group websites booked a total 23.8 million rental car days for the year 2011. That are over 65,000 rental car days a day. Rental car services are offered world wide and at most major airports customers can book using Name Your Own Price®.

**Retail Rental Cars** The international rental car market is served by rental-cars.com offering rental cars in more than 4,000 locations in the world, with customer support in 38 languages. The model used is semi-opaque merchant model; the customer can see the vehicle type, price and rental location, but the supplier will be unknown until the reservation is made.

In the United States retail service is offered on www.priceline.com and other websites. The service is price-disclosed and operates under the agency model. In addition to the international retail, the customer also knows the exact brand.

**Name Your Own Price® Rental Cars** It is possible to use Name Your Own Price® when renting a car at most major United States airports. The process is similar to the hotel variant; the customer selects where and when they want to rent a car, the kind of car (e.g., compact, mid-size, etc.) and the price per day. When the offer is accepted the car is reserved and the customers credit card is charged.

**Airline**

Priceline.com sells airline tickets to customers in the United States. In 2011 a total of 6.2 million airline tickets were sold. That is over 16,500 tickets a day.

**Retail Airline Tickets**

Retail airline tickets can be bought at disclosed price with disclosed itineraries. Prices are determined by the airlines and the airline is the merchant of record. These tickets do not have the restrictions that apply to the Name Your Own Price® airline tickets.

**Name Your Own Price® Airline Tickets**

For airline tickets the service operates in a similar matter as that of the Name Your Own Price® for hotels. The customer selects the origin and destination, the departure and return dates, and the price the customer is willing to pay. Together with a valid credit card the customer can make the offer. When making an offer the customer agrees to similar conditions like those on hotels—see the Priceline.com site for the conditions.
3.5 Services

Vacation Packages

Vacation package contain all three services: a hotel, airline, and an optional rental car. The customer can select the exact hotel, and choose to use a retail or opaque airline ticket. The package itself is sold at a disclosed price, but for the individual components the exact price is unknown.

Destination Services

United States customers can purchase destination services in addition to their airline ticket, hotel, rental car or vacation, but they are also offered as a standalone service. Destination services are things like parking, event tickets, ground transfers, tours, etc.

Cruises

Priceline.com offers cruises, on a commission basis, through World Travel Holding, which is an agent representing major cruise lines. Customers can compare and book cruises, the price is disclosed.

Travel Insurance

For their air, hotel and vacation packages Priceline.com offers customers an optional travel insurance. They retain a fee per package purchased from BerkeleyCare.
**CHAPTER 4**

**Processes and Systems**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Overview</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Processes</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Systems</td>
<td>42</td>
</tr>
</tbody>
</table>

Priceline.com’s business model requires them to gather rates, for the services they provide, at the moment their customers request them. This is different from the other brands in the Priceline Group, because they are actively provided with the new rates. This makes Priceline.com the target company for the project. The project focuses on Priceline.com’s hotel travel service. Therefore, this chapter explains the processes in Section 4.2 and the systems in Section 4.3 that are related to the hotel travel service. In the rest of the thesis it is assumed that the information and the terminology used in this chapter is known to the reader.

### 4.1 Overview

There are many components involved when searching for and booking hotels. An overview of the architecture is shown in Figure 4.1. Each component has a specific name within Priceline.com, but in this document a general and more descriptive name is used. Although the architecture shown is a representation that has quite some detail it still is abstracted to a level sufficient and relevant for the project. The general flow of requests runs from the top of the diagram to the bottom and the processes are explained in the next section. The yellow ellipses at the top, and the Google one in the lower right corner are places where the requests will originate from. The yellow ellipse in the lower left corner are the external systems where the rates are gathered from. The blue ellipses are databases; the InvRef will be referred to as the *Inventory Cache*, the InvS are the *Static/Lookup databases*. The blue rectangles are Priceline.com’s systems. All these systems will be explained in Section 4.3.
4.2 Processes

There are four major processes that needed to be taken in consideration in the design of this project. Two of these processes involve the search and booking of hotels, while the last process is an automated process where robots are mimicking searches. The affiliate booking process is similar to that of the customer booking; the process for the special affiliates is explained using Google as an example.

Customer booking

Hotel searching and booking are two different processes; specifically in how they impact the Inventory Cache. To make a search, the customer visits Priceline.com with its browser. The website is shown by a Webtier. At the site, a customer can search for hotels by specifying the city, checkin and checkout date,
and the number of rooms. This selection is called an itinerary within Priceline.com; although the word itinerary means a detailed plan for a journey, it is used to denote a specific combinations of a city, checkin and checkout, etc.

The search request is then given to the Gateway. The Gateway consults the Static/Lookup Databases to gather information about the hotels—e.g., to find the ones close to the given city. With this information the Gateway checks whether the Inventory Cache has an active rate for the hotels that are part of the search. An active rate is a rate that has been collected a while ago and is expected not to have changed in the meanwhile. How long ago a rate is collected and still is considered active is determined by rudimentary business logic based upon the specific combinations of the itinerary properties. The itineraries that have inactive rates are passed on to the Search Engine. The Search Engine then connects to several systems; like Pegasus, Worldspan, and also the systems of Priceline.com sister companies: Booking.com and Agoda. These systems are collectively called Inventory Sources. These systems are external and Priceline.com can query them, but does not exert control over them. The rates found are returned to the Search Engine. The Search Engine adds the fresh rates to the Inventory Cache.

The combined result is returned to the Gateway. This result does not necessarily have to contain rates for all hotels requested, for the simple reason that some hotels may not have a rate published for that itinerary or do not have any rooms available. The search engine also sends the rates it gathered to the Cache22 system. These rates can either be Global Distributed System (GDS) rates or cache rates. A cache rate is the rate served from the Inventory Cache, and the GDS rate is served from an Inventory Source. Then the Gateway consults the Static/Lookup Databases again, but this time for what is called the Ranking Logic. The Ranking Logic determines the way the hotels are ordered. The ordered result is then passed on to the Webtier that applies its presentation logic and sends the result back to the customer.

The customers now have a list of hotels on their screen. To continue, he or she selects a hotel of their liking, and are presented with a room type page. Here they select the room type they want to book. The request is sent to the Webtier, when the selection is made. Because this is a single property call, the Gateway does not have to perform any actions and simply forwards the request to the Search Engine. The Search Engine knows the request is a booking and therefore is forbidden to serve the rate from the Inventory Cache. The rate used for the booking always has to be the one that is present in the applicable GDS. If both rates are still the same the booking is processed. If the rate quoted differs from the GDS rate, the customer is notified of the price change and can chose to book for the new price or select another hotel.

Searching has a different impact on the Inventory Cache than booking. Searches for rates are mostly served from the Inventory Cache. In case a search generates a GDS call, the rate retrieved through GDS will simultaneously update the Inventory Cache. When booking, only the GDS is used and the Inventory Cache is circumvented. Therefore, every booking does update the Inventory Cache.
4.2 Processes

Affiliate booking

Searches and bookings from affiliates work in a similar way as customer searches and bookings. The main difference is that affiliates do not make use of the website, but get their information through other sources. The XML interface is used by most affiliates. The requests from affiliates are annotated with a tag identifying the affiliate making the search or booking. This tag is convenient for filtering measurements.

However, for a few affiliates the process works differently. Of those, Google is one that plays a significant role in the way the architecture works. Google does not work on a request basis, so they are unable to use the XML interface. Google wants to have the rates stored on their servers, mainly because of the vast amount of user traffic. To be able to push the rates to Google at a regular interval the rates have to be stored for a longer period than currently happens in the Inventory Cache, otherwise there would be too many missing rates. The Cache22 system does this work, in addition to keeping track of all the cache and GDS rates. The Cache22 system keeps a grid with a rate for every itinerary. New rates are updated within this grid and changes are tracked. The changes are pushed to Google in batches. Google shows the rate in their Google Maps™ product—see Figure 3.1. When a customer clicks on the rate within Google Maps™ it goes to the Priceline.com One Page Checkout. The customer can book with one click, and the rate has to be the exact booking rate. Therefore, these pages will trigger a request for the GDS rate. These requests get annotated with the affiliate tag, being Google in this case. The Google tags will provide feedback on how accurate the rates pushed to Google were and whether they need to be pushed more or less often.

Mobile search

The grid managed by the Cache22 system also proved to be useful for high demand applications. Normally when a search is done, the request is limited to one city and due to the presentation logic, which applies pagination for example, not all hotels and their rates have to be gathered at once. In terms of a single search it is not as high as the demand that is needed for the mobile application, which uses a lot of searches simultaneous, of course the vast number of customers doing searches and bookings all the time makes normal system a high demand process too.

For Priceline.com, the mobile platforms are all about user experience, where a focus lies on a intuitive way of navigating while maintaining a fast and spiffy performance. Therefore they chose to use a map as the way to navigate and find the hotel of your liking. Each hotel with rooms available gets a pinpoint on the map with the rate for the selected dates. The screenshot—Figure 4.2—of the iPad app gives an idea of how many hotel rates are shown at once on the map.

A search on the website takes a significant amount of time—remember that it has to gather rates from the Inventory Sources, therefore this was not a viable solution to use for the Priceline.com app designed for mobile devices, such
as the iPhone. A customer would have swiped the map to another location way before the rates would have been loaded. Even worse is that this swiping would cause a cascading effect, the swipe would cause even more search requests, in turn slowing the system down, causing the customer to start swiping again.

Many customers will be swiping at the same time, causing an significant amount of requests. The Priceline.com infrastructure could handle this, but it would cause problems at the underlying Inventory Sources. Therefore, the mobile apps started to use the Cache22’s grid to overcome this problem. Cache22 is less accurate; more inactive rates could be shown, but this is an acceptable tradeoff. It is acceptable, because the app initially has a browsing behavior; the app invites to just look around. When the customer selects a specific hotel then the single property call, as in the booking process, is made and the actual price is loaded, which in the majority of the cases is the same as the price that was shown on the map. This call will give the rates for the different room types and after selecting the room type the booking can be finalized. The section title, mobile search, actually is too restrictive, because a number of affiliates are using this way of search too. Therefore, when the Cache22 system is used as the source for a search it is called a cache search. Based upon the business model and agreements, affiliates can chose to retrieve their rates from a mixture of the normal system and the Cache22 system.

![Figure 4.2: The iPad app showing a big amount of hotel rates.](image)
4.3 Systems Processes and Systems

Robot search

The Cache22 grid is updated by customer and XML affiliate traffic. Even though the sheer size of this traffic will update many spots in the grid, holes and inactive rates will remain present in the grid. There are many possible rate combinations which might not all been searched by customers that use the site, which causes holes in the grid. Other combinations are searched so infrequently, which results in outdated or inactive rates in the cache. Remember that pagination and the hotel selected by the customer both strongly influence which rates served from the Inventory Cache and which are retrieved from the GDS. Only the GDS rates will update the grid, so only when a search is made and the Gateway decides it needs the rate from the Inventory Source the rate is updated, while the cache affiliates and mobile apps might have needed the active rate before the next non-cached search is made by customers or XML affiliates. The cache searches can have a different footprint, and again Google is a good example. When searching for an hotel on Google Maps™, as explained in Section 3.3, the next weekend is selected as checkin and checkout dates by default. Therefore, a robot has been built to, on a regular basis, gather the rates for these specific dates. This will minimize the amount of inactive rates that Google will serve.

There are a couple of other robots, each with their specific tasks. The robot processes are controlled based on information about the quality of the rates, which is gathered by the Cache22 system. A prioritized list of itineraries is fed to the applicable robot. Robots do not need any presentation logic and no logic to determine to use the Inventory Cache or the GDS—they only perform GDS calls, and therefore bypass the Gateway and serve their requests directly to the Search Engine. A tag to track which specific robot performed the search is annotated to each call. The Search Engine handles the search request in the same manner as any request and therefore sends the updates to the Cache22. The Cache22 system will be update, and thus the information about the rate quality. This completes they cycle. Currently, around 5 millions robots requests are made per day. The amount of calls the robots are allowed to make is determined by business rules.

4.3 Systems

There are many systems involved in the searching and booking processes. The following paragraphs will give a quick explanation of what each of this system does.

Website

WebTier The website is hosted across multiple servers, and the requests, using common techniques like load-balancing, etc, are distributed amongst these servers. Each of these servers runs WebTiers which apply the presentation logic that determines how to perform things like pagination and preloading.
Priceline.com Internals

Gateway  After the presentation logic is covered in the Webtiers, the gateway further processes all request types on a business logic level. For the hotel requests it consults the static and lookup databases to determine which hotels need to be searched for. It uses the Inventory Cache to determine which rates the Gateway needs to request from the Search Engine and which can be directly served from the Inventory Cache. After the Gateway receives the completed search results from the Search Engine, it applies the ranking logic and returns the total result to the WebTier.

Search Engine  The Search Engine is responsible for retrieving rates from the Inventory Sources. It does this based on the requests it retrieves from the Gateway. A single request can spawn many threads. The number of threads depends upon the number of hotels in the request; for each hotel a rate has to be retrieved. The results of the threads are gathered using business rules and are then returned to the Gateway. The Search Engine also forwards all the rates it retrieved to the Cache22 system.

The term Global Distributed System (GDS) is adopted from the industry, and is also known as central reservation system. A GDS is a system that is used to store and retrieve information related to travel and to make reservations. Most of the systems were originally created by airlines, but are now managed by independent companies. The term GDS is used within Priceline.com for the Search Engine, because it has a similar function and is a front for all the other GDS’s it gathers the rates from. Therefore, the rate retrieved from the Search Engine is called a GDS rate.

Cache22  The Cache22 system was first developed to provide affiliates like Google with rate updates. Due to it success it is now the source for the high demand mobile apps and other affiliates. The system keeps a grid with the different itineraries and their most recent rates. The system is also responsible for keeping track of its own performance.

Inventory Sources
As explained in the previous paragraph, the Search Engine retrieves it rates from other sources. These sources contain the rates for anything that can be booked. The collection of bookable things is the inventory. Hence, the descriptive name: Inventory Sources. Priceline.com uses multiple sources; among these are: Pegasus, Worldspan, Booking.com, and Agoda.

Pegasus  Pegasus is a GDS that serves hotel content and rates for 90,000 hotels in over 200 countries.

Worldspan  WorldSpan was founded by a group of American airlines. It is now part of Travelport and provides rates for 53,000 hotels, 465 airlines, 35 rental car companies and 40 tour and cruise operators.
Booking.com  Booking.com is part of the Priceline Group and provides Priceline.com with rates for 210,000 hotels. Booking.com started with hotels in Europe, but have extended their inventory containing hotels in 167 countries.

Agoda  Agoda is also part of the Priceline.com Group and provides Priceline.com with 30,000 hotels in the Asia-Pacific region.
PART III

5 PROCEEDINGS
5.1 Problem Context ........................................... 47
5.2 Problem Analysis ........................................ 50

6 Cache22 Esper ............................................. 59
6.1 Introduction ............................................. 60
6.2 Why Esper? ................................................ 61
6.3 Phase-I ..................................................... 62
6.4 Phase-II ..................................................... 77
6.5 Post-Processing ........................................... 79
6.6 Testing ...................................................... 81
6.7 Results ..................................................... 83

7 Cache22 EsperHA DSL ................................. 85
7.1 Motivation ................................................ 85
7.2 Design ...................................................... 86
7.3 Language Definition ..................................... 89
7.4 Implementation ......................................... 92
7.5 Evaluation ................................................ 99
As stated in the previous chapter the Cache22 system retains a grid with rates for every itinerary. The grid can contain holes where the rates are missing and it can contain rates that have been changed in the meanwhile. The ideal Cache22 system would have a cache without holes and only contain rates that are correct. Additionally, the behavior of the cache should be configurable to adhere to business rules. This chapter contains a description of Cache22 context in Section 5.1. With this context in mind, a problem analysis was performed before the start of the project to see in what way the Cache22 could become ideal. The result of the analysis are in Section 5.2 which was only altered when writing this thesis to change some terminology and correct some facts.

5.1 Problem Context

The Cache22 system performs an important role within the Priceline.com infrastructure. It is the rate source used by Priceline’s mobile applications and by affiliates. The system only provides hotel rates; although the techniques it employs could be applicable for other travel services too. The reason why its focus is on hotel rates is fairly simple: hotel rates are the largest part of the business and it was developed, at first, for Google Maps™ which only uses hotel rates. The Cache22 system was initially developed to serve Google Maps™ with rates; during the course of this project the system proved to perform so well that it became the solution for the challenges the mobile apps presented—see Section 4.2.

To perform its task the system keeps track of hotel rates. The hotel rates are stored in a structure that can be thought of as a grid. The columns of the grid represent days and the rows represent the different hotels. The grid is 330 columns wide; this has to do with the so called Advance Purchase (AP) property of an itinerary. The AP is the time difference in whole days between
5.1 Problem Context

Proceedings

the date of booking and the check-in date. The date of booking is referred to as the creation date. How long the customer wants to stay is another property of an itinerary; this is called **Length Of Stay (LOS)**. The **LOS** is defined as the number of days between the check-in and check-out date. Figure 5.1 shows the temporal ordering of the properties, and how the AP and LOS are calculated.

![Figure 5.1: A visual representation of the time dependent itinerary properties.](image)

In general, hotels have multiple room types available. The room types all have their own rate. The **min-rate** is defined as the cheapest of these rates. All the room rates, AP, LOS, and hotel combinations have to be stored. The sheer size of the possible combinations poses a big problem for the data structure and update performance. The grid would be $|AP| \times |LOS| \times |H| \times |R|$ where $|AP| = 330$, $|LOS| = 21$, $|H| = 210,000$ and $|R|$, the number of room types, dependents on the hotel ($H$), but the number of cells would be $1,455,300,000$ even without $|R|$. A partition table is used to overcome the problem such a huge grid poses. The partition is based on the check-in date; when a day passes by the partition of the elapsed date is dropped and a new partition with the check-in date 330 days later is created. Per partition the amount of cells is reduced by storing the room rate types for each LOS in one compressed **Binary Large Object (BLOB)**. The dimensions of the grid are reduced by a factor $|LOS|$. This gives a more manageable amount of almost $70,000,000$ records. Using the **BLOB** is a tradeoff between lower query execution times on the database servers, but a higher processing times in the application servers. To decompress, read, update and recreate the compressed object consumes additional processing time. The tradeoff is favorable towards using the **BLOB** for a number of reasons. First, the application servers can handle the additionally required processing power with ease. Second, the Cache22 project has its own dedicated application servers while the database servers are distributed, therefore shifting the computational power to the application servers provides a better control to manage the hardware resources and relieves the database servers. At last, only using queries is simply not a viable solution given the frequency of the updates.

The messages the Cache22 system receives are driven by traffic generated by searches and bookings, so it is driven by customers. A message can be an update, thus a new rate fetched from a GDS, or it can be a notification telling that a rate was served from cache. Early analysis of the traffic gave insight into the distribution of the demand. Figure 5.2 shows an indication of this distribution. The graph is an indication, because it only contains a limited set of data. The data contains the demand for a couple of hours on a single day,
and only contains GDS calls, and is limited in its AP and LOS combinations. The reason for this is that the data was retrieved by painstakingly executing limited queries—so that they would run within the timeout of the program executing the query—for each LOS. Even though the data set is limited it still provides good observations; the demand is the highest closest to the check-in date, short stays are the most popular, and there are spikes at whole week advanced booking—AP is 7, 14, or 21—and less noticeable but still considerable spikes at the whole week stays—LOS is 7, 14, or 21.

Figure 5.2: The number of request for each combination of AP ≤ 21 and LOS ≤ 21.
Next to demand there are two other metrics that are important: accuracy and coverage. The accuracy is based on the amount of rates served that were still active. Rates change all the time; because each hotel determines its own room prices based on demand and supply. The hotels publish their rates to the GDSs. These GDSs do not send any updates to the companies using them, instead the rates have to be looked up. The rates served from cache can thus become inaccurate. The ratio of accurately served versus inaccurately served rates is the accuracy. The number of holes in the grid will cause cache misses. These missing rates versus the available rates is the coverage. A rate can be missing because it has not been demanded before, or the rate has expired and has been thrown out of the cache. A rate expires when the rate is not retrieved from the GDS within a given time period. This time period is based on the expected lifetime of the rate.

A clear understanding of the implications is needed before—in the long run—the Cache22 system can take over the work of the Inventory Cache. Optimizing the accuracy and coverage will not be enough, because the whole system is now based on traffic generated GDS calls. These calls will not take place if everything is served from the cache. An analysis of the problems facing the Cache22 system is presented next.

5.2 Problem Analysis

In the ideal situation the Search Engine does not have to consult the Inventory Cache and every request would be directly passed through to the Inventory Sources—the GDSs. In this situation there will not be any cache, and no holes or inactive rates. Therefore, every itinerary would be available with the correct rate. The system would be providing complete coverage while being fully accurate. The problem that makes this situation impossible arises at the point where the Search Engine connects to the different Inventory Sources. Most of these GDSs are limited in the amount of traffic they can handle or allow. Therefore, the number of requests poses a problem, even though the Search Engine can handle the load with ease. A second problem is time; it takes a relatively long time to retrieve the rates from not only one but multiple GDSs. This results in limitations on how dynamic and fast the user experience can be. Due to these restrictions the Search Engine can not make calls to GDSs as often and as quickly as Priceline.com would like. Priceline.com uses a rudimentary cache to provide a quality service to its customers. The cache is the combination of the Inventory Cache and the Cache22 system, which were previously explained.

The cache system works, but there is room for improvement. Therefore, an analysis of what has to be done in order to improve the cache was performed before the start of the project. From this extensive analysis the following key question was formulated:

*Which techniques are applicable and how should they be used to create a cache that not only performs more transparently, but also can be configured to adhere to business rules?*
The customer should not be able to discern whether the rate was served from a GDS or from cache. For a customer it should appear as if the rate was served from the GDS and thus the cache would be barely noticeable—in other words, the cache should be transparent. An inactive, or no rate will make the cache noticeable, because the customer will see an incorrect rate or even no rate at all. At full transparency, the cache will have a accuracy and coverage that are both close to a hundred percent.

The analysis of this problem showed that improving the accuracy and coverage can be done in cycles. Each cycle consist of several stages and is, from now on, referred to as the Enhancement Cycle. Figure 5.3 gives an overview of the Enhancement Cycle. There are five stages starting with the Measure Stage, followed by the Predict, Improve, Regulate, and Coverage Stages. The stages and the their sub-question are stated in the following subsection.

![Enhancement Cycle Diagram](image)

**Figure 5.3: The Enhancement Cycle.**

**Stages**

Five different stages were defined from observing the original cache system and brainstorming about a future version. The stages are ordered in the following list according to their importance relative to their feasibility in terms of scope and complexity. The stages are logically ordered in the sense that each improvement will undergo each stage in the cycle in the same way. Note that this does not imply that solutions have to be implemented in the same order. A solution can serve only one type of enhancement or can be reused for multiple enhancements.

- Measurement
- Prediction
- Improvement
- Regulation
- Coverage
5.2 Problem Analysis

The Measurement Stage

This stage contains two main parts. The first part is to measure for the purpose of validation. Measuring is needed to come to know the actual performance of the cache. The effect of changes, made to the cache’s inner workings, is important to assess. The cache and the itineraries have many variables; the accuracy and coverage are among the important ones to be measured. Therefore, fine-tuning the granularity of these measurements is important, because the number of measurements will expand fast, due to the large amounts of data. At the same time there is a need to measure more granular. This many variables and their huge amount of possible values may obfuscate the ability to identify reasons for why the cache performs badly. Therefore, a correct inclusion and exclusion of variables in a properly aggregated result is critical.

It might be possible that, later on, new variables or another aggregation is needed—for example by a request from business. Due to the huge result sets it is difficult to just add a variable or transform these sets, because operations performed on these set will simply take too long. It might also be possible that information is needed that is not stored. The measurement system has to be adaptable to cope with these changes, and thus configurable in how and what it measures.

The second part is the measuring of the real rates. The previous paragraph was about measuring the performance of the cache and can thus be used to validate the relative improvements. To be able to say anything about the performance versus the actual rates in the GDSs, a mechanism that samples and derives statistics is needed. The challenge here is to make as few unnecessary calls as possible, but still have a valid statistical representation at the granular level that is needed. Unnecessary calls would include calls for rates that are already requested by bookings from customers or calls that do not help to detect a rate change. This requires an effective mechanism to reuse and complete the calls from the users, because avoiding unnecessary calls is the reason for the cache altogether.

The following sub-questions are extracted from the two previous paragraphs:

1) What is a valid way to measure the actual performance of the cache? Which events should be included and which should be excluded?

2) Which variables are of importance to measure and how can they be aggregated and summarized into a result set that is concise enough to be usable?

3) In which way can support to easily allow changes be incorporated in the measurement system? Changes could be the addition of variables or different ways of aggregation.

4) How can the customers’ GDS call be efficiently completed with a minimal amount of extra call to provide a statistical valid data set for the different levels of granularity?
The Prediction Stage

The idea behind the prediction stage is that the historic data—e.g., gathered for measuring the performance—is used to predict future variables. For example; if some AP and LOS combination has proven to be 95% accurate for the last couple of days, derived through statistics, it might be possible to predict that it is close to 95% today. For a given variable combination, this knowledge can be viable to decide whether to use the cache at all: when a prediction for a given combination of variables is less than a given percentage, the cache is circumvented and a direct GDS call will be made. This call has the benefit of placing the accurate values into the cache for that given combination, because all GDS calls update the cache. In this way the cache will be self-healing or self-correcting for these combination. A continuous loop back mechanism is needed for the self-correcting behavior to be possible. This loop back mechanism must periodically perform the statistical analysis automatically and provide a formula for the prediction, because when handling the customers’ request there is not enough time for a complex calculation.

This adds up to the following questions:

5) How to generate the prediction formula and what will be the result variables?

6) Which variables can be included or excluded from the formula generation process to minimize its calculation time while remaining accurate enough? And how frequently must the formula be generated?

7) How to make the formula generation process and the percentages configurable, what are good values for the percentages and what level of granularity is needed for them?

The Improvement Stage

In contrast to the Prediction Stage, the Improvement Stage will use the historical data to actively improve the cache to prevent mismatches. This is different from the Prediction Stage which uses the historical data to make a decision to use the cache or not, but does not improve the rates before the combinations of AP and LOS are requested. The rates—thanks to the loop back mechanism—will have a higher chance of being a match the next time, but will still cause a mismatch when, after a while, the rate is requested again.

The Improvement Stage will be updating the rates before they are requested. Improvement has two major benefits: the amount of first mismatches will be less and the requests to the GDS are used more efficiently. The better the rate change behavior is comprehended, the better a robot can do the job of requesting rates from the GDS just after they change. The need for a good comprehension of this behavior will become more and more important; when the cache is running in production the GDS calls will drop and thus the cache will not be filled and updated anymore by customer traffic. The way hotels change their rates can be analyzed to determine when to request rates at the GDS. It is possible that there are relations between the actions of certain hotels, or hotels with a certain star rating, or in a certain area, etc. Such a relation could be that
if a hotel is changing its rate that other similar hotels also will change their rates. The relations will help to better update the cache, because if one change is noticed related rate changes can be expected and be updated in the cache. Additionally, relations could be useful to explain unexpected actions. For example Hotel2 does not change its prices generally on a Monday till 11:00 a.m., but when Hotel1 does change it price, Hotel2 will always follow. So in the event that Hotel1 changes its price at 8:00 a.m. an unexpected event happens for Hotel2 when it changes its rates before 11:00 a.m., but this action will be noticed by the cache thanks to the observed relations.

Analysis of customer behavior is of great value. Knowing that particular combinations are rarely requested for a certain period of time will save the robot work, because then the decision could be made to always serve these directly from the GDS. Another case: some rates are requested for the result listing page of a search but are rarely booked, because customers seldom choose that certain hotel. This could have many reasons, for example, the hotel is the most expensive one or has a low rating. For these cases, lowering the confidence interval for the correctness of that rate will not necessarily translate in a higher chance of a mismatch to be actually noticed by the customer. That rate will be rarely retrieved from the GDS because is not often booked. Lowering confidence intervals does lower the work for the robot. Customer behavior could also be used to increase the confidence interval for certain cases. For example for competitive and frequently requested rates. Machine learning techniques could provide a good solution. Especially data mining and pattern analysis are candidates to enable the discovery of these behaviors and relations.

This raises the following questions:

8) To what extent can the robot be self-learning to improve the general mismatches?

9) How can the robot learn the behaviors needed to shop more efficiently for rates?

10) What is the feasibility of a system where hotels send notification of their rate changes, which can be used by the robot to shop for related rates?

11) How to verify the effectiveness of the robot? Is it possible to visualize the inner working of the robot and use it to correct its behavior?

The Regulation Stage

The prediction formula to make the decision to go directly to the GDS can be seen as a basic rule. The Regulation Stage comprises such rules. In essence, the prediction formula is a conditional exception rule, it determines the cases when not to use the cache. On the other hand, a rule could force the use of the cache and not allow to make a GDS call. To understand why these rules are needed, an explanation on how this cache is different from a normal computer cache follows.
A computer cache is a smaller subset of the actual storage which it is caching. Although the cache is a subset, the subset is located on faster type of storage. This will speed up the access times, because frequently looked up items on the main storage will remain in the cache. A computer cache performs well if it has the most lively, most frequently looked up information, stored. This is, in a way, similar to the Cache22 system, except that this cache system is not a subset. The Cache22 has to contain the whole set, otherwise it will have a negative impact on the coverage. The focus is not on quickly providing the most frequently accessed data, but on the correctness of it. The cache has the primary goal of reducing the load on the underlying systems, while the reduction of access is a secondary goal. Minimizing the load is not the goal; in the Cache22 system it is good to make use of the full capacity of a GDS to retrieve changed rates, as long as it is not negatively effecting the underlying systems. A normal cache updates its values when the demand for them drops, while the Cache22 system has to replace values that have changed. A major difference is that a normal cache would know when a new value is written, but because the GDS does not send notifications the Cache22 is not aware of changes and has to actively find out that they happened. Therefore, the rules are applied to determine when and in which cases the rate has to be retrieved from the GDS again.

The self-correcting cache, with the suggested improvements from the previous stage, will eventually come closer to the 100% accuracy and coverage. At the same time the cache will defer fewer and fewer requests to the GDS calls. The unused capacity can be utilized to direct a robot or to adhere to specific business rules—e.g., some itineraries are not allowed to be served from the cache. The robot could actively gather itineraries from the GDS—know as shopping, that are contributing to the few percent of inaccurate rates, or the robot could dedicate shops to increase the coverage. These rules will be the mechanism in place to direct the robot. The rules do not only have to be based on business logic; the people who manage the cache could be setting rules to steer the robot and try to increase the accuracy and coverage for the combinations unaffected by the self-correcting mechanism or by the improvement techniques from the previous stage. The rules thus regulate and corrects the behavior of the cache: hence the Regulation Stage.

The questions related to this stage are:

12) What is the point at which the underlying systems can not handle the load?

13) Can a language be designed that is expressive enough for people that manage the cache but at the same time is clear enough to be used by people that apply the business rules?

14) How can the rules be applied in such a way that they are not conflicting or overlapping and are executed in the right order?
5.2 Problem Analysis

The Coverage Stage

Imagine all the different combinations of AP and LOS as a grid. The ranges considered are 1...21 and the AP considered ranges from 1...330. This creates a grid of 6930 combinations. This is a size which is still manageable, but this grid exists for every hotel. Priceline.com roughly offers rate from 210,000 different hotels. Now the grid is expanded with another dimension and contains 1,455,300,000 rates. Hotels change rates continuously and therefore the cells are volatile. To even complicate the matter more, hotels do not just have one rate. In the listing page, shown after a search, the rate displayed is the min-rate but a hotel could have multiple rates dependent upon the room quality, location, etc. and special promo rates are another possibility.

The coverage can be defined as $1 - h/n$ where $h$ is the number of holes in the grid and where $n$ is the total number of possible items in the grid.

It's an enormous amount of data; and it consists of items for which it should be possible to update them every couple of seconds. This a challenge for the datastructure and the infrastructure. There is the notion that the coverage affects the accuracy, when the coverage is lowered the accuracy increases. A considerable amount of these combinations are probably rare enough that it makes sense not to store these rates and serve them directly from the cache. Note that this is different than in the Prediction Stage, because this time no decision has to be made; the rate simply does not exist in the cache. The Prediction Stage is about the accuracy of a rate that does exist in the cache. In the original system coverage is provided by customer traffic. The rates are kept for a maximum of ten hours and within ten hours there are a lot of combinations requested, but there is a substantial amount that is not requested and that is why there are holes in the cache.

For normal customer traffic it is not that important to have a high coverage. The customer searches for example for hotels in a certain region, then a list will be returned. This list contains the result of the rates of all the hotels in that region which could provide a rate on time. Some rates will come from the cache and some are served from the GDS. If a hotel is not in the cache it is likely to show up in the list with the rate from the GDS. A timeout will cause the system to ignore all the rates from underlying systems that took too long—e.g., due to problems on their side, so if a rate is not covered and it could not provided it will not be in the list. This is not really noticeable by the customer, because they are unaware that a hotel is missing. A coverage close to 100% would be a benefit in terms of response time and in lowering the load on the GDS. This quality of service is a benefit for the hotels too, they will appear in the list even when their GDS systems are having problems.

Another situation where the coverage is important is in the case of meta search engines. A good example is Google Maps™. When searching for hotels near a location it will give a list of hotels. The prices per hotel from the different providers are shown in this list. The way this works is that Google wants the providers to upload their rates to them. This is why coverage is important, because if no rate is provided it is not shown in the list of prices and thus missing out on possible revenue. On top of that, there are requirements on
the accuracy of these prices. A well-covered cache with a respectable accuracy is a powerful system for marketing and many other purposes, because it enables passively showing accurate rates without needing a search request—e.g., in advertisements.

Coverage has the following questions related to it:

15) What is a good infrastructure to cope with the amount of data?

16) Which datastructure design should be used to compress the data, but at the same time not cause much overhead on the application servers?

17) What is the relation between the coverage and the accuracy? And how to pinpoint the troublesome combinations?

18) How to measure the amounts of misses from the (meta) search engines?

19) What is the acceptable coverage and accuracy combination, and how does this translate to maximizing revenue?
The Cache22 Esper system is a solution for the Measurement Stage and a partial solution for the Prediction Stage. The system will use Complex Event Processing (CEP) to match patterns in the messages coming from the Cache22 system. The pattern gathers information needed for an accurate measurement of the cache's performance. The information is post-processed into a compressed and aggregated result that can be stored in Priceline's databases. The system was developed in two phases.

Phase-I is the first deliverable of the Cache22 Esper system and described in Section 6.3. It contains the main mechanics to connect to the Cache22 infrastructure. The system processes what are called Accuracy Events. The data generated by customer traffic on the Priceline site is collected and provided by the Cache22 infrastructure and turned into these Accuracy Events by the Cache22 Esper system. These events are further processed by Esper. The Esper Engine aggregates and transforms the events into result events that can be post-processed.

Phase-II—Section 6.4—is developed as the (temporary) solution for the memory problems experienced by Esper. It mimics only the specific behavior of Esper needed to measure the cache accuracy.

Both phases perform the same post-processing, which is explained in Section 6.5. Section 6.7 contains the results achieved by the system and Section 6.6 explains how the system was tested.
6.1 Introduction

The overall purpose of enhancements to Cache22 is to increase the accuracy and coverage. In the proceedings five stages were specified, beginning with the Measurement Stage (see Section 5.2). The Cache22 Esper project will focus on the Measurement Stage in Phase-I. The Measurement Stage was chosen, because each of the other stages are dependent on a good measurement and will benefit from the insight gained. Shortly after the problem analysis, it was evident that a CEP system had to be used—see Section 6.2 for the reason why Esper was chosen. The inspiration to use CEP came from the Algorithmic Trading world, where it is applied to tackle problems similar to those encountered in the Cache22 system: analyzing high volume traffic, detecting complex patterns in the data, maintaining configurability and flexibility.

A CEP system is applicable for the other stages as well. The EPL could be used as the rules for the Regulation Stage. The use of events will make the Cache22 system flexible and the ability to filter events using specific variables, will act as a solution for the questions that are part of the Prediction Stage: what variables to include and exclude while producing the result quickly. The statements in the CEP can be used to detect patterns, like the patterns found during the Improvement Stage and a statement can act as a triggering mechanism for the robots. Therefore, the Cache22 Esper project has the Measurement Stage and partially the Prediction Stage as its main goals and is designed with the other stages in mind.

Purpose of Phase-I

The purpose of Phase-I is to build a dynamic measurement system using Esper. Before it is possible to say anything about the effectiveness of an enhancement, a stable measurement instrument must be available to measure the situation before and after the enhancement, so the enhancement’s effectiveness can be determined by comparing the two situations. The Cache22 Esper system will produce the accuracy information.

The Cache22 system receives the raw information, in the form of messages, from the Search Engine. A message contains the itinerary and its source: cache or GDS. The high volume of these messages make measuring a difficult task; around 45 million messages per day will be forwarded by the Search Engine. Taking a measurement is complicated, because to make accurate measurements the instrumentation must be able to match patterns across these messages. This combination of high volume and patterns renders the problem a non-trivial one.

Before the project, each message was stored as a row in a table. The accuracy was determined by running a lengthy query on the table. The query was limited and only measured difference between GDS rates. The reason that the queries are limited is because they take a long time to execute, even on the fast servers that are available. Another problem with the query is that queries are typically static, so when the data is needed in another format it will need an additional query to transform the data, which will also take additional pro-
cessing time, making the whole process inflexible. The result from the query produces an indication of the accuracy, but is not correct because it ignores the cache message.

Even if the problem of the queries is somehow solved, the problem of actually storing the message will become apparent. The Cache22 infrastructure already makes a decision on what to store from the information generated by customer traffic. This is the first place where expressiveness of the data is traded in to save space. The raw data will expand every day and therefore a very rudimentary mechanism called partitioning is put in place. The partitioning will keep a partition for each day and will remove the oldest to make room for the newest. For now, the database administrators will only allow the storage of two weeks of raw data.

The Cache22 system will use pattern matching to include the cache message, producing the correct accuracy value, and will aggregate and compress the number of rows that have to be stored.

**Purpose of Phase-II**

Processing the messages in an event driven way proved to be useful in Phase-I. It is possible to aggregate the messages and produce the correct accuracy results, but to be able to continuously run the system with full production traffic, the problems that Esper was giving, see Section 6.3, had to be overcome. Due to the promising results from Phase-I, there was pressure from business to have the system production ready. Additionally, the Priceline.com managers were eager to see and use the other, Regression, post-processing results—see Section 6.5. Therefore, the decision was made to abandon Esper for the meanwhile, and resolve the issues after the system is up and running. The problems experienced were mainly memory usage issues in Esper, which were likely caused by the volume of messages combined with the pattern that had to be matched. The Cache22 Esper system has a modular design, see Section 6.3, and the non-Esper part of the system was working correctly; actually the overall system was working correctly when the load was limited. The decision was made to replace Esper with a custom Java implementation that would mimic the CEP behavior for the specific case of the Cache22 Esper system, until the problems with Esper were resolved. In this way the results are produced and provided to business, and the rest of the system could be properly tested under load. The EsperHA edition looked promising for resolving the issues, and therefore replaced the custom part from Phase-II in the DSL version—see Chapter 7.

The post-processing part is developed over the course of the two phases. Therefore, it is separately explained in Section 6.5.

6.2 Why Esper?

There are a couple of CEP systems to choose from; ranging from large commercial players to open-source engines. Why Esper was the CEP system was chosen for the project can be briefly explained: it was a decision driven by
business. Nevertheless, Esper was an acceptable candidate for the project, and was endorsed by a comparative study [7] which concluded that Esper is a strong performer:

EsperTech, the only open source product in the evaluation, earned strong scores for runtime architecture, platform administration, and event processing features. Like many open source projects, EsperTech lacks strong tools for business end users and administrators, which put it at a disadvantage against all other platforms in the evaluation. EsperTech's highly embeddable Java architecture, strong CEP feature set, and open source status make it a top candidate to be embedded in other vendor tools or applications.

[7]

Using the open-source version to become acquainted with Esper, made it possible to decide whether it is a good product before a license was purchased, and therefore there was enough reason not to overturn the decision of business. Another positive fact is that Esper is Java based, and thus could be easily integrated in the current development process. Esper was previously used with another project, therefore there was knowledge about Esper available within the company. Esper was causing problems back then, but many versions had been released since then, so Esper was given another chance. The problems with Esper were kept in mind while designing the Cache22 Esper system. Its modular design would make replacing Esper easy. Esper was a good choice being the best of most worlds, altogether there was no need to overturn business's decision.

6.3 Phase-I

Phase-I of the Cache22 Esper system uses the HashMap representation for events, because HashMap's are generic and can easily be extended with another variable without the need to change any code. The HashMap representation of an event can be registered using an EPL and there is no object that has to be referenced during runtime by the Esper Engine—when using a POJO the Esper engine has to be able to instantiate the object.

After getting acquainted with the Esper platform, by executing some examples found on the Internet, the initial design of the system was made. The documentation was not that helpful to people new to the subject, although many features were explained, many connections were missing, turning the documentation into a game of 'Connect the Dots' without the help of numbers. How to separate the engine into its own process was unclear either, so therefore the system still has the support to embed the engine, because that is how it was implemented in the beginning. In the meanwhile a web-service interface and a website was partially build to try to make it possible for the Esper engine to become a standalone process. Luckily, after a couple of weeks the purchase of the EsperEE license was completed. It became apparent that EsperEE had the functionality to run the Esper as a standalone process; it has
the Input/Output adapters and the GUI. At this point it was clear that it was actually possible to have a design with a standalone Esper engine. The design is explained next.

Design

The initial design is two fold: it supports the embedding of the engine and running the engine standalone. To have the engine running standalone was a major design decision. There are several reasons for this. The standalone version forced the design to strongly separate the code base of the processes. This separation meant that it would be easy to replace the Esper engine, since its performance was not guaranteed from previous experience—see Section 6.2. The two processes could be run on different systems spreading the load if needed. In high volume and threaded systems it is hard to determine the exact source of a problem encountered—a lesson Priceline learned from experience, therefore having two different processes is beneficial for figuring out whether the Cache22 Esper code or the Esper engine is giving the trouble. Managing the memory usage and the garbage collection could be done for two processes now instead of one, allowing for tailored settings. When analyzing two different processes, e.g., heapdumps, there is no need to determine which part belongs to what part of the system, because they are already separated. This proved to be useful when analyzing the heapdumps, which was troublesome already due to their size: objects in memory belong to the process that created the heapdump, so there is no guessing which object belongs to which part of the code.

It was decided to not use the option to separate the pre- and post-processing part of the non-Esper process. The reason for this is that although separating processing has the benefits just mentioned it also comes with extra work and separating the pre and post processing would surpass that break-even point. Still, the combined Processing part of the system is designed with the two different parts, pre- and post-processing, in mind and could still be separated if needed.

The design for the Cache22 Esper system having two separated processes contains the components shown in Figure 6.1. The different processes are shown by the color background: blue indicates the Esper engine process and orange and yellow indicate the Processing process. The orange and yellow parts denote the separation that could be made, if needed, in the Processing process.

A general overview of the system will be given next, after which the design decisions that led to the structure are given, and in Section 6.3 the particular components will be explained in more detail. The overview will be presented in the order in which the messages flow through the system. The flow starts at the Cache22 system.

The Cache22 system is the system that already exists in the Priceline architecture—see Chapter 4. The Cache22 Esper system is build as a subsystem of the overall Cache22 system. The information that it receives comes from the overall Cache22 system and the resulting information it produces is used and stored by the
Cache22 system. The Cache22 system produces the GDS and cache messages, and these messages are put on an IBM Message Queue (MQ).

Before continuing to explain the message flow, how the Cache22 Esper system is started needs to be described. First, the Esper engine had to be started. This is done by creating a new EsperEngine object with a url to run the Engine at and call the start method. The url acts as a kind of identifier allowing for
multiple engines to be running on a single machine. The complete Processing part of the system will be started by instantiating the Bootstrap, OutputEngine and the chosen Feeder components. The Bootstrap would create any of the configured Processors needed and makes them register their EPL statements. In this way the whole system is up and running, ready to process the message flow.

The first part of the Processing process is the Feeder. There are a couple of different Feeders available, but they basically perform the same task of transforming the messages into events and sending them to the Esper engine. Each Feeder does this by connecting to the socket adapter in the Engine and sending the events using that connection. The reason for the different types of Feeders originates in the source of the messages; a file, a table, the IBM Message Queue [MQ], and the post-processed events are message sources.

From the Feeder the events arrive at the Esper Engine, through one of its three external interfaces. There are two input interfaces and one output interface. One of the two input interfaces is capable of receiving events: the socket adapter, and the other input interface is for administration purposes: the JMX interface. The JMX interface is used by the Processors to register their statements. The output interface is a MQ and is used to put the resulting events on. The Forwarders take care of the task of receiving the Esper result events and forward them onto the [MQ].

Back at the Processing process the [MQ] is read by the Dispatcher. The Dispatcher reads a processed event and uses the Event Manager to decide where to dispatch the event to. It decides based upon information provided by a Processor. The Processor tells the Dispatcher to which of its Subscribers a specific result event has to go to.

A Subscriber then produces an OutputJob dependent upon and from the events it receives from the Dispatcher. The OutputJob itself performs the specific task needed for further processing the Event and this is where the actual post-processing happens. The specific post-processing for the Accuracy Events is explained in Section 6.5.

There are two predefined jobs: the PostFeedJob and the DB job. Both jobs will be given to the OutputEngine which assigns the job to a thread that execute the jobs. The Post-feed job will send the (post-processed) result event back to the Esper engine using the ConcreteFeeder type. The DB Job will perform the task of preparing the result event to be stored in one of Priceline’s databases. Using a table helper the job will transform the event into a message that can be put onto a queue. The messages on that queue will be processed by the Cache22 system and stores the event in the proper table. This completes the process flow.

The design for the overall processing of the flow of messages was based on several reasons. The reason for the separate processes was previously given, but the design also allows the Processing part to be implemented multiple times and allows each implementation to use just one and the same Esper engine. The reason to use multiple implementation is to clearly segment the specific tasks that have to be performed and the use of different kinds of source information. The usage of different sources is an additional reason to keep the
Esper engine as clean as possible, and have a pre-processing stage.

The Feeders are part of the pre-processing stage. The Feeders provide support for a variety of sources and due to their architecture can be easily extended with another type of feeder for a different source. The Feeder provides flexibility in the case of the Cache22 system with the MQ. If the Cache22 system would have to send the messages directly as events to the Esper engine it meant that code changes would have been needed to a running production system every time something changed in the Cache22 Esper system. Therefore, the preprocessing phase was put in place with a MQ reader to disjoint the Cache22 system from the Esper engine. The Feeders also act as a buffer for crashes. If the Cache22 Esper system would crash for some reason it would have had an impact on the Cache22 system if they were not disjointed, which would be a serious problem for a major backbone system like Cache22. With this design, the Cache22 system just puts its messages onto a queue and any program can do with it what it wants.

The decision to use sockets is based on performance reasons. The two parts of the Cache22 Esper system will be running within a single environment so the communication using sockets is sufficient. Using sockets has a performance benefit compared to the other adapters, because it does not have any overhead from a specific protocol.

The forwards in the Esper engine have a similar motivation as the reasons for the Cache22 and Cache22 Esper decoupling. The Forwarding and the use of the queue makes the Esper engine and the post-processing phase disjoint. Thus in case the post-processing process crashes the Esper engine is not influenced and keeps on putting its results onto the MQ. Additionally, it allows the result events to be replicated across multiple queues, something that can be easily done with the IBM Message Queue (MQ). A completely different process can therefore receive the result events on a queue they read from. The reason that a queue is not used between the feeder and the Esper engine is rather simple, the Feeder just stops with sending when it detects that the engine is down; reading it from one queue and putting it in another queue would be redundant.

The thinking behind the Processors is that they decide what happens with the messages from the Cache22, and because there could be different kinds of messages there also can be different Processors. This keeps the Processors and their task clean and clearly segmented. Therefore, they can also produce and register the EPL statements and control how the result events are forwarded by the Dispatcher. Having multiple Processors is convenient while debugging, because they can be easily turned off and therefore stop the whole processing chain of their specific task. Thus when a new processor is causing problems, they can be pinpointed more easily by turning all the other Processors off.

The reason for the Jobs is to have a high performing system using a thread engine, and a way to reuse the implementation of a specific task for other event types. The Subscribers have the task of converting the event into a Job, but from that point on the Job knows how to perform its task with that event type. The Post-feed Job is designed to have the post feed mechanism in place without having to use EPLs. The DB jobs were provided for the simple reason
of functionality, the Cache22 Esper system would not have been functional without a way to save the results.

The next section will explain how the design is implemented.

**Implementation**

The implementation is segmented in a number of classes that perform a specific task together. These classes generally are part of the same package. Therefore, this section will explain the classes per task or package. The basis of the system is designed in a generic way, but the implementation contains a couple of specific parts for measuring the accuracy data.

![Diagram of Feeder Classes](image)

**Feeders** At the start of the project the events would be read from the table containing the messages that were directly converted to rows. This was done with the TableFeeder. The code for the TableFeeder was written before EsperEE was available, which has native support to play back historical data from a table. However, being able to play back the information from the table with the TableFeeder did lead to a piece of valuable information and that was that it was hard to see, when performing the playback, what was happening within the system and to see if the statements were doing their job right. Therefore, the File Feeder was written. The File Feeder reads the events from a file. The File feeder is particularly useful for testing, because the sequence and properties of the events can be specified manually. This does make it possible to verify the behavior of the system. The ConcreteFeeder is the most basic variant and directly sends the HashMap to Esper. The ConcreteFeeder is used for the feedback mechanism.

The **MQ** Feeder is the important one, because it is the feeder used in production. With the help of the Input engine, which is explained next, it reads the messages from the queue. It has the ability to order the messages it reads according to their timestamp, ensuring the ordering of the events. However, in the production system this caused a bottleneck and the ordering could not
be guaranteed by the Cache22 system anyway, therefore ordering it on the creation time stamp is of little use. The ordering is not really important due to the large number of events and if the messages are processed fast enough the chance that two subsequent GDS events for the exact same itinerary switch in order is almost non-existent, because there would not be two GDS calls short after each other.

All Feeders inherit from the AbstractFeeder. The AbstractFeeder creates the connection with the Esper engine using a socket and takes care of the actual sending of events. It has a mechanism to reset the sockets after a number of events, which is important because sockets keep a in-memory list of sent messages. This will cause a significant memory usages and therefore the sockets have to be reset. The sockets could not reset every time a messages is sent, because they will remain in control of the port for a while causing the operating system to use another port for the socket. With 45 million messages a day it is easy to see why this would cause problems too. Therefore, the OutputStream is reset after a set number of events is sent.

The Table and the File Feeder are based on time. The AbstractFeeder is extended by a AbstractTimedFeeder, which contains a basic timer mechanism. A general version is implemented by the GenericTimedFeeder, because it became clear that the Table and File Feeders need the same specific timing mechanism. The mechanism offsets the current system clock to the first event that has to be replayed, and the rest of the events are sent when their creation date time equals the time of the system clock minus the offset. The GenericTimedFeeder also allowed the rate to be increased and to specify a start and end date time. The abstract classes form the basis for the implementation of future feeders that use different sources.

Figure 6.3: The StatsDataReader architecture.

StatsDataReader or Input Engine. Figure 6.3 shows the StatsDataReader architecture. The StatsDataReader is the specific name for what actually is a thread engine that has the task of reading messages from the MQ. The StatsDataReaderEngine is started as part of the main method and is instantiated as...
a singleton instance. It implements the BaseEngine that creates the number of threads specified in the configuration. These threads will execute BaseJobs from a queue inside the BaseEngine. The BaseEngine will receive notifications from its threads when they finish, and the BaseEngine will assign the finished thread with the next job in its queue. The jobs are created and placed onto the queue by a JobPopulator. In this case the StatsDataJobPopulator is instantiated multiple times and reads the messages from the MQ. A StatsDataPopulator will place the message inside a StatsDataJob. The StatsDataJob will be executed by the StatsDataThread at the moment it is at the front of the queue inside the StatsDataReaderEngine. Each thread has a instance of the MQTimedFeeder and thus a socket connection to the Esper engine. When the StatsDataJob is executed it tells its MQTimedFeeder to send the AnalyzedStatsData object to Esper. The AnalyzedStatsData object is the message that comes from the Cache22 system and contains a batch of GDS and cache messages. The MQTimedFeeder still has to transform the AnalyzedStatsData object into events. This is done with the help of the Event classes and are explained next.

![Event Architecture Diagram](image)

**Events** Figure 6.4 shows the Event architecture. The HashMap event representation is used during this phase of the project. To be able to convert the source format of each of the feeders into events, a description is needed on how to convert this source format to the HashMap representation. Therefore, every event implements the IEvent interface providing it with a toEvent method to make the conversion. The timed Feeders need to know what property determines the time of the event, therefore they implement the ITimedEvent interface providing it with a getTime method. The IMapEvent interface requires the events to have a getName() method providing the name of the HashMap event at the point it is sent, which is needed by Esper. The ITimedMapEvent is a stub interface to force an event to be both a HashMap and a timed event, it is a stub because this could also be achieved by adding both interface to the implements part of the class. The AbstractMapEvent has a standard implementation of the register method that registers the event at the Esper engine using a Processor, provided as an argument of the method. The IMapEvent forces a HashMap event to have a definition method, which provides the list of keys and their value types, which represent the properties of the event. The
6.3 Phase-I

AccuracyEvent is the actual HashMap representation used by the Esper engine. It is inherited by a source specific version. The source specific version have a toObject method that place the source values at the right properties of the AccuracyEvent. In the feeder itself it uses the source class specific toEvent method to convert each value to the right type of the AccuracyEvent definition.

Note that all the source specific AccuracyEvents already implement the ITimedEvent interface thanks to the ITimedMapEvent interface, but have been added as implements for clarity.

The description above is quite abstract. The AccuracyFileEvent will be used as an example to further explain the structure. The FileFeeder implements the AbstractTimedFeeder which uses reflection to determine the name of the event class to use. It requires the class to be an ITimedEvent. The FileFeeder itself then tries to upcast that event class to the ITimedMapEvent, because it uses the HashMap representation, the event class has to be an ITimedMapEvent. The FileFeeder then reads the file and uses the toObject method, per line, to convert the read strings from that line, which are the event properties, into the AccuracyFileEvent. The AccuracyFileEvent currently contains the property values as Strings, because the toObject converted the String[] array into a the general form of an HashMap event: HashMap<String, Object>. That is why the method is called toObject. The FileFeeder uses the GenericTimedFeeder for its timing mechanism, and when the GenericTimedFeeder decides to send an event it calls the toEvent method to turn the HashMap with String into the AccuracyEvent. The toEvent method of the AccuracyFileEvent converts each String to the appropriate type specified in the AccuracyEvent definition. Using this structure every feeder can send any kind of event.

Esper The Esper component basically is a single class that starts the Esper engine with a number of settings loaded from a configuration file. It starts the Socket Adapter, the GUI, DDS and JMS—which are used to output events on the GUI, and the JMX interface. The Esper engine can be configured to listen to unmatched events and will log them when they happen. Unmatched events are received but not processed by any statement. For example, if the AccuracyEvent stream is filtered on having a price above $100, all the AccuracyEvent with a price under $100 will be unmatched events. The most important part of the Esper engine is the CreatedEventListener. This listener will notice that a new statement is registered at the engine and will work as a hook for the Forwarder component, which will be described next.

Forwarder The Forward component exists out of two classes: a MQListenerForwarder and a MQSubscriberForwarder. They perform a simple task and that is to listen or subscribe to a statement and then forward any of the result events they receive on their istream or rstream to respectively the configured Listener MQ or the Subscriber MQ.

For example, the MQSubscriberForwarder implements the update methods needed to receive the resulting Hashmap events. It takes the HashMaps, adds the statement name—which was provided in the hook mentioned in the
previous paragraph and is used for identification purposes later—and wraps them inside a MessageObject. Finally, the MessageObject is placed onto the MQ.

Dispatcher Figure 6.5 shows the Dispatcher architecture. The Dispatcher makes use of the OutputEngine, which is a thread engine like the InputEngine. The Dispatcher retrieves the result events from the MQ. The Dispatcher has one additional class which is the DispatcherEventManager. The manager has a list of subscribers for the given statement names. At the moment an event is read from the MQ, it checks if there are subscribers for that event by checking the statement name that comes with the event. If there are subscribers, the event manager will clone the event and send it to the appropriate subscribers. This process can be seen as the firing of events.

The DispatcherOutputJob is a specific OutputJob and gives the event(s) to the DispatcherEventManager for them to be fired. The thread gets its instance of the DispatcherEventManager from the OutputThread when it has to process a DispatcherOutputJob. The DispatcherEventManager uses instances, because using a singleton instance caused a bottleneck. The instance does use static variables for the subscriber list, because they are only read most of the times and therefore do not cause any thread safe issues.

Processors The Bootstrap class, which is instantiated once during the start of the program, looks in its config for a list of classnames. These are the names of Processor classes, which are instantiated, also once, using reflection. After they are instantiated the Bootstrap class calls the register() method, causing the Processors to do their job. Their job is to create statement, windows, and events within the engine. The Processor also adds subscribers to state-
6.3 Phase-I

Figure 6.6: The Processor architecture.

Figure 6.7: The Subscriber architecture.

**Subscribers** Figure 6.7 shows the Subscriber architecture. The Subscribers have a simple task: they receive the events from the Dispatcher and turn them into an OutputJob. Both TimeExceeded subscribers and the CacheUnknown subscriber are there to handle the exceptional cases as part of the way the statements produce the result events—see Section 6.3. The DbUpdateSubscriber's name is not quite right, because it turns the result AccuracyEvents into jobs,
those jobs perform the post-processing—see Section 6.5. The RegressionSubscriber acts a bit different, it cannot directly turn the result AccuracyEvent into a RegressionJob, so it adds the events to the RegressionConsolidator, who in turn creates the RegressionJobs as part of its post-processing—see Section 6.5. All these jobs are OutputJobs and processed by the OutputEngine, which is explained next.

**Figure 6.8: The OutputEngine architecture.**

**OutputEngine** Figure 6.8 shows the architecture of the OutputEngine. The OutputEngine is another thread engine. This time its task is to execute OutputJobs. These jobs are part of a thread engine, because an OutputJob could have a lengthy post-processing part; for example the AccuracyDBOutputJob and the RegressionJob—see Section 6.5. The PostfeedJob is the only one left; the task it performs is to use a ConcreteFeeder to send the event that is contained within the job back to the Esper engine.

**Figure 6.9: The TableHelper architecture.**

**TableHelpers** Figure 6.9 shows the TableHelper architecture. The TableHelpers are part of the (post-)processing performed in an OutputJob. They are a convenient way to save result events to the Priceline tables. The AbstractTable connects to the MQ that processes database messages. It provides support to prepare the messages and convert the data to the format needed. The specific AccuracyTable and RegressionTable basically are blue prints for the specific tables and contain methods to transform the specific result events
into the format that adheres to these blue prints, so they can be further processed by the AbstractTable’s methods.

This completes the complete flow of data. However, the statements inside the Esper engine were not addressed in this section. These are specific to the Cache22 Esper problem and are explained next.

**Accuracy Measurement and Statements**

The statements are EPLs that are being executed by the Esper engine. Before explaining the specific statements, the overall Cache22 accuracy measurement problem is explained.

![Figure 6.10: The Accuracy problem.](image)

In Figure 6.10, a timeline with a single itinerary is visually represented. Remember that are 45 million of the $G$ and $C$ messages coming from the Search Engine every day. The way to measure the accuracy of the cache is to count the number of times the rate was served correctly versus the number of times it was served incorrectly. The vertical jump of the line represents a price change, the $G$’s are the GDS messages, and the $C$’s are the cache messages. The number of cache messages between two subsequent GDS messages has to be counted. If these two GDS messages still are at the same line, no price change has occurred, and the first GDS and the cache messages in between can be counted as a match. If the two subsequent GDS messages are not on the same line, a price change happened, the first GDS and a part of the cache messages are counted as a mismatch. The other part is counted as a match. The reason to count part of the caches as a match is because the rate is not invalid right after it was shopped. It is only known that the rate has changed somewhere in time. Therefore, a dynamic formula is used to determine which part of the caches to include and which to exclude as matches and mismatches. Currently this is set to include the first 25% of the time difference as matches, because it is better to be pessimistic then to be optimistic. There are some additional difficulties with the aggregation of the different event properties, but those are explained in Section 6.5.
To be able to detect a pattern and produce the correct result events, a number of statements are written as part of the system. The system uses more statements than presented in this section. Those statements can be found in the Appendix. They have the task of creating the windows, preventing race conditions and cleaning up windows containing events that are corner cases.

There are three statements that perform the actual work. The first statement, Listing 20, keeps track of all the cache events (an AccuracyEvent with the property SearchRateFromInv = true). At the moment a cache event arrives the statement will try to merge it into a window. This window aggregates the information of similar itineraries served from the cache. The statement contains two cases: one when the window already contains a similar itinerary and one when it does not. If the windows already contains the itinerary (when matched) it merges the new information into the existing item by adding 1 to the NumHits property and use a custom function to add the event properties to an array. When the itinerary is not present (when not matched) it adds it.

Listing 20: The pattern to follow for a Cache event.

Another statement keeps track of the GDS events. The statement tries to find the two consecutive GDS events (every previous -> current). The two GDS events are considered similar based upon the hotel, the checkin and the length of stay. There are two cases when the matching fails. When the CleanUp event is triggered—see the Appendix, or when no match was found within x days, where x = 7 in this case. Note that this pattern does an insert into, which fires the internally created AccGdsEvent event.
insert into AccGdsEvent
SELECT previous as accEvent,
previous.SearchPrice as PrevPrice,
current.SearchPrice as CurPrice,
current.CreationDateTime as LastEventTime
FROM PATTERN [
EVERY previous = AccuracyEvent(SearchRateFromINV = false)
-> (current = AccuracyEvent(SearchRateFromINV = false,
HotelID = previous.HotelID, CheckIn = previous.CheckIn,
Los = previous.Los)
or cleanup=CleanupEvent(HotelID = previous.HotelID,
CheckIn >= previous.CheckIn, Los = previous.Los)
or timer:interval(7 days)
)]

Listing 21: The GDS pattern.

The generated internal event from the previous statement causes the statement in Listing 22 to fire. This statement merges the GDS event into the aggregating cache window. In the case the event does not exist it adds the GDS event. In the case it already exists it does not have to do anything. In both cases it means that further processing is needed, because two subsequent GDS events were found with or without cache hits in between.

The DBTriggerEvent is created with the unique properties of the GDS event. Through a number of other statements, which can be found in the Appendix, it selects the possible cache entries from the windows populated by the first statement presented in this section. Together the statement and the DBTriggerEvent form the resulting event; matching the pattern $G C^* G$, as shown in Figure 6.10.

The resulting event with the TimeArray and NumHits properties will be further processed by the post-processing—see Section 6.5.

Listing 22: The pattern to follow for a Cache event.
Performance and Memory usage

The system proved to be able to handle the 45 million messages a day while running on a 24-core server. Thanks to the usage of the [MQ](#), it is easy to see when the system is struggling, because the number of messages in the queue will increase, while the number will stay at zero if the system processes the messages fast enough. It only uses a fraction of the available CPU power and does not experience heavy fluctuations when the traffic increases, which indicates that the system is scalable. The whole processing flow is multithreaded, because any single thread processing would cause a bottleneck and would (slowly) fill the queue with messages. The Processing part of the system uses, on average, 300 threads; a number that could be increased if needed. The use of thread engines makes the number of threads configurable, so increasing them is a simple task. The system is currently running, in the form of Phase-II, in production at Priceline.com, and it has been running for a couple of months already.

The Esper part of the system had significant problems with its memory use. Using the [Java Virtual Machine (JVM)](#) optimization flags to fine tune the memory usage would not help. The problem mentioned earlier with the sockets using memory was also tackled. Thanks to the separate processes it was apparent that while the Processing part kept a stable level of memory use, the Esper part’s memory usage grew rapidly. The Esper part consumed 2-3GB of memory per hour, something that was expected due to the large window retaining events until the subsequent [GDS](#) event would arrive, but the usage did not stabilize. Trying to analyze the heapdumps was a difficult task, because of the size of the heapdumps all the analyzing tools crashed or were very slow. The solution was to make a heapdump short after the system was started when the memory usage was still relatively low. This showed that the memory mainly consisted of the HashMaps. Using the GUI, each statement was destroyed to try to find the statement causing the problems. Sadly, it was the most important one: the [GDS](#) pattern statement. After trying to further optimize the memory usage by using the smallest possible keys for the event properties, which helped a bit to reduce some memory usage, but was not a significant enough improvement to see the stabilizing happen before the system would run out of memory. Trying to find the root cause of the excessive memory usage was time consuming, because the system had to run until it would stabilize, or run out of its 50GB of memory, to see if an improvement was effective.

Due to the pressure from business—see Section [6.1](#) while continuing the investigation, a fallback plan was made and Phase-II of the project was developed, which will be explained next.

6.4 Phase-II

The purpose of Phase-II was previously given, see Section [6.1](#) and it is meant to replace Esper and perform the specific task of matching the pattern; some-
thing that the statements in Esper normally would do. The design changes and implementation needed to achieve this are explained in this section.

Again the modular structure of the initial design proved useful, because only a hook had to be placed at the place where the events normally would be sent to Esper. The hook will cause the Feeder to forward events to the Esper replacement, instead of using the socket to send them to Esper. The replacement system will produce results containing the same information as the result event that the Esper engine would have produced. These results are converted to AccuracyOutputJobs and added to the OutputEngine queue, and thus hooking back into the original system again.

![Diagram showing the design for the Esper replacement in Phase-II.](image)

The design exists consists of two packages, the Bypass package and the Simple package. The Bypass package contains classes that hook into the system to bypass Esper. Figure 6.11 shows the architecture of the Esper replacement. The classes can be divided in three parts. First, the top two classes called Bypass provide the bypass functionality described earlier. Second, the bottom part which consist of every class beneath the HotelEventStore, these are part of the state safe mechanism. The other classes are part of the primary mechanism.

The AccuracyBypass converts the HashMap events obtained from a Feeder into AccEvents. The AccEvent is a plain accuracy event without any additional knowledge. This event is given to the MainEventStore. The MainEventStore keeps a number of HotelEventStore partitions. The AccEvents are partitioned based on their HotelID. Based upon the information in the event itself combined with information in the HotelEventStore, the basic AccEvent gets upgraded. This way the GDS events will become a GdsAndCacheEvents ob-
ject and the cache events will become AccCacheEvents. The HotelEventStore keeps track of AccCacheEvent that have the same itinerary properties as the GdsAndCacheEvent. This continues until the second GDS event arrives. Then the ResultantEventConsolidator gives the result event back to the AccuracyBypass. The AccuracyBypass turns it into an AccuracyOutputJob, which is given to the OutputEngine.

The system will downgrade an event to a RemovableEvent when it exits the window of the HotelEventStore. This will cause the event to be removed at the next execution of the PeriodicCleanupProcessor.

The state safe mechanism exists out of a persistence manage that regulates the WriterAgents to store each of the HotelEventStores. These are stored compressed into a file by the FilePersistanceAgent. The file location alternates between each write; in case anything goes wrong the other file will contain the previous state. During a restart the system will use the ReaderAgent to load the state.

### 6.5 Post-Processing

The aggregated result events will be further post-processed. For the accuracy measurement the result has to be split into the matching and mismatching part. The Regression Measurement only uses AccuracyEvents that are GDS events. Both processes are explained next.

#### Accuracy

The AccuracyOutputJob performs a couple of tasks, which depend on the type of accuracy result. The simplest case is when a GDS event is not followed by another event within the time of the window. This type of GDS event does not need any further processing, because it is not possible to say anything about it. The event will be given a zero at the GdsHits and HitCount properties, indicating that it is an event that left the window without a consecutive GDS.

The GDS events that have another consecutive GDS events can result from one of two cases, it can either be a matching rate or a mismatching rate. For the matched rate the TimeArray property has to be split into days. Then, for each day, the TimeArray has to be split and grouped by AffiliateId. This will result in a number of different rows in the result table.

For example, if a GDS event from a day ago with the convenient AffiliateId ‘yesterday’ is matched today with another GDS event today, and in the meanwhile there was a cache shop yesterday with the AffiliateId ‘cacheyesterday’ and a cache shop with the same AffiliateId as today’s GDS event, both ‘today’. This will result in three rows. First the split on day, causing the two shops of today with the same AffiliateId to be aggregated as one row, with a total HitCount of 2 and GdsHit set to 1, because it contained a GDS hit and the AffiliateId were equal. Yesterday’s shops will cause two rows. One for the GDS with GdsHit set to 1 and a HitCount of 1. The other is the cache shop, because it has a different AffiliateId. This row has a GdsHit set to 0 and a HitCount of 1.
In the case the prices are mismatched the TimeArray first has to be split into the part that is still counted as a match and the part that is not a match. The requirement for the prices to be a match can be configured, allowing a margin of error to be set. If the margin is zero, the rates have to match exactly. In the same way the time that is still counted as a match can be set as a percentage, or the method doing the calculation can be overwritten. The split function will check the creation date time for each cache entry and determines whether it has to be flagged as a match or as a mismatch.

The overall mechanism to do all the splitting and marking of the TimeArray is quite complex. The mechanism could be simplified when the POJO event representation was used in the Cache22 Esper DSL version of the system—see Chapter 7, so please consult the code of that version for more details when needed.

**Regression**

The Regression post-processing is not really part of the Measuring Stage, but already a part of the Prediction Stage. It only uses GDS events, because it needs to work with non-cached information. An important source of information is the History Table.

The History Table keeps track of whether there was at least one GDS shop for a certain itinerary in each hour and at what point the rate changed. The History Table’s functionality to keep track of whether a GDS hit happened in an hour was added for this project.

The idea behind the Regression is to use the information in the History Table to create a list with a total sum of matches and mismatch for each combination of specific itinerary combination. For simplicity, let’s say the AP and LOS properties are chosen as the combination. For each of these combinations a list of 72 hours is kept, with a column with the total matches and a column with the total mismatches. The totals are calculated using the information from the History Table.

Figure 6.12 gives a visual representation of the process for a single GDS hit. Each time there is a GDS hit for the combination, let’s say AP = 1 and LOS = 1, the history for that itinerary will be consulted. Let’s assume that the rate changed 50 hours ago and that there were other GDS hits at 100, 71, 60, 45, 30, 9 and 3. The 100 is outside of the 72 hours and is ignored, the 71, 60 and the rate change itself at 50 will cause one mismatch to be added for each row, where the 45, 30, 9 and 3 will add one to the total of matches in the row for that hour.

Over time, the customer traffic will create a good match versus mismatch distribution for all 72 hours of each AP and LOS combination. It is possible to make the combination more granular by adding more properties, e.g. the star rating of the hotel. A regression formula can be made using the 72 hours for each combination, with the hours on the horizontal axis and the percentage on the vertical axis. The formula can be used, to an extent, to guarantee the accuracy percentage, e.g., a certain affiliate wants to be served with rates that are at least 85% accurate, then from the formula it shows that any rate that is
in the cache for no longer than 5 hours is at least 85% accurate, so the rates above 5 hours will be served from the GDS and the others from the cache.

### 6.6 Testing

The Cache22 Esper system has been tested in three different ways, in addition to running the system in production for a long time. The system has been tested manually, in the quality assurance environment, and it has been semi-automatically tested.

The manual testing used the FileFeeder. Several files acted as testcases: feeding events in a specific order. The events were ordered in such a way that they would cover each corner case for the matching statements. A number of files purposefully contained wrong data to test the robustness of the system. As a second stage the system was tested by a member from the infrastructure team who was not involved in the development of the system. After spending...
a significant amount of hours, only a few flaws were detected, which were fixed without much trouble.

The quality assurance environment is a replica of the production environment, but without the customer traffic and therefore less generated data is available. The quality assurance environment does contain the latest releases of every production system and new releases of systems currently under test. The benefit of this is that it contains the full chain of systems, making it possible to run the search and booking processes. This can be done using the QA version of the site, instead of the production version. The Cache22 Esper system was placed in the QA environment to see if it could process the actual format of the events that would result from the search and booking processes and if it could complete the measurements. The databases are available too, so the writing of the results could be tested as well. The system could be monitored and debugged using the remote debugging functionality provided by the JVM. Together with the QA website to initiate a process, the flow of data could be tracked and the final result in the tables could be manually verified. The remote debugging was useful if something would appear to be wrong with the final result.

The manual and QA stage for testing the system are useful, but not enough. A part of the system needs information that is only available in the production system, like the History Table, because the History generated in QA by other developers testing there system would be sparse and only provide applicable information in a few cases. Another rather obvious part that could not be tested in QA is how the system would be able to handle the load. Thanks to the use of the MQ and dynamic config on the production systems, the flow of messages to MQ could be cut of immediately in the case of an error and prevent an error in the Cache22 Esper system to have an impact on the production systems. The Cache22 system is load-balanced across four different servers, and therefore the Cache22 Esper system could first be tested with a fourth of the traffic, by only forwarding the traffic from one of the servers. When the system held, the traffic was increased by another fourth, forwarding half of the total traffic. The system did hold, so it was given full traffic.

The problem with full or even a fourth of the traffic is that it becomes impossible to manually verify the results in the tables. Let alone actively debug the system with the remote debugger. The threads allow many different searches and bookings to be processes at the same time, which makes it impossible to make sense of any of the breakpoints that are hit.

This is where semi-automated verification comes into play. The automated part is the custom written program used to verify the results. The semi part comes from the fact that the program cannot decide on every case, so it asks to manually verify those. The program is not fully automated because there is a significant tradeoff between the additional time needed to make the verifier be able to process the difficult cases versus the time it takes to do the manual verification. It also prevents the verification program from becoming too complex, and thus also prevents uncertainty about whether the verification is doing its job correctly.

The verificator works by reading the results from the table and decompos-
ing the aggregated values. Then it looks into the Cache22 table storing the raw messages, to see if it can find the decomposed information. If it is unable to, or finds too much information then this implies that the Cache22 Esper system did not perform the aggregations correctly. At that moment the verifier will log the specific single search or booking process and these can be manually verified. The error often is caused by a complex situation that is not covered by the verificator; remember the tradeoff mentioned in the previous paragraph. The verificator did do a correct job verifying, because sometimes it pointed out some actual flaws. These flaws were caused by the development process. The Cache22 Esper system is state safe, so when it is stopped it saves the search and booking processes it was currently tracking. During the debugging and fixing of errors found, or overnight and in the weekends, the traffic would be cut off to not overflow the MQ. At the moment the system was resumed and the traffic was enabled again, the messages between the loaded state and the messages currently on the MQ were of course not tracked, causing errors in the results, which would be picked up by the verificator. The combination of the facts that the verificator was able to detect errors correctly, and when it detected errors they were false negatives, and that all normal cases would be marked as correct, showed that the system was running in a correct manner.

6.7 Results

Although Esper is not part of the system in its form after Phase-II, the system itself performs well. The EsperHA edition showed promising improvements, and therefore is implemented as part of the DSL version—see Chapter 7. The system showed a minor increase in accuracy in regard to the previous measuring mechanism that only used GDS shops. This was good news for business. That the percentages did not differ extremely can be explained. The number of matches versus mismatches primarily increases in their count, because the cache shops are added to the count. The proportions, however, roughly stayed the same, explaining why there was not a significant drop or increase in accuracy. This was good news, because it meant that the numbers they had been giving were not bogus, they were not even that far off. In the future this could have caused a bigger difference however, because the number of cache hits will increase significantly.

The major improvement was made using the information provided by the Regression post-processing. The information improved the accuracy of the cache by approximately 7 percent – from 83% to over 90% accuracy. The information was used once to adjust the expiry timing mechanism of the Inventory Cache and this is what caused the significant increase. Remember that adjusting this formula even once is extremely significant, because it directly impacts how the rates are gathered from the GDS and thus how they are shown to the customers. If something were to go wrong this would have a direct effect on the sales, and due to the number of sales a day this could cost a significant amount of money. The increase was so significant that business decided that other problems now became of a higher priority, and thus actually implying
that the problem has been partially solved, at least solved well enough for now. The formulas produced in the post-processing stage are valuable for the services offered to affiliates. The affiliates can now choose, to a certain extent, the quality of the rates they receive from Priceline.com. This will save Priceline.com additional GDS shops, because the company can serve the rates from the cache while still complying with the agreement with the affiliate.

The system itself is—at the moment of writing—still running and has been running fine for five months already, achieving a more accurate measuring of the cache’s performance. The resulting measurements are compressed, because the cache hits are aggregated with the GDS hits, without any loss of information compared to the original messages that were saved as plain rows in a table. The system proved versatile and useful during the course of the development, because the system could be tweaked to the changing wishes of business, due to the configurability of the system. With the DSL version the system is also expected to be serviceable for other problems—see Chapter 7.
7.1 Motivation

The Cache22 Esper system is complex and a lot of effort is needed to adapt it for other applications. The Cache22 EsperHA DSL reduces this effort by introducing a Domain Specific Language (DSL) that can be used to specify the components needed for the other application. The DSL will generate code and that will cover most of the effort needed to adjust the system. The Cache22 EsperHA DSL is a system that will be easy to apply to other applications. The Accuracy Measurement is used as a showcase and implemented using the DSL.

The chapter starts with a motivation for the use of a DSL in Section 7.1. Section 7.2 shows the design of the generated code and explains how the code is generated from the syntax. The design has additional classes, not just the core classes, which are part of the showcase to help explain how to apply the system. This is followed by the language definition in Section 7.3. How the generated system is implemented is described in Section 7.4. Finally, the project is evaluated in Section 7.5.

7.1 Motivation

The Cache22 Esper system is complex; it has many classes and performs a number of different tasks in a couple of stages. The system’s functionality can be used for other applications, but a large portion of the code has to be adapted to handle these other situations. It has to be modified to allow the different data structures, while the overall functionality does not change. The adaptations made to those classes are data structural changes only; those classes still perform the same functional tasks. Therefore, a core library with additional case specific code generated from a Domain Specific Language (DSL)
served as a way to generalize the Cache22 Esper system and makes applying it to other applications a simpler task.

In addition, after the second phase of the Cache22 Esper system, a possible solution for the memory problems experienced would be to use the EsperHA variant together with the POJO representation for the events. The EsperHA should be able to better cope with the memory usage and using the POJO representation would remove the extra overhead used for a HashMap's internal structure. This meant that the system, as in Phase-I, had to be refactored: to use POJO instead of HashMap's and replace the engine core with the EsperHA variant. It was a perfect case that confirmed the statements made in the previous paragraph: the refactoring would not have to involve any functional changes, only changes to be able to use POJO had to be made.

The problem analysis showed that to regulate the way improvements were made—e.g., to handle or correct corner cases—rules could be applied. After the decision was made to apply Esper to solve the problems encountered in the Measurement Stage, it seemed a good platform to serve as a basis for the Regulation Stage. The rules could be written and run with Esper, using EPL statements. The initial idea was to create a DSL to translate rules that could be written by people from business in natural language, which could then be translated into EPL statements. Some time into the project's first phase it became apparent that it would not be feasible to create such a DSL within the scope of the project, due to the expressiveness and the complexity of the EPL itself, let alone to translate it into a natural language. On the other hand, Esper proved to be a powerful tool and ideas to where the Cache22 Esper system—it still had to be finished at that time—could be applied were coming from business. Applying the system as part of the hardware monitoring services was one of the other, more concrete applications. Esper’s ability to detect changes in the stream of notifications from different monitoring systems could provide the people operating them with a better way to assess the relative importance of each event. Therefore, the need to generalize the system for other applications became even stronger. The idea for a DSL and the need to generalize, as part of the thesis, were already there, thus the use of a DSL as the solution to generalize the system led to creating the Cache22 Esper DSL. The Cache22 Esper DSL could later be used to build the monitoring system.

The complex but functionality wise stable structure of the Cache22 Esper system could be adjusted more easily using a DSL. The DSL will generate the changes needed to adapt it for other applications. The refactoring needed to use EsperHA with the POJO representation could be done within the same project. Additionally, the benefit of having a future solution for the monitoring services was another motivation for the Cache22 Esper DSL system.

### 7.2 Design

This part of the project can be seen as the third phase of the Cache22 Esper project. It can be considered as another phase of the Cache22 Esper project because the goal and overall structure remained the same. This phase, like the
other phases, provides a solution for the Measurement Stage and processes the streams in a similar way. The difference is that parts of the code are now generated by a [DSL], the core is the EsperHA variant, and the event representation is changed to [POJO]. The design is adjusted to reflect these changes.

**Figure 7.1:** High-level component overview of the Cache22 EsperHA DSL system.

The [DSL] project consists—just like in Phase-I—out of two separate processes: one for the Esper Core and one for the Processing part. Figure 7.1 gives a high-level overview of the different components. There are five high-level components, two external components and three internal components. External in this context means that these components are not part of the source of the project. One of the two external components provides the raw data, while the other external component takes care of storing the result data. The internal components are running in two different processes: one is running the Esper core, the other runs the pre- and post-processing. Both processes use the third internal component, which is a library containing common code. The Esper Core process possesses links to both the Library and the Processing component. The Esper Core needs a link to the Processing component because it contains the definition of the [POJO] events. The link was not needed during the previous phases, because the HashMap representation was used. The HashMap is part of Java itself, and its properties were communicated by defining the events at runtime using the [JMX] interface.

The Library component contains common code unaffected by the generated code. For example, classes that provide the communication with the Esper Core, abstract classes providing common functionality, the queue reader, feeders, the output engine, etc. The Library is not used by the Esper Core, except for the event definition classes and is the reusable part of the Processing component for each of the different applications of the Cache22 Esper DSL system. The Processing component exists of two parts and uses the Library. One part is the generated code based upon the code provided in the [DSL] file.
and the other part consists of classes that are generated only once and provide customizability for the developer. A number of methods need to be manually implemented because they are dependent on the specific application the system is generated for.

In Figure 7.1, the processing flow runs, again, from left to right starting with an “external” system providing the raw data by a message queue (A). The raw data is then preprocessed and sent to the Esper Core. The preprocessing part will read (1) and transform (2) the messages from the queue(s) into POJO events. The events are fed (3) to Esper by communicating through sockets (I). Using the JMX interface (II) the—in the Processing component (4) implemented—EPL statements will be registered at the Esper Core. These statements will turn the streams into result events in the same manner as before, but now using additional functionality of the EsperHA variant. After processing the events, the result events are forwarded (5) into queues (III). The messages are picked up by subscribers (6) that are part of the Processing process. The post-processing can transform (7) the result events into a format that can be outputted (8) to e.g., the database queue (B). The DSL has the syntax to describe each component and data structure, such as events, of the processing flow. Additionally, the DSL system provides a standard way to describe how the post-processed results have to be written to Priceline’s processing queue for saving data to their database. Together with the library code, the description written in the DSL syntax will generate the major part of the code for the system. A detailed description of how the process is implemented can be found in Section 7.4.

EsperHA provides the functionality to make the statements state-safe; it stores the events and processing status of the statement in a way that—in the case of a crash—the events and statement state can be recovered. EsperHA provides a mechanism to better cope with the use of memory, when the memory usage passes certain limits it will store the status and those events, which it seems least likely to match in the near future, to disk. One of the major design decisions was whether to use the EsperHA version or keep the EsperEE version: the memory and state-safe functionality was the reason why EsperHA was chosen. The memory saving functionality was interesting because of the memory problems encountered in Phase-I, which lead to abandoning Esper in Phase-II. Having the state safe information was an easy choice, the pattern used for measuring the cache performance in Phase-I could cause expression to spread over multiple days and a crash without a state-safe engine would cause the loss of days of measurements and significantly skew the data towards the measurements that can be detected in a small time period. Additionally, the assumption that another application of the generalized system would need a state-safe engine, together with the relative ease to replace the EsperEE engine with the EsperHA engine in the core led to the decision to go for the EsperHA version.

The design decision to use the Transformers has several reasons, and the alternative would have been to directly use the messages from the queue(s) as events. The queue itself could have multiple message types, which might need to be combined into one event or it could contain a single type, which
could be split into multiple events. This last case actually takes place in the Cache22 Esper DSL: the Analyzed Stats data object from the queue is split into a GdsEvent and an CacheEvent. The systems that generate the messages are production systems and are developed by different teams most of the time, thus control is limited. Due to this limited control and flexibility it is hard to have the systems generate the exact events needed or change the events if, later on, different event formats are required. Additional, the system could provide the messages for multiple systems, so it might not even be possible to change the format of the messages.

Of course Esper is able to perform these modifications and transformations of events, but this would require additional EPLs, memory and processing power. The functional EPLs are already difficult enough to understand, so by shifting the transformation and preliminary filtering to the Transformers, the EPLs are minimized to the essential ones and the Esper Engine is kept clean and clear. By shifting the transformations and filtering, the processing power needed increases at the Processing process, but this is not significant and the process could be placed on a different machine if needed. This issue is not significant, because the process has to read the messages from the queue and forward them to the Esper Core anyway. The Processing process therefore already contains the message object in memory and only needs to perform the additional merging, splitting and filtering logic.

The queued decentralized database system that Priceline uses was added as part of the DSL. Instead of simply outputting the resulting event to another queue, the post processing phase supports the transformation of result events to the right message format for the database queues. The choice to add this functionality has predominately practical reasons. The results of the Cache22 Esper system had to go to two different tables and for other applications within Priceline the functionality would have been needed. That the system completes a full cycle—from reading raw data till storing the results—is another reason, where storing the results as rows of a table was the first choice to show that the system is able to perform the whole cycle. Additional ways to transform the resulting events to formats ready to be stored can be added. This was not done due to the fact that the tables were sufficient and time was limited.

### 7.3 Language Definition

The language is defined in a way that reflects the elements from the component diagram, while keeping a set of guidelines in mind [15]. The definition contains entities, actions and controllers. The definition currently only contains one type of controller and that is the notion of a Processor (4). A Processor definition is started with the keyword processor followed by an unique name and it controls the EPL statements that have to be registered at the Esper Core and sets which subscriber (6) to use for the post-processing. The syntax and a example are shown in Listing 23.

The syntax of the processor refers to an entity: the statement. An entity can be seen as a passive thing; the input, queue, event, output, and sub-
7.3 Language Definition

Processor:

'processor' name = ID '{
    (statements += Statement)+
    '}'

 Process 

Listing 23: The Processor syntax with an example.

subscriber are the entities in the Esper [DSL] The language for the entity will describe the object and its properties, but because they are passive no actions have to be declared. The entity statement is defined in Listing [24]

Statement:

'statement':
    'name': name = ID ';'
    ('subscribe':
        subscribe = [Subscriber] ';')?
    ('listen':
        listen = [Listener] ';')?


Listing 24: The Statement syntax with an example.

The statement entity can only exist within the processor definition, that is the reason why its example is wrapped with the processor. In the definition there are brackets around the Subscriber. This means that it is a reference, the syntax checker will expect an ID of a previously defined subscriber and not the definition of a subscriber. The reason to use a reference is that the subscriber can be reused across multiple statement definitions, while the statement definition itself is an integral part of the processor definition and can not exist on its own. The definition for the other entities is similar and the whole syntax can be found in the Appendix.

The definition type that is left is the action. The action defines things that perform an action: the two transforms. There are two types of transform, one will define the transformation of an input to an event and the other an event to an output. Thus the first transforms messages coming from the queue into events, and the other transforms result events into output messages that can be passed onto the database queue. An optional condition can be applied to restrict certain transformations. The transformer definitions are similar; the input to event version follows next:

TransformIn:

'transform input' from = [Input] 'to event' event = [Event] '{
    ('condition':
        condition = Expression)';
}

Listing 25: The Transformer syntax.
transform input AnalyzedStatsData to event CacheEvent {
  condition:
    searchRateFromINV == true
  ;
  affiliateID -> affiliateID;
  checkIn -> checkIn;
  los -> lengthOfStay;
  hotelID -> hotelID;
  creationDateTime -> startDateTime;
}

Listing 26: The Transformer example.

To summarize, the [DSL] contains five entities, two actions and one controller. The entities are:

**queue**  Defines a queue where to read messages from. It currently has one attribute and that is the name of the section in the configuration where the settings for the queue can be found.

**input**  Defines an object that can be retrieved from a queue. It requires an ID, specifies the actual JVM object name, and has a list of attributes that can be used in other definitions later on. The queue where the object originates from has to be specified. An optional filter expression can be added to the definition to only select specific messages to be processed.

**event**  Defines the event that is send to Esper or the result event coming from Esper. It has an ID and a list of attributes. The attributes follow the usual structure of an ID and a JvmTypeReference, which is any valid type.

**subscriber**  Defines a reusable subscriber for statements. Currently only an ID is needed, so it can be referenced in the processor. The subscriber simply gets the events and turns them into jobs for the output engine.

**output**  Defines a way to output result events. Currently there is only one output type and that is the table variant. Therefore a list of columns with their type need to be specified. In addition to the events, tables might not accept null values, so a flag can be set whether a column is allowed to be left null.

The two actions, transform from input to event and from event to output, are used to obtain the right data formats during the different phases of the process flow. The controller, the processor that was previously described, controls how all the parts work together.

The modeling framework used to create the [DSL] is Eclipse Modeling Framework (EMF). The choice to use the Eclipse Modeling Framework (EMF) was partially based on a comparative survey [1] of modeling tools. The fact that
the DSL could be built using Eclipse, an already familiar editor—especially for Priceline.com developers—is a significant benefit. In addition, EMF has the possibility to create a standalone plugin for Eclipse. Therefore, installing the DSL is a simple task. The plugin can be installed just like any other Eclipse plugin and can be updated just as easily. The EsperHA DSL generates Java code that is part of a Java project. The Eclipse IDE is commonly used for development of Java projects, and having the DSL as a part of the IDE is more convenient compared to a separate application that is needed to generate the Java classes.

By using EMF, the Esper DSL experience is completely integrated into the Eclipse IDE. After installing the plugin, the developer only needs to add a file with the extension supported by the DSL—for the Esper DSL it is .edsl. The code using the Esper DSL syntax can be written inside this file with the support of all of the usual IDE features such as: syntax checking, syntax highlighting, auto-completion and suggestions. The code will be generated at the moment the file is saved.

EMF has many benefits for a language developer. It supports the development of the IDE features and provides templating of the code that has to be generated. The most important feature for the Esper DSL was the support for Java types, because all attributes specify types and therefore the support to be able to check and generate them was vital.

### 7.4 Implementation

The implementation of a system with generated code is not trivial. When generating code there are a few best practices [22] that must be adhered to. These practices include keeping generated and non-generated code in separate files, generated files must not be checked-in to a source control system; the file containing the code that is used to generate the code will be checked-in instead. Never modify generated code, because the modifications will be overwritten when the code is regenerated and the generated code will not be checked-in to source control. In the overall architecture it should be clear which parts are generated and which are not, this can be done by using suitable design approaches to join the generated and non-generated code. Keep in mind that the developer has to work with the generated code, for example to actually verify the generator or during debugging.

The decision was taken to use the Generation Gap Pattern [27], because it maintained flexibility and customization for the developer using the DSL while at the same time generating source that provides a significant level of abstraction and simplification for using the Cache22 Esper system in other applications. The Generation Gap pattern adheres to the best practices and was the best fit for the Esper DSL project. The Generation Gap Pattern will be explained next, after which the primary aspects of the Cache22 EsperHA DSL implementation will be described.
Generation Gap Pattern

The Generation Gap Pattern helps when modifications or extensions have to be made to generated code. The modifications only have to be made once no matter how often the code is regenerated. The Generation Gap Pattern overcomes the problem of overwriting changes made to modified code though inheritance. The generated code is encapsulated into a class which is then split into two parts: one encapsulates the generated code and the other contains the modified code [27].

Figure 7.2 shows a class diagram of the pattern. The CoreClass is the generated class which is abstract, is never modified by hand, and is overwritten with each regeneration. The CoreSubClass is the trivial subclass of CoreClass which implements the modifications or extensions that are kept at each regeneration. The key part is the Client class: it only calls the methods using the CoreSubclass.

The Generation Gap Pattern does not describe where to store the generated files itself, therefore the Esper [DSL] uses a mechanism of conditional generation—based on the two gen-folders variant [2]. Figure 7.3 shows the folders and the type of class files residing in them: the Abstract classes are the Core classes and the Concrete are the CoreSubclasses from the Generation Gap Structure. The CoreClasses additionally inherit from other manually written classes. These manual written classes contain code that remains the same and is part of a common framework. These are in the src folder with the two gen-folder variant, in the case of Esper [DSL] it is the Library component stored as another Java project. Instead of using two different folders for the generated once (src-gencond) and manual (src-man) as in the two gen-folders variant, the Esper [DSL] uses the src folder and only generates the concrete classes if they do not exist.
7.4 Implementation

Esper Core

The Esper Core did not change much from its version in Phase-II. There is no code generated for the Esper Core, because the core is kept as small as possible and stays the same for each application. It contains methods for the configuration, starting, and stopping of Esper. A few additional configuration settings that are specific to the HA version were added. The new functionality to make statements resilient, and thus having their state saved, added some difficulties, which were manifested when the Processing part wanted to register the statements again, and after a restart it was troublesome to make sure that the subscriptions on the reloaded statements were properly configured again. After figuring out how the configuration exception system worked, which was not an easy task due to the lack of documentation, the exceptions were logged and provided more insight and the issues could be resolved.

DSL Library

The DSL Library contains the common framework for the Cache22 EsperHA DSL system. The code does not contain anything specific for the Cache22 part of the system; that code is part of the DSL processing and is partly generated as a showcase for the DSL. In Figure 6.1 the classes are positioned in a way that reflects the component structure of the original Cache22 Esper system. The library contains the code that any application using the Esper based processing system needs.

The diagram shows two utility classes in the top; the Constants class contains global constants read from the config and the TimeUtils class has static methods to perform time related calculations. Proceeding with the column of classes on the left side of the diagram, these are the base classes for the Queue...
Readers. These classes contain the common code to set up and run a queue reader.

The next column of classes contains the essentials for the Feeders. It has a BaseEvent which was created by the AbstractFeeder using a new thing: the IEventTransformer. Transformers are separated classes that Transform the input of a Feeder to Events which can be send to Esper. These are separated, in comparison to the Cache22 Esper, for clarity and generation purposes, because Transformers are an active class while events are passive classes. The library is minimized and only contains those items that are really necessary and used, and that is the reason why the TableFeeder has been left out of the library.

The right side of the diagram contains the post-processing classes. The Bootstrap class and the Processor class with the EsperJMX helper class are familiar. The Listener and Subscriber also contain common code that is needed for every application of the system. The Dispatcher and OutputEngine combination contained code that was generic from the start, Phase-I, and is therefore logically part of the framework. To finish with the Table class, which contains the same common code as earlier: code to connect to the database and create messages in the format needed.

**DSL Processing**

The DSL processing code consists of two parts: the generated part and the generated-once part. The generated-once part will be referred to as the manual part, because most of the generated-once classes are just stubs, or contain
a minimal amount of setup code. Remember that the classes are in their functionality close to those of the Cache22 Esper system, that is why the following paragraphs will explain the generation behavior and not the functional behavior.

**Readers and Transformers** The DSL generates a queue reader per specified queue in the syntax. To be able to transform an object from the queue into an event transformers are generated. Figure 6.3 shows the hierarchy. The top two rows of classes are part of the library and not generated, they are there to illustrate how the Readers and Transformers are connected to the Feeders. The lower three rows are ordered according to the Generation Gap Pattern, this will also be the case in the other Figures that follow. The top of the three rows are the classes from the framework, thus part of the Library. The second row contains the generated classes; an AnalyzedStatsQueueReader and the matching transformer are generated based on the syntax in the .edsl file. The BaseReaderEngine is also generated because it contains additional code to add the newly generated AnalyzedStatsQueueReader into the load readers sequence. The bottom row consists of the generated once or manual classes. The ReaderEngine contains some default setup and startup code, the Reader class is a stub, and the AnalyzedStatsReader requires transform methods to be implemented for the property types that do not match and therefore can not be automatically transformed. Remember that the Generation Gap Pattern allows the flexibility to override or extend methods, so if a transformation of a property needs some additional attention, aside that from the automatic variant in the generated method, the method can be manually be extended or overridden to do so.

![Diagram of Readers and Transformers hierarchy](image)

**Events** This time the events are represented as POJOs so their hierarchy is simplified, see Figure 7.6. The top row contains the BaseEvent from the library, providing basic functionality and making it possible to use Events within the framework. It is extended by a generated-once Event, which is a stub. The
class makes it possible to make system-wide additions to every Event. The generated events are determined by the DSL syntax. In the Cache22 EsperHA DSL version there are two kinds of events. The generated version contains getters and setters to all the attributes specified as part of the event definition in the syntax and some additional functions needed for the serialization and for EsperHA. In this specific case the GdsEvent contains a lot of code to aid the post-processing.

Figure 7.6: The hierarchy of Events.

**Bootstrap and Processors** The start of the system with the Bootstrap class and the registering of statements and subscribers with a Processor did not change, but their hierarchy did—see Figure 7.7. The top layer still performs the same task and is part of the framework. The middle layer is generated and the BaseBootstrap will register each generated Processor. The BaseProcessor will add the statement to Esper using the functionality of its framework parent class and create abstract methods for each statement specified in the syntax. The actual EPLs have to be implemented in the Processor; in the create methods which are forced to be implemented by the BaseProcessors abstract methods.

**Subscribers** The hierarchy for the Subscribers is shown in Figure 7.8. The syntax of a single subscriber in the DSL leads to the generation of a subscriber and the corresponding OutputJob. The AbstractSubscriber, which is part of the library, has an abstract method asking to create the jobs and contains the methods needed for receiving the events from the Dispatcher. The Dispatcher and OutputEngine components are skipped in this description because they are only part of the library and do not have any generation behavior. The generated BaseSubscribers implements methods that turn the events into the corresponding OutputJobs. The OutputJobs have an execute method that has to be implemented manually to decide what happens with the events contained in the job during the execution of the job.
The AccuracySubscriber generated once is just a stub in this specific case, but the RegressionSubscriber overwrites the default behavior of creating jobs and sends the events to the RegressionConsolidator instead.

The Table Output is uses a Table and a OutputTransformer—see Figure 7.9. The AbstractTable is from the framework and provides the same functionality as before. For each specified output in the syntax a BaseOutputTable is generated. This BaseOutputTable contains the columns specified in the syntax and knows how to create the messages in the format needed for the MQ that writes to the databases.

The manual AccuracyOutputTable adds a default value for columns that are used for future use, and RegressionOutputTable adds a creation date time.

The BaseOutputTransformers are generated based on the transform keywords in the syntax, more specifically the from event to output variant. The corresponding table will use the transform method of this transformer. The transformer will switch to the right submethod based on the event type it has to transform. Then it performs the transformation based on the linked properties and table columns specified in the DSL syntax. In the same way
as the transformer during the input phase, it requires methods to be implemented for the datatypes it can not automatically match and transform.

![Diagram of Java Classes]

Figure 7.9: The hierarchy of the Output Transformers and Table Output.

This completes the whole measurement cycle, again, but the system is not specific to the measurement of accuracy anymore. It can be used for any application that has a queue that provides messages. The [DSL] will speedup and aid the development of such an application, because it can for a large part be generated from the specified [DSL] syntax.

### 7.5 Evaluation

The decision to create a [DSL] for the Cache22 Esper version turned out to be solid. The [DSL] is useful for creating similar applications; of which the Cache22 EsperHA system was an example. The syntax can be used to specify the needed components, and makes it unnecessary to code everything by hand, which saves a lot of time and possible errors. Changes normally would have to be made in many places, because a single change would have an impact on the code in many places. Adapting the code is simplified by the [DSL] because the generated part of the source is automatically regenerated when changes are made, they propagate to all the components impacted by the change.

The decision to use Eclipse Modeling Framework (EMF) leads to a steep learning curve, but this is partly inherent to [DSL] and any new framework. The decision did have a number of positive outcomes. First, the framework supported the native Eclipse IDE for the development of the [DSL] as well as for the [DSL] editor created, and the language itself was able to match any language structures that already existed in Java. Second, it allowed for the cre-
7.5 Evaluation

...ation of an Eclipse plugin. This plugin was created and easily installed using Eclipse’s update manager at another developer’s Eclipse installation. This provided the developer with the functionality of the DSL right away; it recognized that a new file with the correct extension was created and the syntax checker became active. The fact that the src-gen folder has to be manually added to the project containing the file before any source would be generated is a small inconvenience.

The syntax itself was evaluated by having the developer use it for another application. After needing some time to grasp the concept of a DSL and explaining the specifics of the syntax, he was able to create and generate the basic code needed for this other application. There was no priority and, for that matter, time to complete all the other administrative tasks needed to actually try and run the created product. This would need the creation of a queue and the forwarding mechanism would need to be coded into another production system. On top of that, new tables for the results would need to be created, and the system would have to be properly tested in the QA environment.

The EsperHA edition did not provide a solution for the memory problems. The mechanism that would store events to disk, in case of reaching the limits of the memory, did not help. The DSL did provide an easy way to test many different statements and the actual culprit that caused the memory usage problem was identified. The followed-by statement in combination with the every statement without the use of brackets (every A -> A), caused the system to crash under high load. Using the GUI to see the Esper engine’s internal timer showed that it stopped running normally with this specific statement in a matter of minutes, meaning that the system could not process the messages fast enough. With any other statement the internal system clock did run normally, and at the same pace as the computer clock. Therefore, the Esper system can still be used for many other applications, but not for the specific needs of the Accuracy Measurement.
Conclusion and Future Work

8 The Sampling System

8.1 Mechanism

8.2 Sampling Technique

8.3 Time-series

8.4 Design and implementation

9 Conclusion and Future Work

9.1 Conclusion

9.2 Future Work
The problem analysis in Section 5.2 defines several stages. The analysis shows that solving the problems for the Measurement Stage already fell within the scope of a Master’s project. In the time left at Priceline.com, a beginning was made to solve other stages of the Enhancement Cycle. Each solution for a stage could be a Master’s Project on its own and ideas for possible solutions will be presented in Future Work—Section 9.2. But one of the solutions that did surpass the stage of just being an idea is the Sampling System and is therefore explained in this chapter. However, note that the system still belongs to the Future Work.

Using sampling is the next step in the Enhancement Cycle. The Sampling system is a prerequisite for being able to employ the Improvement Stage effectively. Therefore, it can be seen as a hybrid step between the Measurement and Improvement stages. The accuracy and coverage are metrics of the cache. These metrics convey how closely the cache resembles the real-life situation. The real-life situation will be approximated by applying sampling and time-series analysis. Simply measuring differences in the cache’s metrics between when the improvement is turned off, and when it is turned on will not give an independent result. There are numerous reasons for this, where measuring the overall performance difference, instead of the targeted itineraries is among the evident ones, because other non-targeted itineraries could be the cause of the improved metric. Targeted itineraries are the ones that will be affected by the improvement.

A more unbiased check can be performed by iteratively comparing the change in the difference between the baseline and improved situation. The Sampling Engine running in parallel with the normal cache system will give the baseline situation, which can be compared on an itinerary by itinerary basis to the improved situation. In the improved situation again the sam-
plling and the cache system are running, but now the cache is running with the improvement.

This chapter will discuss the two different situations in more depth and will describe the sampling mechanism in detail.

8.1 Mechanism

The Sampling system is the next step in the Enhancement Cycle. It is a prerequisite for the Improvement Stage and makes use of the Measurement Stage. Therefore, this step can be seen as an hybrid step between the Measurement and Improvement stages. The step is a prerequisite for the improvement angle because it allows for a platform that can objectively measure the effectiveness of the improvements made in the next step.

The concept is that there are two situations that can be compared. When testing the improvement for the first time, comparing these two situations will give an initial difference between how the system operates without the improvement and how it performs with the improvement. The situation without the improvement is the baseline situation (Figure 8.1) and the other one is called the improved situation (Figure 8.2). Gathering the baseline and improved situations for a specific robot is defined as a trial, based on the notion of a trial run, and is not called a test because that might cause confusion with a functional software test—like an unit test. The two situations will be explained next.

The Situations

For both situations a selection of itineraries is made that will be representative for the whole target population. The target population consists of all the itineraries potentially affected by the improvement. Most of the times an improvement is carried out as a robot, so from now the term robot will be solely used, because it can be used more distinctly to avoid confusion. Generally a robot has a specific set of rates to search for when it is triggered by a certain event. The selected itineraries are now considered as the sample for the robot. The main goal of every robot is to find rate changes in the shortest time after they occurred. Remember that there is no way of knowing the exact moment when future rate changes will occur, because these are made independently by numerous property managers of all the different hotels. Also, already occurred changes are generally unknown, because the underlying systems—which are not property of Priceline—do not provide this information. Section 8.2 explains how the representative selection of itineraries is made for a trial.

According to a schedule that is configured as part of the trial, each of these itineraries have to be tracked. The schedule is made by the developer who wants to trial a certain robot. Tracking the itineraries comes down to registering all rate changes as closely as possible after the moment they happen, so the real-life situation for these itineraries is approximated. A simple way to do this would be to ask the underlying systems for the rate every (milli)second, which is impossible because the underlying system’s responses are not quick
enough and the load cannot be transferred to them—see Section 5.2. If this was possible, there would be no reason to have a cache in the first place.

The tracking has to be done efficiently, because the production systems will be running constantly and the amount of extra shops from the unused capacity which are not already used as robot shops are limited. When somebody looks for a rate and it is not served from the cache, but looked up in the GDS, it will add an entry to the rate change history for that specific itinerary. The idea is to use time-series analysis on the rate change history to adapt the polling frequency of the sampling system. In this way the polling frequency can be low at first and increase around the moment the rate change is anticipated to happen. Section 8.3 explains how the rate changes are tracked efficiently.

![Diagram](image)

**Baseline Situation**

After the scheduled period of time this baseline situation will have gathered two pieces of information. First, the lookup behavior and rates for the selected itineraries of the production cache is gathered. This is achieved by letting the Cache22 Esper system forward the events of the selected itineraries to the sampling system—denoted as cache in Figure 8.1. Second, the closely approximated real-life rate change behavior of the selected itineraries is collected—denoted as sampling in Figure 8.1.

In Figure 8.1 a time-line for each of the selected itineraries—denoted as \{Iter[i] | i = 0...n\}—is shown. There are two sets of lines, the sampling set and the cache set. The sampling lines show RC’s, which stand for real-life rate changes registered by the sampling engine at that time. The other set of lines show the changes that occurred in the cache at that time. The G stands for a
8.1 Mechanism

The Sampling System

Sampling

For simplicity it is assumed that the sampling registered the same as in Figure 8.1.

Improved

<table>
<thead>
<tr>
<th>Itin₀</th>
<th>C</th>
<th>G</th>
<th>G</th>
<th>C</th>
<th>rG</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Itin₁</th>
<th>rG</th>
<th>C</th>
<th>G</th>
<th>C</th>
<th>rG</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Itinₙ</th>
<th>C</th>
<th>rG</th>
<th>rG</th>
<th>C</th>
<th>rG</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 8.2: Improved situation; rG = Robot GDS Shop, G = GDS Shop and C = Cache Shop.

GDS shop and the C stands for a cache shop. For Itin₀ the cache is in sync with the GDS shops at the times when the RC happened. This means that the performance for this itinerary is excellent even without the robot on trial. For Itin₁ the cache is consistently making the GDS shops too late. At Itinₙ the cache shopped too early in the beginning, which caused it to be late for the rate change, which in turn made the cache miss a complete rate change to eventually sync again. The C’s, or cache rates, will show their relevance in the Comparison subsection.

Improved Situation

In the improved situation again a representative selection of the target itineraries is made. The major difference in respect to the previous situation is that the improvement cq. robot is now running in the production system. This can be done risk-free, because a robot only causes a rate to be shopped more frequently and makes the cache more accurate, and therefore has no negative impact. If the robot is shopping too often this will be detected easily; monitoring will see its tag show up in too many GDS shops.

So again, after the scheduled period of time, the improved situation will have gathered two pieces of information. First is the lookup behavior and rates for the selected itineraries of the improved production cache—denoted as Improved in Figure 8.2. Second is the closely approximated real-life rate change behavior of the selected itineraries—denoted as cache in Figure 8.1.

E.g., the Itin₀ still is accurate on its own, but the normal user traffic did not cause a shop at time 8 and therefore the robot stepped in with a Robot Shop—denoted as rG. In Itin₁ the robot rightfully shops at 2 to get the changed rate and is also more successful in detecting the rate change at point 8, which it detected at point 9 instead of 10 for the cache. Finally, Itinₙ did capture the change at 6 but now is slightly off at change 10 for the real change at 9, which was shopped at the right time in the cache.
The next section will explain how these situations can be compared and used to determine the effectiveness of a robot on trial.

Comparison

Within each situation the difference between the cache and the real-life rate changes can now be determined. From the registered time-stamped events of the production cache it is known when a rate was looked up—e.g., by a customer. In the case that it was served from the cache it is known if the served rate was a match or a mismatch. See Section 6. The matches do not necessarily mean that the cache was accurate, it could have been a false positive and the cache may have failed to register a rate change that was later changed back to the previous rate in real-life. This is illustrated in Figure 8.3. For a mismatch the same reasoning applies, it could have been a false negative because the rate could have been valid for a longer time, but because it was not shopped until it was changed, the previous rate is considered a mismatch. This is illustrated in Figure 8.4. Counting the false negatives, negatives, false positives and positives will provide the metric for the cache performance in comparison to the real-life situation. The time between a cache shop and the actual rate change could be negative—it was shopped too early, see Figure 8.5, or it can be positive—it was shopped too late, see Figure 8.6. A third possibility is the case where a shop took place between two other shops, but without a rate change between the two initial shops this is an unnecessary shop, see Figure 8.7. The count and duration of these three options provide another metric.

Doing the same analysis for the improved situation will give an insight in how these metrics changed when the robot was enabled. Since the metrics are based upon a representative set for the whole target population of the robot, they will reveal how the cache performance is changed by the robot. Saving the metrics allows for comparisons between different test runs, which aids tweaking and improvement of the robot.

Figure 8.3, 8.4, 8.5, 8.6 and 8.7 show the timing issues and are explained next. Figure 8.3 shows a false positive. It shows that a GDS shop is made, followed by a cache shop and again by a GDS shop. Because the rate of the two subsequent GDS shops is the same, the shop and the cache shop will be counted as a match, while the cache shop should have been a mismatch because in real-life the rate did change twice—up and down to the original price again, but those changes were not registered by the cache system.

Figure 8.4 shows a false negative. Here the cache shop is considered a mismatch, because the two consecutive GDS do not match in price and therefore everything in between is considered a mismatch. It is a false negative because the actual rate change did happen after the cache call. The rate was valid for a longer period and should have been shopped using a GDS call just before the rate change, making the cache call a match.

The other three Figures are about the timing of the GDS shops. To determine whether cache shops were a match or a mismatch two subsequent GDS shops are needed to compare the prices. To exactly track the rate changes, there should be a GDS shop just before the rate change and just after the rate
8.1 Mechanism

The Sampling System

change. This will minimize the time in which a cache shop can be displayed to a customer with the wrong rate. An early shop is one of the timing issues, and is defined as a shop that is executed too far in advance of the rate change, seen in Figure 8.5 as $G_e$. The opposite is a late shop, that is the shop that should be made right after the rate change, but instead was executed too late. This is illustrated in Figure 8.6 as $G_l$. The last thing is an unnecessary shop, this is a GDS shop executed consecutive to another GDS shop and is followed by another consecutive GDS shop again, but without any rate change occurring between any of the three shops, rendering the middle shop $G_u$—or shops, it could be any number of shops—unnecessary.

Figure 8.3: False positive.

Figure 8.4: False negative.

Figure 8.5: Early shop.

Figure 8.6: Late shop.

Figure 8.7: Unnecessary shop.
8.2 Sampling Technique

The limitations of the GDS systems suggest that it is impossible to continuously retrieve the rates for every itinerary. However, there is excess capacity that can be used to gather rates of randomly selected itineraries. The random selection provides a statistical sample; similar to the samples used in surveys or polls. Sampling is a statistical technique used when the target population is too large to be queried as a whole. Instead a subset, or a sample, is surveyed. The results obtained from the sample, using proper sampling techniques, can be considered representative of the whole. Sampling forms the basis of the Sampling System. This sampling of itinerary combinations will provide a way to measure the robot’s performance using the excess capacity that is available, to track the rate changes for the selected sample of itineraries.

There are a couple of different sampling techniques among which random, stratified, cluster, and systematic sampling are the common ones. After researching [24, 25, 6, 18, 16] the sampling techniques and several combined techniques, the stratified sampling technique was found to be most applicable for sampling the itineraries affected by a robot.

For stratified sampling, the itineraries need to be divided into strata. For each stratum the population size is known; it can be calculated from the hotel properties stored in tables combined with the demand for a certain itinerary. The strata can be based on the itinerary’s unique properties. An itinerary has the following properties that can be used as strata:

- Location (granularity can be up till city area)
- Hotel Brand
- Star rating (1-5)
- Guest Rating
- Price
- Checkin day (1-7)
- Checkin date (1-365)
- Advance Purchase
- Number of rooms
- Number of adults
- Number of children
- Amenities

Each stratum is disjoint on its own, because it has discreet properties and it is not possible to take on multiple values. For example, it is impossible for a hotel can to be located at two places at once, and for an itinerary it is impossible to
8.2 Sampling Technique

The Sampling System

have a [LOS] of 2 and 3 at the same time. When there is an interest in using multiple strata the hierarchical aspect of the properties come into play. The previously given list is ordered hierarchically. Each item on the list will make the itinerary more specific, causing the strata to be mutually exclusive.

Most of the robots are purpose-built and do not focus on the whole population, but on a subset that has a specific problem it tries to solve. Therefore, the sampling system allows to preset a property or a property range and thus focus on a more narrow population.

Figure 8.8 shows a specific set for the stratification. From the location strata the locations Americas and Europe are selected, but not Asia. Within each of the two locations, the star rating property is selected. The areas outside of the innermost circles with the stars are empty, because the strata need to be collectively exhaustive. Of course, an even more specific itinerary-based—e.g., the [AP] = 7—stratification could be made, but this would be difficult to illustrate. Figure 8.8 sufficiently explains the hierarchical principle.

Mapping the stratified sampling techniques to the Cache22 problem posed significant challenges. To sample some of the rate changes to be able to make statistically sound estimation about the rest of the rate changes was a challenge, because—in contrast to itineraries—many of the rate change properties are unknown or not precise. However, another look at the literature in the field of sampling yielded a paper that provided the breakthrough. A paper from Chatterjea, S. & Havinga, P. titled: “An adaptive and autonomous sensor sampling frequency control scheme for energy-efficient data acquisition in wireless sensor networks” described a situation quite similar to the Sampling System used here. The wireless sensors are the itineraries and the GDS calls are the energy-efficient data acquisitions. This similarity led to the insight that the problem was two fold. First, the itineraries needed to be sampled, instead of the rate changes, using the stratified sampling technique. Second, for the chosen sample itineraries the rate change frequency should be estimated using time-series analysis.
8.3 Time-series

A time-series is a set of historical values, generally measured at a regular interval. This set is used in time-series forecasting to predict future values [4].

Time-series analysis can be used for tracking the rate changes of an itinerary, using the observed history of rate changes. This will provide an estimator on the frequency of change [5]. The estimator can be used to adapt the polling frequency of the Sampling System to poll at a low frequency when the estimated value for change is low and poll more frequently when the estimate for change increases. An adaptive estimator will better estimate the changes and reduce the number of unnecessary shops even further. An additional way to save shops would be to use the GDS shops from the production system to supplement a poll from the Sampling System if the GDS shop happens within the time span where the Sampling System is scheduled to do a poll. Figure 8.9 shows the adaptive polling frequency. The horizontal lines with the dots represent the price of the itinerary over time; at the dots the price changed. The dotted vertical lines each represent a polling moment, so the sampling engine knows the price of the itinerary at each moment in time where there is a vertical line. The dashed vertical lines are the polls that capture the price change, which means that the sampling engine detected a price change within the times of the two dashed vertical lines. This example shows why it is beneficial to have an adaptive polling frequency. A uniform frequency would have wasted a lot of shops directly after the previous rate change would have been detected and would have been less precise in capturing the moment of the actual price change. The adaptive method will overcome both these problems.

![Figure 8.9: A visual representation of an adaptive polling frequency.](image)

8.4 Design and implementation

The Sampling System is designed to run autonomously. The system will be continuously running and exposes a web-service to administer it. This design decision was made for multiple reasons. As a result of this decision, the system will be centralized, meaning that not every developer has to run its own
version of the Sampling System, but can use it from a central location. The centralization has the benefit that settings and results can be shared more easily. The trials would generally run at times the normal customer traffic is at its lowest, this means that jobs has to be scheduled, which is a tedious task if each developer had to find a time-slot to run the program. At the centralized system a schedule can be defined which can be linked to a trial. The trial will be performed according to that schedule; no need for human interaction, the system will take care of the rest. Administering the system resources would be much harder if every developer would run its own system, and therefore the use of the robot [GDS] capacity would be harder to monitor, while in a central system a single shop might be useful, and thus reused for multiple trials: saving capacity instead of wasting it.

The web-service makes it easier to make other, maybe more specific, GUI’s for the system. The development of a client-side GUI that uses AJAX to communicate with the web-service was started. The GUI is developed in ExtJS, which is a JavaScript framework, as explained earlier. This framework was chosen because Priceline.com already used the framework for other projects, and in-house expertise of the ExtJS framework is readily available for future development of the GUI.

The Sampling system’s design is general and flexible so that it can be adapted to other types of sampling that might be required in the future. However, as soon as the first version of the system’s design was finished, it became clear that finishing the system would not be possible within the time allotted for this Master’s thesis. The indications that stratified sampling would yield representative samples were strong, but not conclusive. However, to see if the design was correct and to have a system that actually could perform a task. The Robot Sampling has been produced, forming the specific part of the design and implementation.

The generic design considers a couple of entities: Strata, SampleProperties, Subjects, SubjectManagers, Trials, Schedules, and Samplers. The specific parts are: RobotSampleProperties, Robot, RobotManager, StratifiedSampler. Figure 8.10 shows the classes and there relations.

The moment a developer wants to verify its robot—in the future this could be something different than a robot—a trial is performed. The term trial is used to avoid confusion with a unit test—and because it caused some naming issues in ExtJS. A trial needs to test something: the subject, during some, possibly repeated, time: the schedule.

The schedule specifies a starting time and a stopping time, and when it has to be repeated. The subject is something which is undergoing the test; currently only a robot is supported. At the moment the schedule decides that the current time is within the specified start and end times, it will signal the trial to start and when to stop again. The trial asks the specific subject manager to start or stop. The manager is created when a subject is chosen for the trial and is only created once by the ManagerFactory. The reason to have the manager (once), is that multiple trials could be running a test on the robot at the same or in overlapping time causing multiple shops for the same robot. The manager makes sure that it samples at the beginning of a sequence of trials.
and that it only tracks the rate changes for these samples once and sends the results to all the active trials.

For example, Robot X and Robot Y are both configured in the system and are Subjects of the type Robot. A developer wants to verify Robot X and sets up a schedule and both the robot and the schedule are made part of a trial, Trial A. The ManagerFactory will now create a RobotManager for Robot X and assigns the RobotManager to the Trial A. At the moment Trial A has to run it tells the manager that it wants to start. The Manager will keep track of the status of the registered trials and makes sure that the results will be send to those trials. Now another developer makes its own Trial B using Robot X and Robot Y, while selecting the same schedule as the first developer. The manager for Robot X already exists and is assigned to Trial B. Robot X is already under test by the Trial A so when Trial B starts the manager only has to forward the results to Trial A and B now. For Robot Y a new manager is created at the moment Trial B starts.

The SubjectManager knows how the specific subject has to be sampled, so when it is created it also instantiates a Sampler. Currently the design incorporates one specific sampler and that is the StratifiedSampler. A sampler will draw a sample based on the SampleProperties provided to it. For each specific Sampler there is a specific SampleProperties interface that has to be implemented; the StratifiedProperties interface for the StratifiedSampler. These specific SampleProperties also have to be implemented by a Subject specific SampleProperties; the RobotSampleProperties. This SampleProperties contains the possible values of the properties for that subject and a set of default
values. In the current design only the default values can be set type wide. A
custom property setting per specific subject and not per subject type is future
work, which also has to further refine the SubjectManager mechanism to avoid
overlapping shops.

The RobotSampleProperties will be sampled by the StratifiedSampler and
therefore contains Strata type properties. As part of the specific design a set
of Strata classes are designed and implemented. Strata can exists in two vari-
ants, either a list of discrete items or a range of continues values. Hence the
ListStrata and the RangedStrata. They both store three things: the possible
options, the default selection and the selection made.

The design and implementation of the part to track the items that were
picked by the samplers, and the actual implementation of the samplers is part
of the future work.

Figure 8.11: The custom servlet providing the web-service for the Sampling
System.

There are many ways to create a web-service with Java. After trying to
use some out-of-the-box web-service frameworks for Java, Metro Web Services
was given a try. This framework generated some problems—e.g., it had thread
safe issues, so a custom servlet was written. This had been done elsewhere in
Priceline.com and actually is straightforward.
See Figure 8.11 for the architectural overview. The SamplingServlet runs on a webserver and gets a request. The SamplingRequestFactory turns the raw HttpRequest into the specific request. The manufactured request is then given to the SamplingHandlerFactory to create the appropriate handler. The handler processes the request and performs the logic that is needed—e.g., loads an item from its store. The handler then makes a response—e.g., the loaded data from the store, and gives the SamplingResponse back to the SamplingServlet. The servlet transforms it into an HttpResponse which is sent to the browser.

The design of the GUI can be seen in Figure 8.12 and implementation of the Robot part as explained in the Appendix, is similar to the implementation of the Schedules and the Trials parts. The ExtJS store will create the applicable requests which are processed as explained in the previous paragraph. The implementation of the Sampling System itself currently contains working code for the custom servlet, the Subject, Robot, Schedule, Trial, RobotSampleProperties, RangedStrata, ListStrata, the Strata, and of course the interfaces. The strata classes read their settings from the config and the ListStrata can obtain properties from a table in the Static/Lookup databases, based upon information specified in the config. Please consult the Appendix for more specifics about the implementation. Section 9.2 will elaborate on the future work needed for the Sampling System.

Figure 8.12: The Sampling GUI to administer the system, built in ExtJS.
CHAPTER 9

CONCLUSION AND FUTURE WORK

9.1 Conclusion .................................................. 117
9.2 Future Work .................................................. 119

9.1 Conclusion

Priceline.com is an online travel company based in the United States. Booking.com and Agoda are subsidiaries of Priceline.com. Booking.com might be more familiar to the European readers, as is Agoda to the Asian readers. Priceline.com provides travel services; offering their customers hotel reservations, car rental reservations, airline tickets, vacation packages, destination services and cruises. The hotel reservation service is the focus area of this thesis. The service offers a choice of 210,000 hotels worldwide and processed over 140 million room night in 2011.

Customers can search for and book hotels on the Priceline.com website. The moment a customer enters his or her hotel specifications (location, dates, etc.), Priceline.com will show a list of hotels with the room rates that satisfy the search specifications. Unlike the model used by the subsidiaries, where hotels provide the room rates, Priceline uses a model that requires room rates to be retrieved directly from intermediate systems set up by consortia of hotels for this purpose, which are called a Global Distributed System (GDS). Priceline.com contacts each of these GDSs that are relevant for the search. The problem is that Priceline has to do this for every search from every customer, causing a significant load on the GDSs. The load generated by Priceline’s customers would even cause some of the GDS systems to collapse, therefore the amount of requests Priceline.com is allowed to do is limited. Additionally, the process introduces latency, because the rate have to be retrieved from these external GDS systems. Note that these systems are external and Priceline.com can not exert any control over them.

Priceline.com’s solution to this problem is to cache the rates. Which is difficult, due to the amount of different rates that need to be cached and because there is no control over the source of the rates. In the case of a normal client-server cache the client asks if the data it has is still valid; which will be (re)sent
by the server if it is not. A harddisk caches to improve latency. In the cache of a harddisk it is known when the sector that is cached is overwritten. Both these situations do not apply to Priceline.com, because it is not possible to ask a GDS if a previously gathered rate is still valid, instead Priceline.com has to make a full request for the current rate. An additional complication is that the GDS systems do not notify Priceline.com when rate changes occur.

Generally, a search will use a mix of cached and non-cached rates. The non-cached rates (for example, rates that have not been retrieved before or outdated rates) will be retrieved directly from a GDS. The cache has a mechanism which removes the rate from the cache when it expires. The mechanism uses a rudimentary logic to determine the expiration time for each rate. Therefore, Priceline’s project description revolved around the question how to improve this logic.

Early on in the project the extensive analysis of the problem revealed that there is a lot more to the problem then trying to find the different values for the rate expiration time. Therefore, the scope of the project was enlarged to address the following question:

*Which techniques are applicable and how should they be used to create a cache that not only performs more transparently, but also can be configured to adhere to business rules?*

Transparent implies that the customer should not be able to notice whether the rate was served from the cache or a GDS. The cache is a production system used by every customer and its performance is directly linked to the number of bookings and Priceline’s revenues. The problem analysis showed that even if a single solution to the problem would exist, there would be too much risk involved to actually implement it as part of the production systems. Therefore, it was decided that the system would undergo multiple enhancements in order to improve the cache in a stepwise manner. Each enhancement is expected to go through several stages. Each of these stages raise new questions.

One of the stages is the Measurement Stage and the Cache22 Esper system is the solution for this stage. The Measurement Stage is the fundamental stage of each enhancement, because it measures the cache performance to determine the effectiveness of each enhancement. Priority was given to develop the Cache22 Esper system as a solution for this stage because the stage is fundamental and the system acts as a solution for every enhancement. The Cache22 Esper system answers the main question. The system uses a technique called Complex Event Processing (CEP) to perform measurements. CEP is versatile and provides the basis for a solution to the other stages as well. The Cache22 Esper system matches a pattern within the messages coming from Priceline’s search engine. The messages contain information about the rate and whether the source was the cache or a GDS. The Cache22 Esper system, using the pattern and CEP, measures the performance of the cache in a valid way. The EPL of the CEP provide the system with the flexibility to make changes to the way the system measures. The CEP produces results in the form of events, these events are further post-processed by the system, which is a separate process that provides an easy way to make changes to the way the results are aggre-
An enhancement as part of the Prediction Stage is covered by the Cache22 Esper system as well. The Regression post-processing process generates a formula that predicts the accuracy of a rate based on how long ago it was refreshed. For now the combination of AP and LOS are sufficient to produce a significant increase of the cache’s performance. Although, due to the risks, the formula generated is only applied once, the system continuously gathers the variable needed and generates the regression formula almost instantly.

Although the Cache22 Esper system does not answer all the questions from the problem analysis, it still answers more than was thought feasible in the time-span of a single project that involves a running production system that processes 45 million messages per day. Till this day the system is running and continuously measuring and producing valid and compressed measurement results. The formula generated by the Regression post-processing is used to adjust the rudimentary logic increased the cache accuracy by almost seven percentage points to over 90 percent. The problem was solved sufficiently and the project generated such good results that no more adjustments needed to be made the Cache22 Esper system.

The Cache22 System architecture and design were further generalized, to act as a solution to other high volume (measurement) problems as well. Although the system’s code was generalized, refactored and stripped of unnecessary components, it still remained complex and labour intensive to apply. The DSL is developed to reduces this effort and complexity. After becoming familiar with the concept of a DSL, the language for the Cache22 Esper system is clear and intuitive enough to generate a system with minimal guidance.

Altogether, the project provides Priceline.com with a system that is not only a significant step in solving the cache performance problem by providing a correct measurement and improving the accuracy to above 90%, but that is also applicable to similar problems with minimal effort due to the generalized architecture and the DSL.

9.2 Future Work

During the problem analysis and the development of the Cache22 Esper system, it became clear that creating a solution for each stage identified in this project will be a time-consuming task. This section will describe the future work for the Cache22 Esper system, the Sampling system, and the other stages.

Measurement Stage The Cache22 Esper project is finished, and led to a configurable and extendable system that running in production. The system is a solution that supports a wide range of enhancements, because it is a system that measures the global performance of the cache, and that of a single enhancement. The DSL version of the system uses Esper, but Esper proved to be limited for the accuracy measurement and was abandoned. It still needs to be determined if this limitation is specific to the Accuracy Measurement or not. If it is a general limitation it will require another CEP platform as a replacement.
for Esper. The task of determining the extent of the limitation and finding a proper replacement is part of the future work.

The DSL version contains a syntax definition that can be used to describe the complete process flow of the Cache22 Esper system. The functionality is sufficient, but—as part of future work—additional functionality could be implemented. It is inherent for DSLs to grow in size anyway [15, p. 7]. Examples of additional functionality would be the support for different types of outputs. The system currently supports the splitting and the conditional selection of input messages, but there is no support to merge messages from multiple readers into a single event. The DSL version was used to describe the flow for another application, and the code was generated accordingly, but this new system should—as part of the future work—be tested and run in production.

The Sampling system uses sampling of specific itineraries to measure the rate changes at the GDS in order to test the robots. Future work for the Sampling system would be to research if and how the system could be used as a robot itself. Obviously the system has to be further implemented. But there are a few problems that need to be solved first. The stratified sampling method in combination with the large number of properties of an itinerary causes implementation problems. The number of strata grows rapidly if property ranges are too granular and the selection of multiple properties is allowed. The challenge is to find data types that support the large number of combinations. The Sets data structure of Google produced an overflow with its cartesian product method; this was partially solved by selecting the hotel properties first and find the IDs of the hotels with those properties and then use the grouped IDs as strata instead of each single hotel property. Still a careful selection of the property ranges was needed. Additionally, the IDs created problems inside the having clause of SQL for Oracle, because it has a limit of 1000 items. This means that the SQL statements to find the population size have to be split, which means more queries, adding execution times to the list of problems. A possible solution would be to periodically determine the ids of each strata and track changes made to the hotel inventory to update the strata accordingly. Finding solutions to these problems is part of the future work.

The use of time-series to manage the frequency for the shopping behavior of the Sampling system is not a trivial task either. Time-series techniques need the exact date at which the datapoint was measured, this information is not available, because the GDS does not provide the time the rate changed. Missing values are another problem. Time-series analysis works with historic data, but the current Search Engine works only shops for rates when they are needed. Therefore, there is the possibility that a rate change is not noticed. Because the change date and time are not available in the GDS it is not possible to know whether the rate changed just once or multiple times between two shops. How to apply time-series analysis to the Sampling system and solve the missing value problems are future work for the Sampling system.
**Prediction Stage**  The post-processing of the Cache22 Esper system has the Regression component to predict when a rate with a certain AP and LOS combination is valid. Examining the impact and usefulness of a different granularity of the itinerary properties is part of the future work. The functionality of the system already supports expanding the number of properties that are used to determine the time a rate is valid. Future work would conceive of a mechanism that allows to test a more granular selection without the risk of impacting the production systems. A too granular selection might cause problems with the number of measurements that have to be stored and purged, therefore the scalability of the storage mechanisms need to be verified and otherwise be prepared.

The produced regression formulas for each combination were used only once, because of the (financial) risk tied to their use. Similarly to the testing of a more granular selection of properties, the automatic generated formulas will need an evaluation system. If the formulas are considered risk free and deemed an improvement, the system should push them into production. Such a system will be future work.

**Improvement Stage**  For the Improvement stage exploratory research was done in the field of data mining. Studying the different techniques presented in *Introduction to Data Mining* [23], the use of decision trees seemed most promising. The challenge lies with classifying the itineraries. The classifier would be the time a rate would remain valid, which is essentially a paradox because that is the overall problem that has to be solved. Still, the Sampling system could be used to produce the training sets of itineraries with known labels. The best fit for association analysis seem to be the rule generation techniques. These rules can be turned into EPL, and Esper or another CEP system can then use these rules to feed robots or even act as a Decision Engine in front of the Search Engine that decides whether to use the cache or GDS based on the outcome of the rules.

**Regulation Stage**  If a Decision Engine as described in the previous paragraph would be implemented it can also serve as a solution to allow decisions based on manually created business rules. The initial idea to use a DSL to generate EPL looks to be a viable solution when the DSL is written from the business perspective. The expressiveness of the EPL would be hard to capture in a DSL, but the other way around it would imply that the business rule expressivity should be captured in a DSL. This is a fundamental difference, instead of trying to translate EPL into a language that is understandable for business people, it will translate what business people want to say into EPL. This language will be more specific and probably fits within a scope of a Master’s project. Additionally, a DSL which generates EPLs that trigger the robots can be made to create and modify robots based on a language that describes what they should do.
Coverage  No research was done to find solutions for the coverage stage. There are two ideas that could be used as a basis for future work. Both would involve visualization techniques: one to visualize the impact of changes a robot makes to the coverage of the cache, and the other would be to visualize problem areas within the cache based on their request demand.
# APPENDICES

## APPENDIX

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Esper Code Samples</td>
<td>125</td>
</tr>
<tr>
<td>ExtJS Code Examples</td>
<td>127</td>
</tr>
<tr>
<td>Cache22 Esper</td>
<td>130</td>
</tr>
</tbody>
</table>

## BIBLIOGRAPHY

139

## ACRONYMS

141

## GLOSSARY

143
The Appendix contains code samples to explain certain Esper features in
more detail, like: the Socket Adapter, Subscribers, and information about
some operator specifics. Then it follows with the Robot part of the Sam-
pling system code to explain how ExtJS was used. The appendix ends with
two extended descriptions of some of the internals of Cache22 Esper sys-
tem: all the EPL and the complete Cache22 EsperHA DSL syntax.

Esper Code Samples

This appendix section contains code examples for Esper. Starting with an ex-
ample of how the Socket Adapter is used. Followed by a description of the
Subscriber mechanism.

```java
// connect first
Socket requestSocket = new Socket("localhost", port);
ObjectOutputStream out =
        new ObjectOutputStream(requestSocket.getOutputStream());

// send a few events, here we send only one
out.writeObject(new MyEvent("Hello World"));
out.flush();

// out.reset();
// close when done
out.close();
requestSocket.close();
```

Listing 27: Client-side socket code [10, p. 18–19].

Esper IO: Sockets

The code fragment in Listing 28 contains the actual Cache22 Esper code that
starts the sockets in the Esper Engine. See the EsperIO documentation [10]
p. 17–18] for more information and how to configure the socket using xml.
Other adapters that can be used to deliver events can also be found in the
EsperIO reference documentation.

After the Socket Adapter has been setup on the Esper side it will accept
connections and events can be sent when the connection is established. The
following example is sample code for sending an event from the client to the
engine using a socket, shown in Listing 27.
/**  
 * Start the Socket.  
 */  
private void startSocket() {  
    ConfigurationSocketAdapter adapterConfig =  
        new ConfigurationSocketAdapter();  

    // Configure the socket  
    SocketConfig socket;  

    for (int i = 0; i <= SOCKET_COUNT; i++) {  
        socket = new SocketConfig();  
        socket.setDataType(DataType.OBJECT);  
        socket.setPort(SOCKET_PORT + i);  
        adapterConfig.getSockets().put("SocketService" + (SOCKET_PORT + i),  
            socket);  
        LOGGER.info("SocketService" + (SOCKET_PORT + i)  
                       + " registered on port: " + (SOCKET_PORT + i));  
    }  

    // start adapter  
    socketAdapter = new EsperIOSocketAdapter(adapterConfig, uri);  
    socketAdapter.start();  
}  

Listing 28: Engine-side socket code.

**Esper Subscribers**

There are multiple ways to receive events. A subscriber can use row-by-row delivery, illustrated with the following statement in Listing 29 and subscriber code in Listing 30.

\[
\text{select orderId, price, count(*) from OrderEvent}
\]

Listing 29: The example statement to illustrate the row-by-row delivery.

```java
public class MySubscriber {
    ...
    public void update(String orderId, double price, long count) {...}
    ...
}
```

Listing 30: Row-by-row delivery update method signature.

When a wildcard is used the whole event object has to be used as an argument for the method. The rows can also be delivered as a Map or Object arrays. The code template of how the update method should be implemented for Map or Object array delivery is shown in Listing 31.
Appendix

ExtJS Code Examples

// Map
public void update(Map row) {...}

// Object Array
public void update(Object[] row) {...}

Listing 31: Map and Object array delivery update method signature.

In the Esper Engine the multi-row delivery is used. In Phase-I the Map[] was used, while in the DSL version the Object[][] is used. With the Map[], each map represents one event and the Map entries are the columns of the select clause. The Object[][]'s first dimension of each Object[] is the event row, and the second dimension is the column of the select clause.

// Map
public void update(Map[] insertStream, Map[] removeStream){..}

// Object Array
public void update(Object[][] insertStream, Object[][] removeStream){..}

Listing 32: Map and Object array delivery update method signature.

Pattern Operator Precedence

Esper uses the operator precedence explained in Table 1. The first column is the order of precedence, the second column lists the operator category, the third column shows the operator(s) of the category, and the final column shows an example statement.

Every Operator

The every clause can be used in several ways when combined with parentheses. Table 2 shows the operator examples. Each description uses the following event sequence: A₁ B₁ C₁ B₂ A₂ D₁ A₃ B₃ E₁ A₄ F₁ B₄.

ExtJS Code Examples

This section contains code from Cache22 Sampling which is developed using ExtJS from Sencha. The Sencha sample [21] was the place to start, because it was the most applicable to the Cache22 Sampling project and gave a good general understanding of ExtJS.

ExtJS uses a what they call “unified directory structure” that is the same for each app. In this structure the index.html acts as the bootstrap for ExtJS and is shown in Listing 33.
## Precedence

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>unary</td>
<td>every, not, every-distinct</td>
<td>every MyEvent timer:interval(5 min) and not MyEvent</td>
</tr>
<tr>
<td>2</td>
<td>guard post-fix</td>
<td>where timer:within and while (expression) (incl. withinmax and plug-in pattern guard)</td>
<td>MyEvent where timer:within(1 sec)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a=MyEvent while (a.price between 1 and 10)</td>
</tr>
<tr>
<td>4</td>
<td>and</td>
<td>and</td>
<td>every (MyEvent and MyOtherEvent)</td>
</tr>
<tr>
<td>5</td>
<td>or</td>
<td>or</td>
<td>every (MyEvent or MyOtherEvent)</td>
</tr>
<tr>
<td>6</td>
<td>followed-by</td>
<td>-&gt;</td>
<td>every (MyEvent -&gt; MyOtherEvent)</td>
</tr>
</tbody>
</table>

Table 1: Pattern Operator Precedence [8, p. 153].

### Listing 33: The index.html code.

```html
<!DOCTYPE html>
<html>
<head>
  <link type="text/css" rel="stylesheet" href="lib/extjs/resources/css/ext-all.css"/>
  <link type="text/css" rel="stylesheet" href="css/style.css"/>
  <script type="text/javascript" src="lib/extjs/ext-all-debug.js"/>
</head>

<script type="text/javascript" src="app.js"></script>

<title>Priceline.com - Cache22 Sampling</title>
</body>
</html>
```

Within the app.js the actual Javascript application is started by instantiating the Application class, which looks as shown in Listing 34.

The controller used to perform the actions and render the views and thus
### Example Description

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>every ((A \rightarrow B))</td>
<td>Detect an (A) event followed by a (B) event. At the time when (B) occurs the pattern matches, then the pattern matcher restarts and looks for the next (A) event.</td>
</tr>
<tr>
<td></td>
<td>1. Matches on (B_1) for combination (A_1,B_1)</td>
</tr>
<tr>
<td></td>
<td>2. Matches on (B_3) for combination (A_2,B_3)</td>
</tr>
<tr>
<td></td>
<td>3. Matches on (B_4) for combination (A_4,B_4)</td>
</tr>
<tr>
<td>every (A \rightarrow B)</td>
<td>The pattern fires for every (A) event followed by a (B) event.</td>
</tr>
<tr>
<td></td>
<td>1. Matches on (B_1) for combination (A_1,B_1)</td>
</tr>
<tr>
<td></td>
<td>2. Matches on (B_3) for combination (A_2,B_3) and (A_3,B_3)</td>
</tr>
<tr>
<td></td>
<td>3. Matches on (B_4) for combination (A_4,B_4)</td>
</tr>
<tr>
<td>(A \rightarrow every B)</td>
<td>The pattern fires for an (A) event followed by every (B) event.</td>
</tr>
<tr>
<td></td>
<td>1. Matches on (B_1) for combination (A_1,B_1)</td>
</tr>
<tr>
<td></td>
<td>2. Matches on (B_2) for combination (A_1,B_2)</td>
</tr>
<tr>
<td></td>
<td>3. Matches on (B_3) for combination (A_1,B_3)</td>
</tr>
<tr>
<td></td>
<td>4. Matches on (B_4) for combination (A_1,B_4)</td>
</tr>
<tr>
<td>every (A \rightarrow every B)</td>
<td>The pattern fires for every (A) event followed by every (B) event.</td>
</tr>
<tr>
<td></td>
<td>1. Matches on (B_1) for combination (A_1,B_1)</td>
</tr>
<tr>
<td></td>
<td>2. Matches on (B_2) for combination (A_1,B_2)</td>
</tr>
<tr>
<td></td>
<td>3. Matches on (B_3) for combination (A_1,B_3) and (A_2,B_3) and (A_3,B_3)</td>
</tr>
<tr>
<td></td>
<td>4. Matches on (B_4) for combination (A_1,B_4) and (A_2,B_4) and (A_3,B_4) and (A_4,B_4)</td>
</tr>
</tbody>
</table>

Table 2: Every Operator [8 p. 157].
providing the logic of the application. The code in Listing 35 is the controller for managing the Robots within the Cache22 Sampling.

The Robot controller does multiple things and is concerned with the CRUD operations of the Robots. This is where the views come to play. A view is a reusable component that displays other components. The next example is one of the views of the Cache22 Sampling project and it is part of the Robot controller—see Listing 36.

To be able to show data in the view’s grid a store has to be defined. A store is a placeholder for data and allows for the general CRUD operations. The data package contains multiple ways to create a store, just as a simple fields description and an array of data to a store that is linked to a web-service. This is shown in Listing 37.

A model is used to capture more complex data structures and make them reusable over multiple stores. The following code contains a Proxy definition. This Proxy is used to save data to a server. The proxy can be defined in either the store or the model. Here the proxy is created within the model because the data has to be stored in the same way across all stores. See Listing 38 for the Robot model code.

**Cache22 Esper**

**EPL statements**

This section contains the EPL statements of the Cache22 Esper system. Listing 39 shows the definition of the events by creating a schema. These are internal events, the Processor registers the event sent by the Feeders. The statement in Listing 40 creates the Cache Event window where the cache events are merged and pickup from when the second GDS event arrives. The next three Listings 41, 42, and 43 are familiar, because the were explained earlier in Section 6.3. The last two Listings 44 and Listing 45 take care of cleaning up corner case events, which would otherwise pollute the window.

**Cache22 EsperHA DSL Syntax definition**

This section contains the complete syntax definition of the Cache22 EsperHA DSL. The section can be used as a reference when using the syntax and can be of interest for people who would like to know how the syntax is defined. The definition is spread over three Listings 46, 47, and 48.
Ext.onReady(function(){
  Ext.application({
    name: 'AM',
    appFolder: '.',
    controllers: [
      'Trials',
      'Robots',
      'Schedules'
    ],
    launch: function() {
      Ext.create('Ext.Viewport', {
        layout:'border',
        ...
        items: [{
          id: 'toolbar',
          xtype: 'toolbar',
          region: 'north',
          ...
        }, { // LEFT
          title: 'Navigation',
          region: 'west',
          ...
          layout: 'accordion',
          items: [
            {xtype: 'trialslist'},
            {xtype: 'robotslist'},
            {xtype: 'scheduleslist'}
          ]
        }, { // MAIN
          id: 'main',
          region: 'center',
          layout: 'card',
          ...
          items: [
            { html: '<h1>The Cache22 Sampling Engine</h1><p>...</p>' },
            { id: 'trialsedit', xtype: 'trialsedit' },
            { id: 'robotsedit', xtype: 'robotsedit' },
            { id: 'schedulesedit', xtype: 'schedulesedit' }
          ]
        }]
      });
  });

  var tb = Ext.getCmp('toolbar');
  tb.add({
    text: 'Trials',
    menu: [{
      xtype: 'menu',
      items: [{
        text: 'Add'
      }, { text: 'View' }
    ]
  }],
  ...

Listing 34: The Application class instantiation.
Ext.define('AM.controller.Robots', {
    extend: 'Ext.app.Controller',
    // Configuration
    stores: ['Robots'],
    models: ['Robot'],
    views: [
        'robots.List',
        'robots.Edit'
    ],
    refs: [{
        ref: 'form',
        selector: 'robotsedit form'
    }],
    /**
     * Add a Robot.
     */
    addRobot: function() { ... },
    /**
     * Edit a Robot.
     */
    editRobot: function(view, record, row, e, opt) { ... },
    /**
     * Save a Robot.
     */
    saveRobot: function(button) { ... },
    /**
     * Delete a Robot.
     */
    deleteRobot: function(button) { ... },
    /**
     * Initialize the controller.
     */
    init: function() {
        // Bind events
        this.control({
            'robotsedit button[action=save]': {
                click: this.saveRobot
            },
            'robotslist button[action=add]': {
                click: this.addRobot
            },
            'robotslist button[action=delete]': {
                click: this.deleteRobot
            },
            'robotslist': {
                itemclick: this.editRobot
            }
        });
    }
});

Listing 35: The (Robot) Controller class instantiation.
Ext.define('AM.view.robots.List', {
    extend: 'Ext.grid.Panel',
    alias : 'widget.robotslist',
    // Configuration
    title : 'Robots',
    store: 'Robots',
    columnLines: true,
    border: false,
    frame: false,
    // Components
    // The grid
    columns: [
        {header: 'Name', dataIndex: 'identifier', flex: 1},
        {header: 'Description', dataIndex: 'description', flex: 1}
    ],
    // Buttons
    dockedItems: [{
        xtype: 'toolbar',
        items: [
            // Add button
            Ext.create('Ext.Action', {
                iconCls : 'icon-add',
                text: 'Add',
                action: 'add'
            } ),
            // Separator
            '-',
            // Delete button
            Ext.create('Ext.Action', {
                id: 'listRobotDelete',
                iconCls: 'icon-delete',
                text: 'Delete',
                disabled: true,
                action: 'delete'
            } ),
        ],
        dock: 'bottom'
    }],
    // Initialize the component.
    initComponent: function() {
        this.callParent(arguments);
        // Bind the selection event
        this.getSelectionModel().on('selectionchange', this.onSelectChange, this);
    },
    // List select event handler.
    onSelectChange: function(selModel, selections){
        this.down('#listRobotDelete').setDisabled(selections.length === 0);
    }
});

Listing 36: The (Robot list) view class instantiation.
Cache22 Esper

Listing 37: The (Robot) store class instantiation.

Listing 38: The (Robot) model class instantiation.

Listing 39: The creation of schema, which are internal events.
create window CSAccCacheEventWindow.std:unique(HotelID, CheckIn, Los).win:time(7 days) as (HotelID Long, CheckIn Long, Los Integer, TimeArray String, NumHits Integer)

Listing 40: The creation of a the cache window.

On DBTriggerEvent as ae
insert into DbUpdateEvent
    select ae as gdsEvent, csw.NumHits as NumHits,
          csw.timeArray as TimeArray
    from CSAccCacheEventWindow as csw
    where csw.HotelID = ae.accEvent.HotelID
          and csw.CheckIn = ae.accEvent.CheckIn
          and csw.Los = ae.accEvent.Los

Listing 41: The on DbTriggerEvent statement.

on DbUpdateEvent as de
delete from CSAccCacheEventWindow as accWindow
    where accWindow.HotelID = de.gdsEvent.accEvent.HotelID
          and accWindow.CheckIn = de.gdsEvent.accEvent.CheckIn
          and accWindow.Los = de.gdsEvent.accEvent.Los

Listing 42: The on DbUpdate statement.

select * from DbUpdateEvent

Listing 43: The select DbUpdate statement.

on pattern [every timer:interval(8 hours)]
delete from CSAccCacheEventWindow as csace
    where csace.CheckIn < (current_timestamp - (24 * 60 * 60 * 1000))

Listing 44: The delete expired cache statement.

on CleanupEvent as ce
delete from CSAccCacheEventWindow as accWindow
    where accWindow.HotelID = ce.HotelID
          and accWindow.CheckIn <= ce.CheckIn
          and accWindow.Los = ce.Los

Listing 45: The cache clean up event.
grammar com.priceline.cache22.esper.EsperDSL with org.eclipse.xtext.xbase.Xbase

generate esperDSL "http://www.priceline.com/cache22/esper/EsperDSL"

import "http://www.eclipse.org/xtext/common/JavaVMTypes" as types
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Model:
  imports += Import*
  types += (Type)*
  package = Package
;

Import:
  'import' importedNamespace=QualifiedNameWithWildCard ';'
;
QualifiedNameWithWildCard:
  QualifiedName ('.' ' *')?
;

Package:
  'package' name = QualifiedName ';'
  entries += Entry*
;

Entry:
  Input | Event | Output | Transform | Queue | Processor | Subscriber | Listener
;

Listener:
  'listener' name=ID '{'
    'events':'' (events+= [Event] ';')*
  '}'
;

Subscriber:
  'subscriber' name=ID '{'
    'events':'' (events+= [Event] ';')*
  '}'
;

Queue:
  'queue' name = ID '{'
    ('section' ' = ' section = ID ';')?
  '}'
;

Listing 46: Cache22 EsperHA DSL Syntax definition 1/3.
Input:

```java
'input' name = ID '{
    ('object' '=' object = [types::JvmType] ';')
    (attributes += Attribute)*
    ('queue' ':'
        queue = [Queue]
    ');
    ('filter' ':'
        (expression = Expression)
    ');
}'}

Expression:

```java
left = LeafExpression ({Expression.left=current}
    op = LogicOperand expr = LeafExpression)*
```

LogicOperand:

```java
'&&' | '&&' | '|' | '||' | '||'
```

LeafExpression returns Expression:

```java
'(' inner = Expression ')' | leftfunction = FunctionCall op = CompareOperand value = Value | attr = [Attribute] op = CompareOperand value = Value
```

Value:

```java
string = STRING | int = INT | bool = 'true' | bool = 'false' | function = FunctionCall
```

CompareOperand:

```java
'==' | '!==' | '<>' | '<' | '>' | '<=' | '>='
```

Type:

```java
'type' name=ID 'mapped-to' javaType=[types::JvmType | QualifiedName] ';
```

FunctionCall:

```java
name = QualifiedName '(' (args += [Attribute] (',' args += [Attribute] )*? ')')
```

Attribute:

```java
name = ID ':' type = JvmTypeReference '
```

AttributeNull:

```java
name = ID ':' type = JvmTypeReference '|'? null = ('null')? '
```

Listing 47: Cache22 EsperHA DSL Syntax definition 2/3.
Event:
  'event' name = ID ('type = ('in' | 'out') ')'? '{
  (attributes += Attribute)*
  '}

Output:
  'output' name = ID '{
  (attributes += AttributeNull)*
  '}

Transform:
  TransformIn | TransformOut

Processor:
  'processor' name = ID '{
  (statements+= Statement)+
  '}

Statement:
  'statement':
  'name': name = ID ;
  ('subscribe': subscribe = [Subscriber] ')?
  ('listen': listen = [Listener] ')?

TransformIn:
  'transform input' from = [Input] 'to event' event = [Event] '{
  ('condition':
    (condition = Expression)
  ';')?
  (conversions += Conversion)*
  '}

TransformOut:
  'transform event' event = [Event] 'to output' to = [Output] '{
  ('condition':
    (condition = Expression)
  ';')?
  (conversions += ConversionNull)*
  '}

Conversion:
  this = [Attribute] '->' that = [Attribute] ';

ConversionNull:
  this = [Attribute] '->' that = [AttributeNull] ';


**Acronyms**

**AP**  Advance Purchase.

**CEP**  Complex Event Processing.

**CSV**  Comma-Separated Values.

**DSL**  Domain Specific Language.

**EPL**  Event Processing Language.

**EsperEE**  Esper Enterprise Edition.

**EsperHA**  Esper High-Availability.

**GDS**  Global Distributed System.

**GPL**  GNU General Public License.

**JMX**  Java Management Extensions.

**JVM**  Java Virtual Machine.

**LOS**  Length of Stay.

**MQ**  IBM Message Queue.

**OTA**  Online Travel Agency.

**POJO**  Plain Old Java Object.

**SQL**  Structured Query Language.

**XML**  Extensible Markup Language.
AP  Advance Purchase (AP) is the number of days between the booking date and the checkin date.

BLOB  A Binary Large OBject is a datatype used for the storing large amounts of data. Binary data exists in many flavors; PDFs, pictures, and raw data are all examples of BLOB data.

CamelCase  Variable and methods, etc, that are compound words are written with each element's initial letter capitalized within the compound and without spaces. The first letter can either be upper or lower case. This convention is called CamelCase.

CEP  Complex event processing (CEP) are software infrastructures that can detect patterns of events (and expected events that did not occur) by filtering, correlating, contextualizing, and analyzing data captured from disparate live data sources to respond as defined using the platform's development tools [7, p. 2].

DSL  A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [26, p. 1].

EMF  The Eclipse Modeling Framework Project is a modeling framework and code generation facility for building tools and other applications based on a structured data model.

EPL  Event Processing Language (EPL) is a language for Esper to express filtering, aggregation, and joins, possibly over sliding windows of multiple event streams. It also includes pattern semantics to express complex temporal causality among events (followed-by relationship) [11].

Esper  Esper is an open source event stream processing (ESP) and event correlation engine (CEP) [11].

Esper Enterprise Edition  Esper Enterprise Edition is a complete turnkey solution for addressing Event Processing and CEP requirements in enterprise grade deployments. It combines Esper, Data Distribution Service, Rich Multi-Window GUI and Continuous Displays, EsperJMX, EsperJDBC in one single certified and supported package [12].
**Esper High-Availability** EsperHA is a complete solution for zero-downtime ESP/CEP event processing. It combines Esper with local in-memory caching, resilient overflow to disk or database and clustered configuration with hot backup capabilities. EsperHA ensures no event or state is lost upon a failure, and enables you to deal with massive data volume that otherwise don’t fit in process memory. Its fine grained configuration enables you to fine tune high availability and resiliency on a per stream and continuous query level [13].

**GDS** Global Distributed System (GDS) is a system within Priceline.com that connects to all the rate providers.

**Java Management Extensions** The JMX technology provides the tools for building distributed, Web-based, modular and dynamic solutions for managing and monitoring devices, applications, and service-driven networks. By design, this standard is suitable for adapting legacy systems, implementing new management and monitoring solutions, and plugging into those of the future [19].

**JMS** Java Message Service (JMS) The Java Message Service (JMS) API is a messaging standard that allows application components based on the Java 2 Platform, Enterprise Edition (J2EE) to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous [20].

**LOS** Length Of Stay (LOS) is an itinerary property stating the number of nights a room is booked.

**Map** The `java.util.Map`—in short: Map—is a Java interface. It is an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

**min-rate** min-rate stand for minimum rate, this is the minimum rate that a hotel has for the certain [AP] and [LOS] combination.

**MVC** Model View Controller is a design pattern, commonly used for web-applications, that consists of three parts with the goal to separate the user interface from the underlying data. The Model is the part that contains the data that is presented by the View. The Controller translates the actions performed in the View to the modification of the data in the Model [17].

**SpringJMS** Spring provides a JMS integration framework that simplifies the use of the JMS API and shields the user from differences between the JMS 1.0.2 and 1.1 APIs. JMS can be roughly divided into two areas of functionality, namely the production and consumption of messages.
XML. Extensible Markup Language; a markup language that encodes a description of the document’s storage layout and structure. The encoding is both human- and machine-readable.