Leveraging Big Data Technologies in Marketing Automation

LAURENS VIJNCK

Promotor:
Prof. Dr. Frank NEVEN

Supervisors:
Dr. Jonny DAENEN
Ing. Dirk DUPONT

A thesis presented for the degree of
Master of Computer Science

Departement of computer science
2018-2019
Abstract

Selligent Marketing Cloud offers end-to-end marketing solutions for businesses. The platform captures tons of user-interaction data on a daily basis and aims to expose the data to provide insightful reporting and analysis. While Selligent has been able to successfully provide such reporting functionalities, the company is longing for a future proof solution that provides increased scalability. Moreover, the reporting functionalities are limited in the sense that they focus on a single reporting dimension. In reality, however, the interaction data can be used to deliver insights in multiple dimensions.

This work describes how recent advancements in the field of large-scale, distributed processing are leveraged to create a scalable end-to-end reporting application. This application delivers near real-time results and allows for multiple dimensions of the interaction data to be addressed. The activities carried out within the framework of this thesis already contribute directly to production innovations at Selligent.
Acknowledgements

I would like to take this opportunity to express my gratitude to my promotor Prof. Dr. Frank Neven and my supervisors, Dr. Jonny Daenen and Ing. Dirk Dupont for their intense supervision during this thesis. They provided me with extreme motivation and appropriate feedback during the course of this academic year.

Moreover, I would like to thank Selligent Marketing Cloud for the amazing opportunity to perform this thesis within the framework of their platform. I am thankful for the contributions made by the fellow workers, Hanne van Briel, Cedric Spaas, Benjamin Daerden and Oliver Hermans, to the initial stages of this thesis. Additionally, I would like to thank Google, and especially Lorin Sleeuwagen, for arranging the financial resources to experiment with the fascinating landscape of technologies offered by the Google Cloud Platform.

Finally, I would like to thank Kirill Ismagulov for his support during the extensive process of conducting the load tests and associated activities.
# Contents

## 1 Introduction
1.1 Advancements in Distributed Data Processing ................................................. 1
1.2 Marketing Automation ..................................................................................... 3
1.3 Research Goal ................................................................................................. 4

## 2 Marketing Insights
2.1 Online Marketing ............................................................................................. 6
2.2 Concepts .......................................................................................................... 6
2.3 Selligent Marketing Cloud
   2.3.1 Content ...................................................................................................... 8
   2.3.2 Consumer .................................................................................................. 10
   2.3.3 Channels ................................................................................................... 10
   2.3.4 Time .......................................................................................................... 11
   2.3.5 Journeys ................................................................................................... 12
2.4 Outbound versus Inbound Communication ....................................................... 14
2.5 Data Model ..................................................................................................... 15
2.6 Reporting ........................................................................................................ 16
   2.6.1 Evaluation Metrics ..................................................................................... 17
   2.6.2 Reporting Dimensions ................................................................................. 18
   2.6.3 AS-IS Architecture ...................................................................................... 19
   2.6.4 Shortcomings in the Current Architecture .................................................... 19
   2.6.5 TO-BE Architecture ................................................................................... 19

## 3 Data Models
3.1 OLAP versus OLTP ........................................................................................ 21
3.2 Relational Model ............................................................................................ 22
   3.2.1 Normalization ............................................................................................ 23
   3.2.2 Column-Oriented Storage .......................................................................... 25
3.3 NoSQL Landscape .......................................................................................... 26
   3.3.1 Key-Value Stores ....................................................................................... 26
   3.3.2 Document Databases ................................................................................ 27
   3.3.3 Graph Data Model ..................................................................................... 28
## 4 Data Technologies

4.1 Google Bigtable
   4.1.1 Data Model .............................................. 30
   4.1.2 Cluster Overview ..................................... 31
   4.1.3 Tablet Representation ................................ 34

4.2 Megastore .................................................. 35
   4.2.1 Data Model ............................................. 36

4.3 Google BigQuery .......................................... 37

4.4 Cloud Data Technologies Summary ................... 38

4.5 Publish/Subscribe Messaging ......................... 40
   4.5.1 Google Cloud Pub/Sub .................................. 40
   4.5.2 Architectural Overview ................................ 42
   4.5.3 Kafka .................................................. 43
   4.5.4 Messages and Batches .................................. 43
   4.5.5 Topics, Partitions and Streams ....................... 44
   4.5.6 Producers, Consumers and Consumer Groups ........ 44
   4.5.7 Brokers ............................................... 45

## 5 Data Processing

5.1 Batch Processing ........................................... 47

5.2 The Lambda Architecture ................................ 48

5.3 Stream Processing ......................................... 48

5.4 Preliminaries ............................................... 49

5.5 Duality of Streams and Tables ....................... 50

5.6 Time Domains .............................................. 51
   5.6.1 Time-domain Skew .................................... 52

5.7 Windowing .................................................. 57
   5.7.1 Window Alignment ..................................... 58
   5.7.2 Windowing Strategies .................................. 58
   5.7.3 Event Time and Processing Time Windowing ........ 61

## 6 Apache Beam

6.1 Introduction ................................................. 64

6.2 Completeness, Latency and Cost ...................... 65

6.3 Pipeline Topology ........................................ 67
   6.3.1 PCollections .......................................... 68
   6.3.2 PTransforms .......................................... 69

6.4 Windowing Model ......................................... 75

6.5 Watermarks ................................................. 79

6.6 Triggers and Panes ...................................... 82
   6.6.1 Global Windowing .................................... 86
   6.6.2 Allowed Lateness ..................................... 86

6.7 Accumulation .............................................. 89

6.8 Tracking Pipeline Progress Through Watermarks .... 91
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8.1 Output Timestamps</td>
<td>97</td>
</tr>
<tr>
<td>6.9 Data Latency Versus System Lag</td>
<td>98</td>
</tr>
<tr>
<td>6.10 Verifying Pipeline Correctness</td>
<td>99</td>
</tr>
<tr>
<td>7 Pipeline Execution</td>
<td>102</td>
</tr>
<tr>
<td>7.1 Google Cloud Dataflow</td>
<td>102</td>
</tr>
<tr>
<td>7.2 Fault Tolerance</td>
<td>103</td>
</tr>
<tr>
<td>7.3 Processing guarantees</td>
<td>103</td>
</tr>
<tr>
<td>7.4 Parallelism</td>
<td>104</td>
</tr>
<tr>
<td>7.5 Side Effects</td>
<td>108</td>
</tr>
<tr>
<td>7.6 Graph Construction</td>
<td>114</td>
</tr>
<tr>
<td>7.7 Optimizing the Execution Plan</td>
<td>115</td>
</tr>
<tr>
<td>7.7.1 Fusion Optimization</td>
<td>115</td>
</tr>
<tr>
<td>7.7.2 Combine Optimization</td>
<td>115</td>
</tr>
<tr>
<td>7.8 Bundling of Elements</td>
<td>118</td>
</tr>
<tr>
<td>7.9 Handling Failures</td>
<td>120</td>
</tr>
<tr>
<td>7.10 Sinks</td>
<td>121</td>
</tr>
<tr>
<td>7.11 Sources</td>
<td>122</td>
</tr>
<tr>
<td>7.12 Dynamic Autoscaling</td>
<td>124</td>
</tr>
<tr>
<td>7.13 Conclusion</td>
<td>125</td>
</tr>
<tr>
<td>8 Send Time Optimization</td>
<td>126</td>
</tr>
<tr>
<td>8.1 Introduction</td>
<td>126</td>
</tr>
<tr>
<td>8.2 Consumer Profiles</td>
<td>127</td>
</tr>
<tr>
<td>8.3 Implementation</td>
<td>128</td>
</tr>
<tr>
<td>8.3.1 Ingestion</td>
<td>128</td>
</tr>
<tr>
<td>8.3.2 Global Windows</td>
<td>129</td>
</tr>
<tr>
<td>8.3.3 Internal State</td>
<td>130</td>
</tr>
<tr>
<td>8.3.4 External State</td>
<td>131</td>
</tr>
<tr>
<td>8.3.5 Cloud Functions</td>
<td>135</td>
</tr>
<tr>
<td>8.3.6 Hybrid Approach</td>
<td>136</td>
</tr>
<tr>
<td>8.3.7 Hybrid Approach with Cache Invalidation</td>
<td>137</td>
</tr>
<tr>
<td>9 Experimental Results</td>
<td>139</td>
</tr>
<tr>
<td>9.1 Setup</td>
<td>139</td>
</tr>
<tr>
<td>9.2 Metrics</td>
<td>140</td>
</tr>
<tr>
<td>9.3 Throughput Testing</td>
<td>141</td>
</tr>
<tr>
<td>9.3.1 Scenario 1</td>
<td>141</td>
</tr>
<tr>
<td>9.4 Scenario 2</td>
<td>146</td>
</tr>
<tr>
<td>9.5 Scenario 3</td>
<td>153</td>
</tr>
<tr>
<td>9.5.1 Autoscaling Test</td>
<td>157</td>
</tr>
<tr>
<td>9.6 Streaming Tests</td>
<td>161</td>
</tr>
<tr>
<td>9.6.1 Scenario 1</td>
<td>161</td>
</tr>
<tr>
<td>9.6.2 Scenario 2</td>
<td>162</td>
</tr>
</tbody>
</table>
CONTENTS

9.7 Conclusion .................................................. 166

10 End-to-end Reporting Application .................................. 167
  10.1 Reporting Use-cases ......................................... 167
     10.1.1 Overview ............................................... 167
     10.1.2 Consumer Profile ....................................... 169
     10.1.3 Organization Level Insights ............................ 170
     10.1.4 Journey ................................................ 171
  10.2 Introduction ............................................... 172
  10.3 High-level Overview ....................................... 173
  10.4 Ingestion ................................................. 173
  10.5 Processing .............................................. 175
  10.6 Storage ................................................. 175
  10.7 Back-end API ............................................ 176
  10.8 Front-end Application .................................... 176
  10.9 Conclusion ............................................... 177

11 Conclusion and Future Work ...................................... 180

A Nederlandse Samenvatting ........................................ 188
Chapter 1

Introduction

Selligent Marketing Cloud offers end-to-end marketing solutions for businesses. The platform captures tons of user-interaction data on a daily basis and aims to expose the data to provide insightful reporting and analysis. While Selligent has been able to successfully provide such reporting functionalities, the company is longing for a more future proof solution.

1.1 Advancements in Distributed Data Processing

Large scale distributed processing is complex and exciting field. In the modern society, massive global-scale datasets are the rule rather than the exception [22]. Canonical examples of such datasets are web requests logs, mobile statistics or network sensor data. Given the significant amount of business value that they carry, there is an increased interest in extracting information from these datasets. Most of the logic to extract the information of interest is conceptually simple, e.g., determining the most frequent queries in a given day or computing hourly aggregates. However, to process such global-scale datasets, computations have to be distributed across clusters in order to finish in a reasonable amount of time [65]. These clusters commonly consist of hundreds or thousands of commodity machines wherein failures are frequently encountered. Executing programs in a distributed fashion introduces many challenges. The computation has to be parallelized, input data has to be distributed, and mechanisms to cope robustly with machine failures have to be introduced [42]. Addressing these issues translates to an inherent complexity, in the otherwise simple computation logic.

MapReduce made the first major contribution to the world of distributed data processing. The authors introduce a programming model, and associated run-time system, that expresses computational logic using two related functions, i.e., the map and the
reduce functions. Programs written using this model are automatically parallelized and executed on a cluster. This model abstracts the programmer from the highly complex details of distributed execution. Specifically, the run-time system takes care of the details of partitioning the input data, scheduling the program’s execution across a set of machines, handling machine failures, and managing the required inter-machine communication [34]. The work inspired a number of open-source initiatives including, but not limited to, Apache Hadoop [1].

While the functional map-reduce style of programming allows for a wide variety of use-cases to be addressed, it lacks the expressiveness required for many real-world computations. In reality, many classes of applications required a sequence, or a pipeline, of MapReduce jobs [30]. Chaining MapReduce stages requires additional work to manage the creation and later deletion of the intermediate results between pipeline stages. Moreover, scheduling multiple MapReduces map lead to a performance bottleneck. To maintain the desired throughput, these pipelines had to be optimized by hand. Chambers et al. therefore introduce a pipeline orchestration framework called FlumeJava. The framework builds on top of MapReduce, but offers the higher-level abstraction of a pipeline. Pipelines are constructed using the combination PCollections, an abstraction for parallel collections, and related operations. Prior to execution, the pipeline is automatically optimized (e.g., by fusing different operations into a single execution stage), thereby assuring efficient execution.

These early batch systems rely on the assumption that the input data is bounded, i.e., finite in size. For example, the sorting operator in MapReduce has to read the entire input collection before it can produce any output [32]. As a consequence, these systems always introduce latency inherent with collecting all the input data. Many real-world datasources are in fact unbounded in size, i.e., it is unknown a priori when all the data is received. An common approach taken in practice was to artificially divide unbounded datasets into finite chunks for processing [22].

The latency introduced by collecting all the input data is not feasible for many use-cases. As an example, consider detecting fraudulent users on a web-service. Detecting such fraudulent users near real-time is paramount to be able to take appropriate measures, e.g., blocking the user from accessing the service. Apache Storm [52] was the first streaming system that was able to meet these low latency requirements. This came at a price, as the system essentially traded strong consistency guarantees for lower latency. As a result, the output produced by Storm computations was not subject to any guarantees. To offer both low latency and strongly consistent results, batch and streaming approaches were combined in the Lambda Architecture [53]. The general idea behind this architecture is that a streaming system processes the latest data to produce low-latency, inaccurate results. Additionally, a periodical batch job is scheduled, on the entire input collection, to yield the correct result. The problem with this setup is that the batch and streaming layers required different code bases that had to be maintained independently, yet produce the same results [47].
Apache Spark [65] introduced yet another abstraction, namely the Resilient Distributed Dataset [64]. In this abstraction, fault tolerance is achieved through the notion of lineage. The general idea of lineage is that the operations on the data are maintained, thereby allowing intermediate results to be recalculated in the face of failures. Spark initially started as a batch processing system, but its characteristics made it possible to schedule multiple batches after another to process streaming data. This approach to processing unbounded data was referred to as microbatching in literature. Spark provided a means to process unbounded data with low latency results, while simultaneously offering strong consistency guarantees. The system is therefore considered a huge milestone in the field. The main drawback is that Spark only provided the means to process in-order arriving data, thereby allowing only use-cases to be addressed that do not care about the actual event-time of the data.

MillWheel [24] was introduced to be able to address out-of-order arriving data, with strongly consistency guarantees and low latency. The system provided the first, general-purpose stream processing architecture. Shortly after, the Dataflow Model [22] was introduced. The programming model aims to unify batch and stream processing, as these approaches to processing data are intimately related. The model allows the programmer to make explicit tradeoffs regarding completeness, latency and cost. The project was later donated to the Apache software foundation, which in turn gave rise to Apache Beam [1]. Apache Beam is different from the other systems, in the sense that it is only a programming model. In fact, it is a portability framework for data-parallel pipelines. Pipelines are described using a domain-specific language and once written, they can be executed on any supporting execution engine. To date, there are a number of supporting execution engines such as Google Cloud Dataflow [14], Apache Flink [29] and Apache Spark [65].

Google Cloud Dataflow [14] is a fully managed, cloud-based solution for executing Beam pipelines. Running an on-premise cluster for stream processing requires the cluster to be provisioned for the expected peak workload. Under-provisioning leads to increased latency, whereas over-provisioning incurs unnecessarily costs. Thanks to the cloud-based nature of the engine, Dataflow is able to provide a feature called auto-scaling. Autoscaling allows for the number of worker machines to be changed dynamically based on the runtime conditions of the pipeline [36].

### 1.2 Marketing Automation

Selligent Marketing Cloud serves over 700 organizations active in a wide variety sectors, established in over 30 countries. Traditional marketing approaches, such as physical mailing, allowed little or no personalization. Online marketing, on the other hand, provides endless opportunities for two-way communication with the consumer. Reporting and analysis is key to understanding the consumer and supports the marketeer in per-
sonalizing the interaction with their audience. Given the fact that companies usually target millions of consumers simultaneously, providing a one-on-one personalized experience is a very non-trivial endeavor. To further complicate the matters, marketeers are longing for real-time insights in the data, as these insights are leveraged to guide business decisions. The main challenge is that interaction data is a highly spiky workload. The input rate of the data exhibits major peaks at, for example, Black Friday. While Selligent has been able to successfully provide insightful reporting functionalities, the company is longing for a future proof solution that provides increased scalability. Moreover, these functionalities are limited in the sense that they focus on a single reporting dimension. In reality, however, the interaction data can be used to deliver insights in multiple dimensions. Additionally, due to the fact that e-mail has historically been the most prevalent communication channel, reporting is mainly e-mail centric. The existing solution can thus be improved to address new use-cases that allow the marketeer to create additional business value.

1.3 Research Goal

This thesis studies how the advancements in the field of large scale, distributed data processing can be leveraged in marketing automation. Due to the vast landscape of big data technologies, the thesis focusses mainly on the technologies offered by the Google Cloud Platform. The Google Cloud Platform is particularly interesting as it allows for quick iteration and testing of the various reporting scenarios. The contribution of this thesis is two-fold:

1. The first, academic contribution is a survey in latest big data technologies and associated concepts.

2. The Selligent challenge is to build a modern marketing reporting pipeline using these technologies.

Specific requirements for this pipeline are scalability, availability, low-latency results and the ability to serve multiple tenants. Moreover, the pipeline should be able to deliver insights in multiple dimensions of the interaction data. Additionally, the interaction data is used to create a new feature devoted to support the marketeers in maximizing the impact of their marketing campaigns. The concrete problem of send-time optimization is studied, i.e., the problem of establishing the the most appropriate time to approach each individual consumer, based on historical data.

The thesis is organized as follows, Chapter 2 introduces the Selligent Marketing Cloud and the features provided to realize valuable interactions at scale. Chapter 3 examines the data models that are most comprehensively studied in literature. Chapter 4 provides
a study in specific data technologies, in particular, the technologies offered by the Google Cloud Platform. General stream processing concepts are covered in Chapter 5, whereas Chapter 6 zooms in on the specifications and concepts of the Beam model. Understanding how pipelines are executed is key to characterizing the correctness of the produced output. To this end, Chapter 7 studies pipeline execution in the context of Google Cloud Dataflow. As a proof of concept, multiple approaches to send-time optimization were implemented. Chapter 8 discusses these various implementations and argues about their correctness. To assess the performance characteristics of these implementations, a series of load tests were performed. Chapter 9 provides an exhaustive overview on the test cases performed, along with the most significant findings. Chapter 10 introduces a number of interesting reporting use cases and discusses the implementation of the end-to-end reporting application.
Chapter 2

Marketing Insights

The main purpose of this chapter is to introduce the terminology and concepts used throughout this work. In particular we explore the concepts required to answer the questions of what, who, where and when in the context of online marketing. Next, we describe the common thread throughout this work, namely an interaction. Hereafter, we introduce the Selligent Marketing Cloud, a marketing platform designed to create valuable interactions at scale. Additionally, the tools that the platform offers to this end are examined. On this follows a deep dive in the components that underpin the platform. Finally, the shortcomings in this current architecture are identified and evaluation criteria for an improvement version are enumerated.

2.1 Online Marketing

Online marketing is the process of promoting products and services through the internet. As opposed to traditional marketing tactics, online marketing provides opportunities for two-way communication with the customer. This encourages positive relationships, improves consumer satisfaction and yields competitive advantage. Launching campaigns online allows for a specific group of consumers to be reached nearly instantaneously. Moreover, online marketing is not subject to opening hours, i.e., it never closes. This creates endless possibilities for engagement and revenue opportunities.

2.2 Concepts

Companies may launch marketing campaigns for a widespread of reasons. For instance, to increase the visibility of products and services, to connect with customers, or to cope
up with the competition. Regardless of the goal, there are essentially four intimately related core concepts to consider. Together, these concepts address the questions of what, who, where and when respectively. Consider the following definitions:

1. **Content**: the messages that are sent over the various communication channels to the consumers. These messages are most commonly created from a template.

2. **Consumer**: the person that is targeted by a company. Consumers are often accommodated in groups, referred to as audience segments.

3. **Channel**: the communication medium used to send content to a person, i.e., the medium used to approach the consumer.

4. **Time**: the moment in time the content was sent to the consumer via the medium.

To appreciate how these concepts relate, refer to the schematic overview provided by Figure 2.1. It is noteworthy that interactions must not necessarily be initiated by the company. Interactions might also be recorded as a consequence of an action taken by a consumer, e.g., opening an e-mail, visiting a web page, purchasing an order, and so on. Section 2.4 provides a more in-depth discussion on this matter. A specific instance of each of these concepts, is hereinafter collectively referred to as an interaction. For a marketing campaign to have the desired result on the consumer, the content has to be both personal and relevant. Moreover, it has to be sent through an appropriate communication medium in a timely manner.

**Figure 2.1**: Visualizing the relations of the core marketing concepts. Content is delivered to a consumer through a communication channel, at a particular moment in time.

Given the fact that companies usually target millions of consumers simultaneously, providing a personalized experience is a very non-trivial endeavor. In particular, consumers usually have very different interests, and not everyone is as easy incentivized for interaction with the company. Additionally, the channel used to approach the consumers may influence their reaction, and not all consumers can be approached using the same
CHAPTER 2. MARKETING INSIGHTS

communication medium. Specific consumers may be reachable via e-mail, whereas others may only respond to push notifications. To further complicate the matters, content has to be optimized for the devices used to view the content. For instance, one group of consumers may read e-mails using a smart-watch, while others use their laptop.

2.3 Selligent Marketing Cloud

Selligent Marketing Cloud is a marketing platform that automates end-to-end online marketing. The goal of the platform is to provide a means to realize valuable interactions at scale. Selligent Marketing Cloud offers robust omnichannel marketing solutions. This empowers companies to launch integrated campaigns and manage consumer interactions seamlessly across all communication channels. The following sections describe how Selligent Marketing Cloud allows for the questions of Section 2.2 to be addressed. While the platform offers a very broad toolset, the sections focus exclusively on the subset that is relevant in the scope of this thesis.

In the interest of making further discussions less ambiguous, let us establish some more profound definitions. First of all, the term tenant is reserved to refer to a company that relies on the marketing platform. Moreover, tenants are able to maintain multiple distinct brands in the same account. These different brands are referred to as organizations. In the context of a campaign, the group of consumers targeted by the organization is called an audience.

2.3.1 Content

Personalizing each message manually is unfeasible given the extensive audiences that companies wish to target. Instead, the messages in Selligent Marketing Cloud are created from a template. The template contains the company’s branding, and defines the message on a high level using a custom templating language. As illustrated by Figure 2.2, the language allows for accessing data to be used in the message, e.g., personal data, interaction history, product information, and so on. When the campaign is effectively launched, and hence send to consumers, the template is enriched with consumer specific data. This results in a message that is tailored to the consumer at hand.

Example 1. A company delivers a weekly newsletter to its customers. In the interest of creating a more personalized experience, the company wants to greet each of the consumers by their own name. To this end, the marketeer of the company creates a template that defines the newsletter. To personalize the salutation, it is equipped with

\[1\] The distinction between tenants and organizations does not serve a practical use in our study, and the terms are therefore used interchangeably.
CHAPTER 2. MARKETING INSIGHTS

Figure 2.2: Creating personalized messages from a template content based on personal data, product information and interaction history.

Templates may optionally contain call-to-action (CTA) links as visualized by Figure 2.2. These are an explicit call to the reader of the content to take a specific action. For example, to purchase a product, to fill out an online survey, or to visit a web page.

To yield more relevant and valuable interactions, Selligent Marketing Cloud enables marketers to leverage the power of machine learning in their marketing campaigns. This is done through a feature that is referred to as smart content. The functionality of smart content is encapsulated within so called smart blocks. These blocks can be used in a template and learn from the consumers’ past behavior to the end of making highly relevant content recommendations. For example, they may recommend products to consumers, remind them about pages they have visited on the website, or provide content about subjects they have recently read about. This facilitates the process of creating and refining content that is personal to the consumer.

Content can alternatively be created using a responsive design editor. This WYSIWYG² approach of content creation allows for immediate visual result. The responsive design

²This is an acronym for “what you see is what you get.”
editor also offers functionalities to emulate specific devices, this allows the marketeer to optimize the content for specific devices.

### 2.3.2 Consumer

The content that a company wants to deliver is not necessarily relevant for all of its customers. For example, some ads may only be relevant for males, whereas others may only be relevant for unmarried females. Selligent Marketing Cloud therefore allows for the audience to be segmented in fine-grained audience segments. These segments allow the marketeer to send relevant content to a specific group of consumers. There are essentially two prevalent types of audience segments to consider:

1. **Dynamic segments**: Dynamic consumer segments are created by constraining the data made available by the consumer. These constraints can be created by the filter designer. The filter designer is a tool to refine specific segments such as restricting the audience segment to consumer who have, for example, been identified as male. Dynamic consumer segments are automatically updated as new consumers meet the constraints or as the consumer’s data changes and the constraints no longer apply. This is especially useful because it guarantees that the constraints defined on the profiles within the segment are always satisfied.

2. **Static segments**: provide a snapshot of customers at a particular moment of time. The segmentation strategy is based on an external file and is particularly useful when engaging with customers that are managed in an external system such as an event ticketing system. The selection of the users within the segment is done by comparing the customers in the audience list with records in the external text file or CSV file. If a user in the external file matches a record in the audience list, it is added to the segment. This type of segment, in contrast to the dynamic segment, is not automatically maintained.

### 2.3.3 Channels

Selligent Marketing Cloud offers an omnichannel marketing platform. This implies that content can be delivered to consumers regardless of the communication channel that they use. Possible channels include text-messages, website customizations, push notifications and emails. For each communication channel, the marketeer can rely on a variety of built-in design tools to create the content to be distributed.

---

3Alternatively, static segments can be created from the result of a query or filter expression.
Email correspondence    Email correspondence is the most prevalent communication channel. This is justified by the fact that emails can be highly customized to the needs of the company’s marketing department. To create email content, the marketeer can simply import an HTML template and send it to the target audience.

Website customizations    Webpages allow for a wide variety of interactions with the consumer. They can be used as landing pages, display personalized banners, home a survey to learn more about the customer, or as a variety of other marketing applications designed to engage the customer to interact with the company. Moreover, when entering the world of webpages, tracking mechanisms become available to learn even more about the consumers’ behavior.

SMS and push notifications    Engaging the customer, wherever they are, can be done by displaying mobile messages. Push notifications can be configured for users that are using the company’s mobile applications.

2.3.4 Time

The moment in time at which the audience is approached greatly influences the impact of the marketing campaign. The time used to approach the consumer is hereafter referred to as the send time. Monitoring when mails are opened, or when consumers respond to call-to-actions, yields a very rich source for information extraction. In particular, the data can be analyzed to deliver valuable insights on the relationship between the send time and the actions that follow. These insights may be exploited to define general guidelines for when the audience is best approached. For example, it makes more sense to deliver a mail during the week than in the weekend. Especially during office hours, the probability that a consumer opens a message is generally higher. Conversely, sending a mail during the night is a no-go, as consumers tend to sleep with their smartphones and might be awakened by the notification.

Instead of launching campaigns instantaneously, Selligent Marketing Cloud allows for campaigns to be scheduled at a predefined moment in time. Deferred launch equips the marketeer with a tool to make explicit data-driven decisions. The main problem is that the optimal send time is highly specific to the consumer. As such, for most audiences, it is intractable to infer a set of rules to determine the optimal send time. Ideally, the platform should be able to determine the optimal send for each individual consumer automatically. Establishing the optimal send time for each individual consumer is the main subject of Chapter 8.
2.3.5 Journeys

Organizations commonly launch a marketing campaign to incentivize consumers to take a specific action. For example, to like or to react on a post on social media, to share a web page, or to buy an article on the website. Satisfying such goals often requires not one, but multiple interactions with the consumer. This is highlighted in the example below.

Example 2. Consider the concrete example of a traditional clothing retailer that aims to make their digital transition. Incentivizing a user to buy an article on a website requires a sequence of interactions to take place over time. Assume that the consumer in question already made purchases in the physical store of the organization. First, the user has to find out that the website even exists. An interaction that promotes the visibility of the website is thus the first step in the sequence. Once the consumer visits the website, the website has to present relevant articles. Labeling relevant articles for the consumers at hand requires examination of their purchase history. Further interactions are recorded as the user navigates through the website and visits specific products. To realize the purchase, the company may have to remind the consumer of these products over time. If this all fails, the service may send a coupon to finally convince the user.

In marketing terminology, the desired action is called a conversion. Similarly, when a consumer performs the specific action, the consumer is referred to as converted. Moreover, the interactions required to reach the conversion, is called the conversion path. Managing the conversion path for each individual consumer manually is like looking for a needle in a haystack. Instead, Selligent Marketing Cloud provides the notion of a journey. Journeys are leveraged to form sequences of interactions. Specifically, they act as a roadmap for moving consumers through interaction with the organization. The journey determines how and when messages are delivered to the audience and what happens after they are received. Figure 2.3 presents a highly simplified representation of a journey. The journey is started by means of a triggering mechanism. Triggers that are encountered in practice are, for example, weekly triggers, or triggers that fire when a new consumer registers an account on the website. When the trigger fires, a specific action is executed. For our purposes, it suffices to think of an action as sending a message. For flexibility, the action maintains a reference to the template of the message, this allows for the template to be changed during runtime. There exist numerous of other components designed to the end of create more advanced workflows. A prevalent example is the filter, which can be used to place additional constraints on the consumers in the audience. Journeys are created from scratch using a drag-and-drop interface.

Additionally, Selligent Marketing Cloud provides standardized templates for the types of journeys that are most commonly encountered. The following paragraphs introduce the the different templates offered, and describes their relevance by means of an example.
Figure 2.3: Highly simplified representation of a journey. The journey is started by a trigger and has the result that an action is executed. Additional components, such as filters, can be used to create more sophisticated workflows.

**Single Batch**  Single batch journeys are designed for non-recurring, single messages that are delivered to all consumers in an audience segment. Typical use cases for single batch journeys include contest invitations or newsletters.

**Example 3.** An organization wants to send an invitation to each consumer that has been using their service for at least one year. To this end, the organization creates an audience segment consisting of consumers that meet this criterium. Next, the organization creates a single batch journey dedicated to this segment.

**A/B Messaging**  This type of journey offers the ability to send test variations of messages to optimize interaction. The marketeer determines which test version is delivered to which subset of consumers and how and when the performance is evaluated. The performance is measured based on the evaluation metrics described in Section 2.6.1.

**Example 4.** In the interest of establishing which version of an email subject line gives the most traction, an organization creates an A/B messaging journey. The marketeer adds the two variants of the content and sends it to different subsets of the target audience.

**Recurring Batch**  Within recurring batch journeys, the same message is delivered to an audience segment at a regular interval for a fixed period of time. Use cases for recurring batch journeys include product reminders and discount emails.

**Example 5.** Every week, on Monday, a journey is executed that selects all upcoming birthdays for the week and sends a coupon code to respective consumers.
**Transactional Journey**  Transactional journeys allow for messages to be send as a result of an action taken by a consumer. For instance, when the user signs up on the website or checks out a purchase.

**Example 6.** To increase the consumer experience, an organization wants to deliver a welcome mail once a consumer registers an account on their service. As such, the marketeer sets up a transactional journey that forwards a welcome mail in the face of a new registration.

### 2.4 Outbound versus Inbound Communication

Selligent Marketing Cloud deals with two distinct types of communication streams, each exhibiting very different characteristics. The first stream is outbound communication, i.e., the stream wherein tenants interact with their consumers. This is by definition a very bulky workload, due to the fact that companies try to reach potentially millions of consumers simultaneously. On the other hand, there is inbound communication. Recall that interactions are not necessarily initiated by the company, but may also take place on behalf of the consumer. These interactions constitute to the inbound communication stream. In particular, when a consumer approaches the tenant, e.g., the consumer visits the website, opens a mail or places an order. This workload is always dedicated to a specific consumer, as illustrated by Figure 2.4.

![Figure 2.4: Visualizing outbound and inbound communication.](image)

Recall the different types of journeys introduced in Section 2.3.5. Single batch, A/B Messaging and Recurring batch provide means to engage in outbound communication. Conversely, the transactional journey allows the organization to anticipate to interactions in the inbound communication stream.
2.5 Data Model

The granularity of the consumer data available has a great impact on the effectiveness of a marketing campaign. Specifically, understanding how customer data is stored and organized is key to targeting specific groups of customers and fully personalize their interactions with the organization. Selligent Marketing Cloud follows a so called star-model \( \square \) where the core profile has a central spot. Additional information is linked to the profiles as can be seen in Figure \( \ref{fig:2.5} \).

![Diagram of data model](image)

**Figure 2.5:** Structuring consumer data.

The central building block, in the context of customer data storage, is the audience list. This audience list is housed in the Consumer table and captures data that is changing at low rate. Within this table profile information is stored, this includes data such as the name, email, city and state of the consumer. Every consumer record in this table is augmented with a unique identifier. The audience list contains a number of standardized fields but is flexible in the sense that it allows the consumer profile to be extended with custom fields. There is an inherent trade-off between fields that are standardized and custom within the customer table, as we see in the following example.

**Example 7.** Two businesses, A and B determine that the street and number of the consumer should be part of the consumer profile. Since these fields are not part of the standardized data model, they decide to augment the consumer table. Company A includes a single field containing the concatenation of the street and number, while B included distinct fields for the attributes.

The presence of standardized fields allow Selligent to make certain assumptions on the
semantics of the data model. This allows for better and more in-depth reporting (Section 2.6). However when dealing with custom fields, this is not trivial as suggested by Example 7. Therefore as much data as possible should be standardized while offering a sufficiently flexible data-model for the organization.

Rapidly changing information—such as preferences, orders and subscriptions—is housed in separate tables. These tables are linked in a star-like pattern using the aforementioned identifier. There is a distinction to be made for these tables based on their relation to the customer table. Tables that relate to the consumer in a one-to-one fashion are called profile extensions. Tables that link to the consumer in a one-to-many manner are called lookup lists. Profile extensions can be used to personalize the messages that are delivered to the consumer directly, while lookup lists can be used to retrieve information about the customer, for example their last order. Kleppmann [42] refers to the customer table as the *fact-table*, while the remaining tables are called *dimension tables*.

### 2.6 Reporting

As introduced in Section 2.2, launching an effective marketing campaign requires for the questions of who, what, where and when to be answered. Selligent Marketing Cloud provides a rich set of features devoted to address these questions, i.e., it contributes to *how* valuable interactions are created. Launching a campaign is only one side of the coin, as marketeers are longing for insights in the performance of their campaigns.

Selligent Marketing Cloud maintains an explicit record of all the interactions that occurootnote{Selligent Marketing Cloud captures interaction data using sophisticated tracking mechanisms or relies on the tenant to explicitly report the interactions.}. Based on this record, the platform provides predefined reports that support the marketer in understanding the results of their marketing strategy. Reports are presented in the form of a dashboards that map out the effectiveness of the campaign using tables, graphics, and images. Additionally, the performance of the campaign is quantified using specialized evaluation metrics. Section 2.6.1 provides an overview of the most prevalent metrics. Further analysis can be done on behalf of the marketer to identify patterns in the interaction data. In the interest of making interactions more valuable, these patterns may be employed to make better, faster and more fine-grained decisions. The idea is that these decisions ultimately translate to increased company business value.

For example, the reports can be used to identify which communication channels are best suited for the audience segment at hand. Moreover, insights in the audience segment quality and growth can be obtained. Reporting also gives an overview of which devices are used to view the messages and offers support to identify the best types of offers to incentivize clicks on the call-to-action links.
Simultaneously, the interaction data is leveraged to create additional functionalities that allow for the who, what, where and when to be answered more precisely and accurately. The smart content introduced in Section 2.3.1 is an example of one of the data-driven features offered by Selligent Marketing Cloud. The dual use served by the interaction data is emphasized by Figure 2.6.

![Figure 2.6: Reporting value diagram. Interaction data is tracked to provide predefined analysis and reports, thereby allowing marketeers to create added value. The data is simultaneously leveraged to create new internal features devoted to support the marketeer in answering the core questions more precisely and accurately.]

### 2.6.1 Evaluation Metrics

To quantify the performance of the marketing campaign, numerous evaluation metrics have been introduced in the literature [35]. The following paragraphs present an overview of the most prevalent metrics, along with a concrete example.

**Open Rate** The open rate is the percentage of receivers of a message that actually decide to open it. One factor to positively influence the opening percentage for e-mails is choosing a catchy subject line. When a campaign targets 750 people and 180 of them opened, the open rate would be 24%.
CHAPTER 2. MARKETING INSIGHTS

**Click Through Rate (CTR)**  The CTR is defined as the ration between the number of click through a message and the number of deliveries thereof. The metric is most commonly defined as a percentage. For instance, if a email is clicked as often as it is delivered, the click-through ratio is 100%. Similarly, if a email was delivered to 1000 consumers and has been clicked 10 times, it has a CTR of 1%.

**Click to Open Rate (CTOR)**  CTOR stands for the click-to-open rate and compares the number of people that opened the email to the number that actually clicked. In contrast to the CTR, which is expressed using deliveries, this metric measures the performance of the content of the email.

**Bounce Rate**  In the context of web-based content, the bounce rate is the percentage of visitors who only viewed a single page when visiting a website. In particular, they left the website immediately, without taking any further action. A high bounce rate may indicate that the visitors directly found the information they were looking. However, in reality, this is usually the opposite. In the world of e-mailing, the bounce rate refers to the percentage of email addresses in your subscriber list that did not receive the message because it was returned by a recipient mail server. In both cases, bounce rate should be minimized.

**Conversion Rate**  The conversion rate is the number of conversions divided by the total number of consumers reached. For example, if an e-commerce site receives 200 visitors in a month and has 50 sales, the conversion rate would be 50 divided by 200, or 25%. As already described, a conversion can refer to any desired action that you want the user to take. This can include anything from a click on a button to making a purchase and becoming a customer. Different channels often have multiple conversion goals, and each will have its own conversion rate.

### 2.6.2 Reporting Dimensions

The interaction data captured by Selligent Marketing Cloud offers a very valuable source for information extraction. In particular, it allows for information to be inferred in regard to many different dimensions. Prevalent dimensions, in the context of online marketing, are as follows:

1. **Content**: Are consumers reading the content? Does the content incentivize interaction with the organization? Is the content readable on all the devices?

2. **Consumer**: Is the consumer open to interaction? When is the consumer best approached? Which channels can be used to approach the consumer? Which
CHAPTER 2. MARKETING INSIGHTS

touch-points were established with the consumer? What is the customer lifetime value?

3. **Audience**: How geographically spread is the audience? How did the audience evolve over time? What is the fraction of consumers, in the audience, that can be reached via a specific channel?

4. **Channel**: How popular is the channel? How many consumers are reachable via the channel? What is the open rate for the channel?

5. **Time**: How does time influence the open rate for a message? At what time of the day are the most people active?

6. **Journey**: Is the journey having the desired effect on the consumers? How many consumers did the journey reach? What is the conversion rate? Are there bottlenecks in the journey? What is the total of revenue gained by the journey?

### 2.6.3 AS-IS Architecture

The AS-IS architecture was examined in the interest of mapping out potential shortcomings. The section has been removed due to the fact that it disclosed confidential information.

### 2.6.4 Shortcomings in the Current Architecture

While the shortcomings in the current architecture were thoroughly identified, this section has been removed due to the fact that it disclosed confidential information.

### 2.6.5 TO-BE Architecture

The new reporting pipeline has to be flexible, scalable, available and be able to deliver low-latency results. It should be flexible in the sense that the data can be analyzed in any way desired. Specifically, the pipeline should be able to deliver insights in multiple dimensions, particularly those identified in Section 2.6.2. The correctness and completeness of the results are paramount. Incorrect or incomplete results may, after all, result to mistrust in the system. The term scalability can refer to various dimensions, namely the number of customers, the number of consumers, the number of channels and the number of journeys. The system should thus be able to cope transparently with increasing numbers in each of these dimensions. The availability of the pipeline is measured on how well it deals with different types of crashes, gracefully failing over in a way that is unnoticeable to the end users. Latency is a time-based measure of the performance
of the system, ideally, the latency is minimized. Moreover, a cost effective solution is a requisite, the pipeline should be cheap to build, operate and maintain. In this light, hosted technologies are preferred because there is no capital expenditure required on the companies’ behalf for the underlaying infrastructure. While at the same time, these technologies allow engineering teams to focus on application development, instead of spending their precious time managing infrastructure.
Chapter 3

Data Models

Data storage and retrieval is central to a data processing system. This chapter delves into the data models and structures that underpin modern data storage technologies. The choice of distributing data across multiple machines may be justified by the following requirements [42]:

1. **Scalability** when the read load, write load or extend of the data exceeds the capabilities of a single machine, the load can be distributed across multiple machines.

2. **High availability** for certain services, it is unacceptable for the entire application to go down in the face of a system crash. Multiple machines may be used to provide redundancy. That is, in the face of a failure, a redundant system may failover.

3. **Latency** if the application is available at global scale, servers may be distributed geographically to reduce latency introduced by network round trip time.

### 3.1 OLAP versus OLTP

Database systems are usually designed with a specific type of workload in mind. Traditional applications, such as point-of-sales systems and social networks, roughly exhibit the same database access pattern. These applications typically look up a small number of records by some key and insert or update records based on user input. This access pattern is referred to as *online-transaction-processing* (OLTP). The main emphasis in OLTP systems is put on very fast query processing, maintaining data integrity and maximizing transaction throughput.

Nowadays, databases are increasingly used for analytic purposes whereby data is accessed in a distinct pattern. Queries of analytic nature read a large portion of the records
CHAPTER 3. DATA MODELS

contained in a table while only a few number of columns per record are considered. Such queries produce aggregate statistics in contrast to the raw data returned by transactional queries. This access pattern is called *online analytic processing* (OLAP). For these systems, response time is an effectiveness measure. A relevant analytic query, in the context of Selligent, may calculate an average open-rate for each customer in the audience list. Typical characteristics for OLAP and OLTP systems are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Property</th>
<th>Transaction processing systems (OLTP)</th>
<th>Analytic processing systems (OLAP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main read pattern</td>
<td>Small number of records per query, fetched by key</td>
<td>Aggregate over large number of records</td>
</tr>
<tr>
<td>Main write pattern</td>
<td>Random-access, low-latency writes from user input</td>
<td>Bulk import (ETL) or event stream</td>
</tr>
<tr>
<td>Primarily used by</td>
<td>End user/customer, via web application</td>
<td>Internal analyst, for decision support</td>
</tr>
<tr>
<td>What data represents</td>
<td>Latest state of data (snapshot)</td>
<td>History of events that happened over time</td>
</tr>
<tr>
<td>Dataset size</td>
<td>Gigabytes to terabytes</td>
<td>Terabytes to petabytes</td>
</tr>
</tbody>
</table>

Table 3.1: *OLTP versus OLAP system characteristics* [42].

Selligent Marketing Cloud serves both transactional and analytical workloads. Inbound marketing (Section 2.4) is primarily transactional, whereas reporting procedures (Section 2.6) are of analytical nature.

3.2 Relational Model

The relational data model is currently the the most popular data model employed in traditional database systems. Relational databases use relational query languages, which are based on the relational algebra, or equivalently, on the first-order relational calculus, both introduced by Codd [37].

In the relational model, data is organized into *relations* and each relation is an unordered collection of tuples. In the early days of data processing, developers were forced to consider the internal representation of the data residing in the database. The queries that were written to read data from these early database systems were typically imperative. This imperative style is opposite to the declarative query languages, such as SQL. The purpose of the relational model was to hide these implementation details behind a uniform interface. Declarative programming [62] is a programming paradigm that expresses
the logic of a computation without explicitly describing its control flow. The focus is placed on describing the structure of the data of interest, rather than describing how the data should be computed. The data structure used to power most relational databases is the B-Tree. This data-structure is a self-balancing tree that maintains sorted data and allows lookups, range queries, insertions, and deletions in logarithmic time [37].

3.2.1 Normalization

Normalization is a systematic approach of decomposing tables to minimize data redundancy. Normalization also has the additional benefit that undesirable characteristics such as insertion-, update- and deletion anomalies are eliminated [37]. In a normalized relational schema, disk wastage is prevented. The decomposition of the relation $R$ involves splitting the attributes of $R$ to construct the schemas of two new relations. Entities of the new relations are linked using a foreign key, thereby allowing the original relation to be reconstructed. To reconstruct the original relation, the join operator [37] is leveraged. Normalization is carried out in multiple steps that each put the data in their relational form, thereby removing duplicated data from the relation tables in the process [37].

Example 8. Consider for example the setting of a social network. Users maintain a profile containing information about themselves, post status updates and establish friendships. Expressing such data in the relational model involves multiple tables as result of normalization. Figure 3.1 visualizes the resulting model graphically. There are distinct relations for the users, posts, friends and countries. In order to present the complete profile for a user, including their posts and friends, one has to consult multiple tables and perform a series join operations.

One of the challenges in the relational paradigm is that normalized models are generally not feasible for real-world requirements [59]. This is due to the overhead produced in the query processor by an over-normalized structure. For many production systems, a normalized schema must in practice be further adapted for specific application needs. More concrete, to make the relational model perform well enough for real-world scenarios the true domain affinity has to be abandoned and the designer has to accept the fact that the data model has to be tuned to suit the database engine at hand. This technique is called referred to as denormalization [59]. Denormalization is thus the inverse process of normalization where data redundancy is added in favor of increased performance. It can be conceptualized as a technique in which the result of a join of tables in the normal-form representation is stored as a base relation. This way of working inherently introduces data redundancy, as such, additional work is required to keep the denormalized data consistent. This is illustrated by the following example.

Example 9. As discussed, to retrieve a complete profile for a user, a series of join operations are executed. To tune the performance of the database, a possible step could
Figure 3.1: Modeling a social network in the relational model.
be to denormalize the user and country tables, thereby explicitly storing the country name as an attribute of the user. This is illustrated by Figure 3.2. While it may be very uncommon, updating the name of a country requires now more work as all its instances need to be tracked and updated accordingly.

3.2.2 Column-Oriented Storage

In most relational databases systems, storages is organized in a row-oriented fashion, i.e., values of different attributes from the same tuple are stored together. This is especially efficient for OLTP workloads, where per-record operations are the most common. A column oriented database system, on the other hand, is one in which each attribute is stored in a separate column. This way, successive values of that attribute are stored consecutively on disk.

Column stores do not increase the expressiveness of the data model, rather its performance on analytical workloads is generally increased [42]. This is mainly due to the fact that most analytical queries compute aggregates over the records of a column. This is illustrated by the example below.

Example 10. An online webshop maintains a table that contains purchase information. Assume for simplicity, that the table contains a single column that represents the yield for each purchase. A canonical example of an analytic query, in this setting, is computing the total revenue. Answering the query involves computing a sum over all the rows in
this table. When data is stored in a per-record fashion, the values from the column are generally distributed across the disk. As a result, multiple expensive disk I/O calls are required to fetch the values. Storing data in a column oriented fashion minimizes the I/O calls required.

Moreover, storing data in a column-oriented fashion considerably increases the similarity of subsequent records on disk, thereby offering more opportunities for compression. Indeed, column stores are well suited to compression techniques that compress values from multiple rows simultaneously. Run-length encoding is one of such algorithms; sequences of the same element are expressed as (value, run-length) pairs. This algorithm is particularly useful when the data is maintained in sorted order [21].

3.3 NoSQL Landschape

NoSQL databases are driving by the need for greater scalability than relational databases can archive, specialized query operations that are awkward in the relational model or frustration with the restrictiveness of relational schemas and the desire for a more dynamic and expressive data model. In the following chapter, we study the most prevalent types of NoSQL data models.

Another argument in favor of NoSQL databases is the object-relational mismatch [42]. When the data from object-oriented programming languages is stored in tables, translation is required between the objects in the code and relational database model. Object-relational mapping techniques (ORM) are often used to overcome this translation layer.

3.3.1 Key-Value Stores

A key-value store is a data storage model designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash table [42]. Dictionaries maintain a collection of objects, or records, which in turn contains many different fields, each containing data. These records are stored and retrieved using a key that uniquely identifies the record, and is used to quickly locate the data within the database.

Key-value stores operate very differently from the better known relational databases. The relational model pre-defines the data structure in the database as a series of tables, each containing fields with strictly defined data types. Exposing the data types to the database program allows it to apply a number of optimizations. In contrast, key-value systems treat the data as a single opaque collection, which may have different fields for every record.
Key-value databases can use consistency models ranging from eventual consistency to serializability [42]. Implementations may also support explicit ordering of keys. Moreover, some stores maintain data in-memory, while others employ persistent storage.

### 3.3.2 Document Databases

Document databases can achieve better performance by exploiting locality and are particularly suited for semi-structured data [42]. Central to a document-oriented database, is the notion of a document. While vendors differ on the details of the definition, documents are generally considered to encapsulate and encode data in a standardized format. Formats that are the most commonly encountered in practice are XML, YAML and JSON.

**Example 11.** Refer back to the social network introduced in Example [9]. The profile for a user is essentially a self-contained document, creating such per-user documents allows for greater use of locality. Presenting the complete profile of a user, requires solely the document for the user to be accessed. Figure [3.3] models the social network using JSON documents.

Similar to the relational model, where entities are linked by foreign keys, documents may reference to each other through a document reference. Document-oriented databases are inherently a subclass of the key-value store, as they are typically retrieved by the use of a key. The difference lies in the way the data is processed; in a key-value store, the data
is considered to be inherently opaque to the database. Document-oriented database, on the other hand, use the internal structure of the document for information retrieval. Although the difference is often subtle, conceptually the document-store is generally considered a richer experience with modern programming techniques.

3.3.3 Graph Data Model

For applications dealing with many-to-many relationships, the document model becomes less attractive. Graph databases and their representations were extensively studied in by Laurens Vijnck [63]. A graph is a collection of vertices and edges, also referred to as nodes and relationships respectively. Entities in graphs are represented by nodes, while edges represent the relationships among these nodes. The relationships in this model are directed, and therefore start and end at a particular node. The nodes and relationships in this model are often extended to contain properties in the form of arbitrary key-value pairs. Furthermore, nodes and relationships may be enriched with labels to form groups with distinct roles within the dataset. This general purpose, expressive structure is a versatile and intuitive data model to represent RDF data, social networks and many other sources of big data.

Relationships do exists in the context of relational databases, but only at modeling time in the process of joining tables. Though often, disambiguation is required on the semantics of the relationships that connect entities as well as qualify their importance. Relationships in the relational model do not offer any support for these requirements. As the overall structure of the dataset becomes increasingly more complex and less uniform, the relational model becomes burdened with large join tables, sparsely populated rows and null checking logic. The rise in connectedness translates in the relational world in an increased amount of join operations. These joins have their repercussions on the overall performance of the database system and obstruct the evolvability in response to rapidly changing business requirements [58].

Example 12. Reconsider the social network from Example 9. When modeling the social network as a graph, users and posts are represented by nodes, while edges are leveraged to represent relations. Different roles are created for the users, countries and posts of the social network. Figure 3.4 visualizes the result, the roles are made explicit using different shapes. In reality, attributes are added to a node using key/value pairs. However, for brevity of the figure, the attribute value is added on top of the label.

Relationships are first class citizens in the graph data model. i.e., data is natively stored as connected data. Most graph based database systems, such as Neo4j [58], employ a technique named index-free adjacency, whereby elements maintain direct links to their neighbors. This technique ensures that traversing the data within a graph is extremely efficient. Analogous to the relational model, there exits declarative query languages to
Figure 3.4: Modeling a social network in graph model. Differences in the roles of the nodes are highlighted by different shapes.

query graphs. A prevalent example is the Cypher query language, created for the Neo4j graph database [42].

We can conclude that, while the relational model was initially designed to handle transactional workloads, it can also be used for analytics due to the rise of column-oriented storage. However, for non-relational sources of data, other models such as the key/value stores are more appropriate. The graph model is the best approach when dealing with highly connected data.
Chapter 4

Data Technologies

This chapter provides a study in concrete data technologies to the end of making more informed choices regarding the technology stack for the end-to-end reporting application. Specifically, data storage and communication technologies are considered. The landscape is extremely vast, as such, our study is limited to the hosted technologies offered by the Google Cloud Platform. Technologies to be examined include Bigtable, Datastore, BigQuery and Cloud Pub/Sub.

4.1 Google Bigtable

Google Bigtable [31] is a distributed key/value store for managing a wide variety of both structured and semistructured data. The system is particularly designed to handle massively internet-scale datasets and has proven to reliably scale to petabytes of data across thousands of commodity machines. Products rely on Bigtable for a wide variety of workloads, each placing very diverging demands both in terms of size and latency requirements. Commonly encountered usage scenarios range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. Canonical examples of supported data types are time-series and marketing data [17]. The most noteworthy design decision made by the developers is the fact that Bigtable does not support a full relational data model, but instead provides a simplified data model that allows dynamic control over the data layout and format. This model empowers clients to reason about the locality properties of the data represented in the underlying storage. Bigtable incentivized a number of open-source projects including Cassandra [48] and HBase [5].
CHAPTER 4. DATA TECHNOLOGIES

Figure 4.1: Modeling a social network in Bigtable. This network is visualized by a table for simplicity. In reality, however, the cells with the grey background color are not materialized.

4.1.1 Data Model

Data in Bigtable is housed in a sparsely populated, distributed, multi-dimensional, lexically ordered map. Values in the map are treated as an uninterpreted array of bytes. This map, when viewed in its entirety, resembles a table consisting of rows that typically describe a single entity. Rows in turn are composed of columns which contain values for the individual attributes of the entity. In contrast to the relational model, where columns are subject to a schema, arbitrarily columns can be added and removed as desired\(^1\). The sparse approach to storing data implies that unwritten cells do not occupy space. As such defining a very large number of columns is perfectly valid, even when null values are common in most the these columns for the majority of rows [11].

For each row a single value is indexed\(^2\), this value is referred to as the row key. Constructing a row key such that common queries are facilitated is of utmost importance to the overall performance of the system. This is mainly due to the fact that efficient queries use either the key, row key prefix or a row range to retrieve the data of interest. All remaining queries trigger a full table scan, thereby sacrificing efficiency.

Example 13. To continue with the social network as introduced in Example 9, let us consider how the characteristics of Bigtable can be used to model a social network. Figure 4.1 presents a visualization, the grey background color is used to explicitly visualize the absence of a cell. While the graphical visualization resembles close affinity to an adjacency matrix, the way used to store this model is more an adjacency list style approach.

Table rows are lexicographically ordered based on the row key, clients can exploit this property by designating row keys that yield increased locality for their data accesses.

\(^1\)This implies that the names and format of the columns can vary from row to row.

\(^2\)Note that there is no such notion as secondary indices in Bigtable.
CHAPTER 4. DATA TECHNOLOGIES

Figure 4.2: Representing time line data in Bigtable. Locality of the data is controlled by designating a timestamp as the row key.

While exploiting the ordering is encouraged to improve data locality, clients should at the same time choose row keys such that read and writes are distributed evenly across the row space of the table [11].

Example 14. The fact that records are maintained in lexicographical ordered based on the row key, offers opportunities for representing timeline data. For example, in the context of monitoring statistics. Figure 4.2 visualizes this idea, a timestamp is used as a key. To group the timelines from worker nodes, de name of the node has been promoted inside the key.

Scalability issues inherent with supporting a full transactional schema are resolved by solely offering atomic, single row transactions. In this light, every read or write of data under a single row key is atomic, regardless of the number of columns involved in the process. This greatly simplifies reasoning about the engines’ behavior in the face of concurrent updates to the same row. This restricted transaction schema implies that there is no such notion of general, cross-key transactions. Consequently, schema designs that require transactions across multiple rows should be avoided [11].

Cells in Bigtable are versioned, in a sense that mutations are layered on top of the previous value, thereby providing a record of how the stored data has been altered over time. As a consequence, mutations come at the cost of increased disk usage. To remedy this fact, a garbage collection procedure is invoked periodically. The garbage collection is a process designed to compact the cells versions. By default, solely the latest version of each cell is preserved, but more advanced retention policies are available. For instance, the client can specify that only the last $n$ versions of a cell should be maintained.

The concept of column families was introduced to group together closely related at-
tributes. Column families physically colocate the columns of a row and its values, primarily for performance considerations. The number of distinct column families is assumed to be relatively small and static, that is, rarely changing over time. This is, however, no limit on the number of columns that may be included in the family. Access control and both disk access are performed at the column-family level. The column key is defined as the concatenation of the column family and qualifier, separated by a semi-colon. Addressing a specific cell in the table thus requires a combination of row key, column key and time stamp to be specified \[31\]:

\[
\text{row: string, column: string, time: int64} \rightarrow \text{value: byte[]}\]

Bigtable falls under the category of a wide column store. This category of databases must not be confused with column oriented databases as introduced in Section \[3.2.2\]. Wide column stores are databases with an ability to hold very large numbers of dynamic columns. Column-oriented storage, on the other hand, is a concept for improving the performance of the relational model for analytic workloads.

### 4.1.2 Cluster Overview

The row range for a table is dynamically partitioned, these row partitions are called tablets and form the unit of distribution and load balancing on the cluster. A Bigtable cluster consists out of a master node, along with an arbitrary number of slaves, referred to as tablet servers. Each tablet server manages a collection of tablets. Specifically, it handles read and write requests and splits tablets as they become to extensive. The master node is responsible for the assignment of tablets to tablet servers and detects the addition or expiration of tablet servers during runtime. Moreover, the master learns access patterns and adjusts the distribution of the tablets over the tablet servers accordingly. This structure differs from the traditional master-slave architecture in the sense that clients communicate directly with tablet servers. In this light, no intervention from the master node is required to serve requests from the clients. Consequently, the load on the master node is thus kept relatively low. A high-level visualization, outlining the various components that make up a Bigtable cluster, is presented in Figure \[4.3\].
A Bigtable cluster requires two external components to operate. Firstly, the nodes and storage are separate within Bigtable, resulting in a highly scalable system. In the process of adding a new node to the cluster, only the metadata specifying tablet locations is interchanged. Adding nodes to the system is therefore a linear upwards scale operation. To store the log (used to recover from failures) and data files, Bigtable relies on a heavily replicated, super-durable filesystem such as the Google File System [38]. The idea is that nodes cache the data from GFS and that at most one tablet server at the same time.

Secondly, Bigtable uses the highly-available, distributed lock service called Chubby [28]. Chubby is used for a wide variety of tasks, including leader election, tablet server discovery, tablet server expiration establishment and the storage of schema information.

4.1.3 Tablet Representation

To physically store the contents of the tablet, Google uses an hybrid data structure that was heavily inspired by the concept of an LSM-tree [57]. The persistent state of the tablet is maintained within GFS as depicted on Figure 4.4. Tablet updates are appended to a commit log that stores redo records. The most recent updates are buffered in a sorted, in-memory, data-structure called a memtable. To be able to recover from failures, these in-memory updates are logged. The older updates are stored in a sequence of SSTables. This approach results in $O(1)$ complexity for inserts, making Bigtable particularly interesting for write-heavy applications.
When the size of the memtable exceeds a predefined threshold, the table is materialized in the form of an SSTable and written to Google File System\(^3\) while simultaneously a new memtable is created. This process is referred to as the minor compaction progress, and serves as a means to shrink the memory used by the tablet server. Moreover, a merging compaction process is periodically invoked designed to merge the contents of multiple SStables into a single SSTable.

### 4.2 Megastore

Megastore \([26]\) is a document-oriented, distributed storage system designed to meet the demanding requirements of today’s interactive online services. The system is built on top of Bigtable\(^4\). While a wide variety of applications successfully use Bigtable, it became apparent that it can be difficult to use for applications that have complex, evolving schemas or those that require strong consistency in the presence of wide-area replication \([33]\). It is particularly optimized for structured, transactional, non-relational sources of data. Typical workloads include user profiles on mobile and web applications, durable session state and hierarchical data. The documents in Megastore are referred to as *entities*.

Megastore offers, in contrast to Bigtable, strong consistency guarantees within fine-grained partitions of data. These partitions are referred to *entity groups*. Data within an entity group is replicated across a wide area network in a synchronous manner \([26]\). Megastore leveraged the single-row transactions provided by Bigtable to this end. The

---

\(^3\)This is a highly-available, replicated distributed file system  
\(^4\)At the time of writing, Google is migrating Megastore to Firestore. As a result, the concepts introduced might no longer be relevant.
general idea is that Megastore serializes the entities within an entity group in the same Bigtable row. To avoid collisions in this process, the concatenation of the Megastore table name and the property name is used for the Bigtable attribute name [26]. Limited consistency guarantees are provided across these partitions. Moreover, Megastore offers extensive support for secondary and composite indexes. The fact that keys in Bigtable are maintained in lexicographical order is the primary driver for these secondary indexes. Indeed, additional Bigtable tables are constructed to offer these type of indexes.

4.2.1 Data Model

The data model used by Megastore borrows concepts from both traditional database management systems and NoSQL databases. The data is declared in a schema and is strongly typed. Schemas have a collection of tables, each consisting of a set of entities. These entities in turn contain a collection of named and typed values, called properties. A fixed set of properties is designated as the primary key for the entity. Megastore exploits Bigtables’ ability to layer multiple values over each other to implement multiversion concurrency control [37].

Tables in Megastore are either entity group root tables or child tables. Tables of the latter kind must declare a foreign key that references a root table. More precisely, each child entity references a particular entity in its root table, referred to as the root entity. An entity group consists of a root entity along with the collection entities in child tables that reference it. These entity groups allow the client to explicitly express data locality. For example, the entity groups can be used to model parent-child relationships. This is visualized by Figure 4.5.

![Visualizing an entity group](image)

**Figure 4.5:** Visualizing an entity group [12]. Entities refer to a root entity, thereby creating a hierarchy.

A megastore instance allows for multiple root tables to be defined, thereby allowing multiple entity groups to coexist. Entities within an entity group are mutated via single-
phase ACID transactions. For operations across entity groups, clients have two distinct tools at their disposal. The first, and least expensive approach is the use of a highly efficient, asynchronous communication queue. Secondly, particularly designed for operations that require atomic updates across entity groups, is the employment of two-phase commits.

![Diagram](a) Scalable, synchronous replication. (b) Operations across entity groups.

Figure 4.6: Partitioning and locality within Megastore [26].

While multi-tenancy can be achieved in Bigtable through key promotion, the notion of multi-tenancy is exposed as a first class citizen in Datastore.

### 4.3 Google BigQuery

BigQuery is a fully-managed, cloud-based, massively parallel query execution engine and warehousing solution. BigQuery is essentially an implementation of Dremel [54], launched to general availability. This externalization made it possible for organizations outside of Google to leverage the power of Dremel for their big data processing requirements.

Unlike traditional databases, Dremel is capable of operating on in situ nested data. In situ refers to the ability to access data in place. For example, data stored in a distributed file system such as GFS or another storage layer such as BigTable. This implies that BigQuery does not maintain a dedicated data structure to process queries, but accesses the data directly via the distributed file system. The idea is that, to process queries, BigQuery temporarily allocates resources in the cloud. When the result has been computed, the resources are released.

BigQuery is particularly suited for analytical workloads or business intelligence use cases that require relatively low-latency results. The cloud-based nature of BigQuery provides
extremely high full-scan query performance and cost effectiveness compared to traditional data warehouse solutions and appliances [60]. BigQuery comes with a very interesting cost-model. Instead of charging a hourly rate, users are charged for the amount of data processed by their queries. A high-level SQL based query language is provided to express ad hoc queries that are executed natively. This in contrast to map-reduce abstractions such as Pig and Hive that execute queries by translating them into MapReduce jobs.

Dremel is able to achieve this unprecedented performance by the employment of two core technologies. First, Dremel uses a column-striped storage representation, similar to Apache Parquet [6]. Storing records in a column oriented fashion considerably increases both the compression ratio and scan throughput [21]. Secondly, a multi-level serving tree architecture is used for dispatching queries and aggregating results across thousands of machines at a blazingly fast speed. This tree-architecture is outlined in Figure 4.7. Root servers are responsible for handling incoming queries. They read metadata from the tables, rewrite the query and subsequently route the query to the next level in the serving tree. Leaf servers communicate with the storage layer or access the data on local disk. On the way up, intermediate servers perform parallel aggregation of partial results.

Figure 4.7: Tree Architecture in Dremel. A query is executed using a tree, each layer merges the result computed by the previous layer until the output is complete.

4.4 Cloud Data Technologies Summary

Table 4.1 summarizes the characteristics for each of the data stores provided by the Google Cloud Platform. The provided technologies are not limited by the ones discussed in this chapter. To illustrate the bigger picture, other technologies have been included in the classification. The other technologies fall out of the scope of this thesis.
<table>
<thead>
<tr>
<th>Data store</th>
<th>Access metaphor</th>
<th>Schema</th>
<th>Multiple Indexes</th>
<th>Query Language</th>
<th>Consistency Guarantees</th>
<th>Replication</th>
<th>Transactions</th>
<th>Access pattern</th>
<th>Updates</th>
<th>Data Structure</th>
<th>Time sharding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud SQL</td>
<td>Relational</td>
<td>✓</td>
<td>✓</td>
<td>SQL</td>
<td>Strong</td>
<td>Synchronous</td>
<td>Cross row</td>
<td>OLTP</td>
<td>✓</td>
<td>B-Tree</td>
<td>✓</td>
</tr>
<tr>
<td>Bigtable</td>
<td>Key-value map²</td>
<td>API</td>
<td>Eventual</td>
<td>Asynchronous</td>
<td>Single row</td>
<td>OLAP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Megastore</td>
<td>Semi-relational²</td>
<td>✓</td>
<td>✓³</td>
<td>SQL like</td>
<td>Strong⁴</td>
<td>Synchronous</td>
<td>Cross row⁴</td>
<td>OLTP</td>
<td>✓</td>
<td>LSM-Tree</td>
<td>✓</td>
</tr>
<tr>
<td>Spanner</td>
<td>Semi-relational²</td>
<td>✓</td>
<td>✓³</td>
<td>SQL like</td>
<td>Strong⁵</td>
<td>Synchronous</td>
<td>Cross row⁵</td>
<td>OLTP</td>
<td>✓</td>
<td>Paxos FSM⁶</td>
<td>✓</td>
</tr>
<tr>
<td>BigQuery⁷</td>
<td>Columnar</td>
<td>✓</td>
<td>✓</td>
<td>SQL like</td>
<td>N/A⁸</td>
<td>N/A</td>
<td>OLAP</td>
<td>Capacitor</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenTSDB⁹</td>
<td>Key-value map</td>
<td>API</td>
<td>Eventual</td>
<td>Asynchronous</td>
<td>Single row</td>
<td>OLAP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 4.1:** Classification of data stores available on the Google Cloud Platform.

---

¹The main data-structure used to empower the database engine.
²Database engine provides the ability to express locality relationships.
³Distinction between two high level classes of indexes: local and global.
⁴Solely within the context of fine-grained partitions of data, limited consistency guarantees across them.
⁵Externally-consistent distributed transactions, achieved by means of the TrueTime API [33].
⁶Built on top of Paxos State Machines [33].
⁷Serverless data warehouse service with append-only tables.
⁸Replication is handled by the underlying distributed storage system, i.e. GFS or Colossus.
⁹Time series database hosted on top of BigTable.
4.5 Publish/Subscribe Messaging

Publish/subscribe messaging, often called pub/sub messaging, is a form of asynchronous communication. In this model, messages are not directly addressed to a receiver, but instead published in a topic. The ultimate goal of this model is to decouple the reads and writes of the messages. Receivers can subscribe to certain topics and messages published within a topic are immediately broadcasted to the subscribers to the topic. This fashion of messaging is usually supported by an intermediate component, called a broker, in which the messages are centralized [20].

The subscribers to the topic can each do something different with the message in parallel. The publisher does not keep track of who is using the information that it is broadcasting, and the subscribers do not need to know the sender of the message. This stimulates low-coupling among the various components of a system, or the services in micro-service architectures. The fact that the receiver is not explicitly defined differentiates the pub/sub model from traditional messaging-queues, in which the sender of a message needs to know the destination it is sending to. In the sections to come we map out the characteristics of two industry-leading communication systems, namely Google Cloud Pub/Sub [19] and Kafka [46].

4.5.1 Google Cloud Pub/Sub

Cloud Pub/Sub is a fully-managed, asynchronous messaging service designed to be highly reliable and scalable [9]. The service provides a low-latency system for delivering messages and comes with at-least-once message delivery guarantees. Figure 4.9 presents an
overview of the basic message flow through Cloud Pub/Sub. A publisher application creates and send messages to a topic. Subscriber applications can receive these messages by creating a subscription to the topic. These messages are persisted in a message queue until they are delivered and explicitly acknowledged by subscribers. Upon being acknowledged they are removed from the subscriptions’ queue of messages. Messages can optionally contain attributes that define certain characteristics of the message, e.g., the content type or language. The communication can be one-to-many, many-to-one and many-to-many [19]. Typical use-cases include distributing workload in network clusters, implementing asynchronous workflows, dispatching event notifications and data streaming from various processes.

Publisher

1. Message

2. Topic

3. Subscription

4. Message

5. Ack

Subscriber

307x767

Figure 4.9: Publishing messages in Google Pub/Sub [19]. A message is produced to a topic, and is subsequently persisted to storage. Subscriptions to the topic automatically receive the message and reply with an acknowledgement.

Pub/Sub offers flexible delivery options, that is, both pull- and push-based subscriptions are supported. The former allows subscribing application to process the messages at their own rate, while the latter is particularly suited for alerting mechanisms. To date, push subscribers must be Webhook endpoints that accept POST requests over HTTPS [19].
4.5.2 Architectural Overview

Pub/Sub is designed to be horizontally scalable. In this light, increasing numbers of topics, subscriptions or messages are handled by increasing the number of instances of running workers [9]. The service is composed of two distinct planes, the data plane on the one hand, and the control plane on the other. The data plane is responsible for the delivery of messages from the publishers to the subscribers, the nodes that make up the data plane are called forwards. The control plane assigns publishers and subscribers to servers on the data plane, nodes in the control plane are referred to as routers.

Google Pub/Sub is a global service, implying that it is cross data-center. Routers assign clients to data centers such that the network distance is minimized. Within a data center, consistent hashing [55] is leveraged to distribute the overall load across the set of available forwarders. The data plane receives messages from publishers and relays them to subscribers. When a publisher sends a message to a specific topic, it is sent to a publishing forwarder, that is, a forwarder to which the publisher is connected. This is illustrated by Figure 4.10.

In order to ensure at-least-once delivery, the message is immediately persisted to storage. In this light, the forwarder writes the message to multiple clusters and waits for a quorum [42] of clusters to acknowledge the message before it considers the message to be persisted. The publishing forwarder stores a list of subscriptions attached to a topic and is responsible for maintaining publish message sources. The collection of messages stored by a publishing forwarder along with the tracking of acknowledged messages, for a particular topic, is referred to as the publish message source. Similarly, subscribers may connect to multiple subscribing forwarders to receive messages. Pub/Sub dynamically tunes the number of publishing and subscribing forwarders for a particular topic as the throughput changes.
Subscribing forwarders thus make requests to one or more publishing forwarders that maintain a publish message source for the considered topic. The publishing forwarder in turn, sends the unacknowledged messages to the subscribing forwarder, which are thereafter delivered to the subscriber. Upon receiving the messages, the subscriber acknowledge message to the subscribing forwarder, which is subsequently relayed to the publishing forwarder. Once all subscriptions for a topic have acknowledges a message, the message is deleted from the publish message source. Additionally, users also specify a maximum duration that a message is allowed to be stuck in the publish message source. This avoids that messages are stored indefinitely, and defaults to seven days [19].

4.5.3 Kafka

Kafka is a distributed publish/subscribe messaging system developed for collecting and delivering high volumes of log data with low latency [46]. Figure 4.11 visualizes the architecture of a Kafka cluster, each of the concepts will be discussed in the following sections. This introduction to Kafka was adapted from reference [56].

4.5.4 Messages and Batches

The unit of data within Kafka is called a message. A message is defined to contain solely a payload of bytes augmented with some optional metadata. Analogous to Pub/Sub, the
data within the payload is not subject to any restrictions, allowing users to choose their own serialization method to encode a message. For efficiency, messages are collected in batches and a batch of messages is included in a publish request. This allows multiple messages to be sent in a single request, avoiding the need of a full network roundtrip for each individual message.

4.5.5 Topics, Partitions and Streams

Messages in Kafka are accommodated in topics. These topics are divided in partitions and each partition can be hosted on a different server. This allows horizontal scaling of a topic across multiple servers, yielding performance far beyond the capabilities of a single server. A stream is defined as a single topic of data, regardless of the number of partitions.

Each partition of a log corresponds to a logical append-only log \[56\]. At the physical layer, a log is implemented as a collection of segment files of roughly equal size. When a new message is published, it is appended to the last segment file by the broker. Segments are periodically flushed to disk, after the segment is flushed to disk the contained messages become available for the consumers.

As opposed to Pub/Sub, Kafka does not store a message identifier for messages. Rather, messages are identified by their offset in the partitions’ logical log. Each message in a specific partition thus has a unique offset. This in contrast to typical messaging systems where an explicit message id is maintained. This choice is justified by the fact that it eliminates the overhead of maintaining a random-access index structure that maps message identifiers to message locations. As a consequence, message identifiers are monotonically increasing but non-consecutive. Only messages within a specific partition of a topic are guaranteed to be time-ordered. As a consequence, cross-partition time-ordering is not guaranteed. This is due to the fact that there is no synchronization among the partitions.

4.5.6 Producers, Consumers and Consumer Groups

In Kafka, clients that publish messages in a topic are called producers, while the clients subscribed to a topic are called consumers. There is however, nothing that stops a client from having both roles. The consumer subscribes to a set of topics and always consumes the messages from a particular partition sequentially. It is the responsibility of the consumer to keep track of which messages it has consumed. This is achieved by maintaining the offset for the last consumed message of each partition. The consumer can acknowledge a particular offset, indicating it has received all the messages prior to that offset in the partition. This allows consumers to stop and restart while retaining
their position in the log. The consumer issues asynchronous pull requests to the broker, containing the offset of the message from where the consumption begins along with the number of bytes to fetch. Allowing consumers to process the topic at their own rate. This pull-model has the benefit that a consumer can rewind back to an old offset and re-consume data.

Consumers are organized in *consumer groups*, consisting out of a set of consumers collaborating to consume a topic. Within a group, each partition is consumed by a single member. This member is referred to as the *owner* of the partition. Organizing consumers in groups allows horizontal scaling to consume large topics. If a member of the group fails, the partitions will be rebalanced. Different consumer groups each independently consume the full set of subscribed messages without coordination.

### 4.5.7 Brokers

A single Kafka server is called a *broker*. The broker is the intermediate component that serves both the producers and the consumers. The responsibilities of the broker are as follows:

1. Responding to fetch requests for partitions
2. Responding with messages that have been flushed to disk
3. Appending incoming messages to the log
4. Flushing segments containing messages to disk

It is the responsibility of the consumer to keep track of which messages it has consumed. The consequence of this is that it is impossible to know for the broker when to delete a message. This problem is solved by retention: a message is deleted if it is older than a certain period\(^5\).

Brokers are designed to operate as part of a cluster. In a cluster of brokers, one broker will take the role as the *controller*. The controller assigns partitions to clusters and monitors the brokers for failures. A partition is always owned by a broker that is called the *leader*. To make the system fault-tolerant, a partition can be assigned to multiple brokers. This results in multiple replications of the partition in question.

While the systems studied in this chapter are used for the same purpose, their characteristics are fundamentally different. Usage of Kafka is rather complicated, as the cluster needs to be maintained and configured. Moreover, numerous configurations have to be

\(^5\)An alternative option is to delete messages when the size of the log exceeds a certain threshold.
fine-tuned, i.e., the number of partitions, the number of brokers and the number of consumer groups. Conversely, the cloud-based, fully managed character of Cloud Pub/Sub makes it particularly interesting to use. To use the service, the developer creates a topic and the associated subscriptions. The caveat of using Pub/Sub is that it guarantees at-least-once processing, implying that duplicates might be delivered to consumers. This is, however, not the case for Kafka. The fact that consumers maintain their offset in the log makes it possible to deduplicate such duplicates, thereby assuring exactly-once.
Chapter 5

Data Processing

Stream processing has gained a significant amount of attention in the field over the recent years. In today’s society massive unbounded and unordered internet-scale datasets are the rule rather than the exception. While at the same time, the consumers of these datasets have evolved sophisticated requirements, such as event-time ordering and windowing by features of the data themselves while simultaneously craving even-more timely results \[22\]. Stream processing allows organizations to make faster and better decisions, particularly due to their ability to provide low-latency results \[24\]. Furthermore, processing records as they arrive yields a better spread of workloads over time, thereby allowing resources to be utilized more consistently and predictable. This chapter introduces concepts and terminology required to describe the Beam Model (Chapter 6).

5.1 Batch Processing

Batch processing systems are employed in high-throughput use-cases, where the latency by which results are produced is not important. In batch processing, data is generally assumed to be bounded. Many operations in batch processing systems heavily rely on this assumption. A common example is the sorting operator in MapReduce \[34\], which has to read the entire data before it can produce any output. Batch systems essentially always introduce latency problems inherent with collecting all the input data \[61\].

An approach taken by batch processing systems is therefore to artificially divide the data into chunks of fixed size. Many MapReduce jobs are scheduled periodically, for example over night, which implies that changes are only reflected in the output produced the following day. This latency is not feasible for many applications, as they want to anticipate to events more quickly. An attempt to reduce this latency is to run these jobs more frequently or even continuously, thereby abandoning the fixed-time chunks.
and simply process the events in real time.

5.2 The Lambda Architecture

The Lambda architecture is a data-processing architecture designed to handle large datasets by combining both batch and stream-processing methods. This approach was mainly introduced due to the fact that early streaming systems provided limited or no guarantees about the output produced. The streaming system is fed the latest data to yield low-latency, speculative results. To produce the correct result, a batch job is periodically scheduled with the complete input collection.

Figure 5.1: Lambda architecture layout. The streaming system provides low-latency, speculative results. Periodically, a batch job is scheduled to correct the results produced by the streaming system.

Criticism of lambda architecture has focused on its inherent complexity and its limiting influence. The batch and streaming sides each require a different code base that must be maintained and kept in sync so that processed data produces the same result from both paths.

5.3 Stream Processing

It is becoming more and more apparent that a huge number of today’s large scale data processing applications handle data that is, in reality, produced continuously. As such, we must assume that such datasets are no longer limited, they are infinite in size. Stream processing refers to the ongoing processing of one or more event streams. It is a continuous and non-blocking approach, particularly fit for business that endeavor low-low
latency results. Like request-response and batch processing, stream processing is just another programming paradigm.

An event stream is an abstract representation of an unbounded dataset [43]. In the context of batch processing, the input of a job is presented in the form of a file consisting of a sequence of records. When considering streams, a record is commonly referred to as an event. An event is a self-contained unit that is usually augmented with a timestamp, reflecting the time at which the event itself occurred. The majority of data sources are born as continuous event streams, for example sensor events, user activity on a website, financial trades, and so on. There are a few attributes of event streams, in addition to their unbounded nature [50]:

1. Event streams are ordered, implying that events occur before or after other events. This notion becomes apparent when dealing with events representing changes to the state of a database.

2. Events are immutable, once occurred, they can never be modified. Events do never disappear. Instead, additional events are written to the stream to record the cancellation of a former event.

3. An optional, but nevertheless desirable property of events is replayability. For the majority of applications, it is critical to be able to replay a raw stream of events to correct errors, try new analysis methods.

5.4 Preliminaries

The concepts covered in the following sections are defined rather somewhat inconsistently in literature. For this reason, let us establish some more profound definitions. Most of these definitions were adapted from references [61, 23, 22].

Datasets come in many flavors, but the primarily distinguishing features for a dataset in this discussion are its cardinality and constitution. The former dictates the size of the dataset, with the decisive aspect of the cardinality being finite or infinite. As such, bounded data is a type of dataset that is finite in size, while unbounded data is infinite in size. In this work, the terms bounded/unbounded are preferred over the often used batch/streaming terms in literature, due to the latter being tied to the use of a specific type of execution engine. In reality, however, unbounded datasets have been processed via repeated runs of batch systems [65] and well-designed streaming systems are perfectly capable of processing bounded data [22] [29]. In this light, the distinction between streaming and batch is irrelevant for datasets, and the terms will be used exclusively when describing runtime execution engines.
The constitution of a dataset refers to its access metaphor, and consequently defines the ways to interact with the data. The primary constitutions of interest are tables and streams. Tables, as encountered in traditional database systems, capture a snapshot of the data at a specific point in time. Streams, on the other hand, provide an element-by-element view of the evolution of a dataset over time. A streaming system is a type of data processing engine designed with unbounded datasets in mind. It is noteworthy that this definition of a streaming system encapsulates both true streaming and microbatch approaches. Examples of true streaming systems include Apache Flink \[29\], Kafka Streams \[16\] and Google Dataflow \[14\] while Spark Streaming \[65\] is a canonical example of a microbatch system.

5.5 Duality of Streams and Tables

Streams and tables are intimately related, that is, a stream can be viewed as a table, and a table can be viewed as a stream \[16\]. This phenomenon is referred to as the Stream-table Duality \[42\]. More concretely, a stream can be seen as a change-log of the table, where each record in the stream captures a state change of the table. This change-log resembles close affinity to the append-only log as encountered in traditional database systems \[37\]. Conversely, a table can be seen as a snapshot of the latest value for each key in the stream, at a particular moment in time. Akidau et al. \[61\], refer to tables as data at rest, whereas streams are considered data in motion.

**Example 15.** The close relationship between tables and streams is best explained in the context of a concrete example. Consider the setting in which a table is used to store the latest interaction for each consumer. The first column of Figure 5.2 visualizes snapshots of the table at successive points in time. Whenever an interaction with consumer takes place, the state of the table is updated accordingly. The changes to the table can be represented as a change-log stream, as visualized by the second column. The last column used the stream to the end of reconstructing the original table.

Example 15 also introduces the idea of change data capture \[42\]. Consider the setting of maintaining a database on two physically separated locations. Change data capture can be conceptualized as the process of using a stream to keep the replications of a database up-to-date.

---

1 Microbatch systems essentially emulate true streaming systems by processing streams as a sequence of small batches of data.
5.6 Time Domains

Unbounded data processing requires a clear understanding of the involved domains of time. When processing data relating to events in time there are two inherent time domains to consider [22]:

1. **Event time**, the time at which the event itself actually occurred, i.e., a record of the system clock time for the system that generated the event at the time of occurrence. In the literature, the systems that generate events are called *producers* [42].

2. **Processing time**, which is the time at which an event is observed at any given point during processing within the pipeline, i.e., the current time according to the system clock.

Another time domain that is commonly described in literature [29, 16] is ingestion time. Conceptually, ingestion time is positioned in between event time and processing time. Upon arrival at the processing system, the event is assigned a stable timestamp. The main advantage of this domain is that subsequent window operations over the stream will refer to an identical timestamp, whereas in processing time each window operator may assign the record to a different window based on the local system clock and transport delay [13].

Figure 5.2: Visualizing the duality of streams and tables.
CHAPTER 5. DATA PROCESSING

3. **Ingestion time** which is the point in time when an event or data record is received by the stream processor, i.e., the system clock time of the source operator at the time of arrival.

The event-time may be irrelevant for certain applications, such applications are referred to as time-agnostic. However, it is paramount for use-cases such as anomaly detection and billing applications. Event times and ingestion times are essentially always static, implying that remain constant processing. Processing time changes constantly for each event as it flows through the pipeline and time elapses. Figure 5.3 provides a schematic overview of these different time domains.

**Figure 5.3:** Schematic overview of the time domains involved when processing unbounded data.

It is noteworthy that ingestion time may also be defined at the level of the messaging queue, i.e., the time at which the event is received by the messaging queue. However, in the context of streaming systems, the ingestion time of an event corresponds to the time at which the event is ingested in the stream processing engine. The ingestion time domain is mentioned solely for the sake of completeness and will be omitted in this work.

### 5.6.1 Time-domain Skew

In an ideal world, the event time and processing time would always be the same, that is, incoming events are processed instantaneously. In this ideal world, there exists a consistent mapping between the involved time domains, as illustrated by Figure 5.3. In this illustration, event time and processing time are represented arrows, and the incoming events are visualized by the squares purple squares.
Unfortunately, when attempting to process data in real-time\(^2\), the realities of the employed systems come into play and result in a highly variable distortion in the mapping between the event time and processing time \([22, 61]\). This distortion is referred to as \textit{time-domain skew}, where the qualifier time-domain is often omitted for brevity. In practice, this skew originates from a wide variety of sources. Martin Kleppman \([42]\) introduces four prevalent classes:

1. \textbf{Shared resource limitations}: such as network congestion, network partitions or shared CPU utilization in a nondedicated environment, \ldots
2. \textbf{Hardware causes}: hardware crashes, failure recovery procedures, \ldots
3. \textbf{Software causes}: distributed system logic, scheduling algorithms, time spent processing, \ldots
4. \textbf{Features of the data}: such as key distribution, variance in throughput, variance in disorder, \ldots

A real-world setting, which lacks a consistent mapping between event time and processing time as a result of skew, requires the time domains to be treated as independent entities. From now on, the axis representing the time domains are therefore displayed orthogonally. The horizontal axis represents event time, while the vertical axis represents processing time. Throughout this work, diagrams representing this highly dynamic relationship between the domains will be referred to as time-domain mappings. Plotting the event- and processing-time timestamps of events arriving in a real-world system yields a time-domain mapping similar to the one depicted on Figure 5.5\(^5\).

\(^2\)Note that, when processing historical data, this distortion is even more devastating.
The optimal scenario, where event- and processing-time are equal, is represented by the dashed line in Figure 5.5. The solid red line represents reality, thereby highlighting the highly dynamic nature of the time-domain skew. Initially, the system that generates this graph starts to skew away from real-time as the pipeline lags, gradually converging towards real-time before lagging again and coinciding towards the end.

**Example 16.** To the end of making Figure 5.5 more tractable, four specific event instances have been highlighted. Assume for simplicity of discussion that events $e_1, e_2$ respectively $e_3, e_4$ occur within identical event time timespan $t$. In an ideal world, the processing time counterparts of $t$, $t'$ and $t''$ respectively should also be the same. However, intervals $t'$ and $t''$ differ considerably due to the time-domain skew introduced by the distributed nature of the underlaying systems.

Discussing time-domain skew is only relevant when processing events according to their event-time. Though, processing in event time can essentially be conceptualized as the ideal scenario from Figure 5.5, namely where event-time and processing time are equal. This implies that the events can be shifted towards the dashed line representing the idealized world, as illustrated by Figure 5.6.
CHAPTER 5. DATA PROCESSING

Figure 5.6: Conceptualizing processing in the processing time domain as the idealized event-time processing scenario.

Refer back to the red line representing reality in Figure 5.5. The time-domain skew can be defined identically in the two different time domains [61]:

1. **Processing time**: the vertical distance between the ideal-scenario and reality is the lag as observed in the processing time domain. The distance captures the delay between the event-time for a given event and its respective processing time.

2. **Event time**: the horizontal distance indicates the amount of event-time skew in the pipeline. It qualifies how far behind real-time the pipeline currently is.

Skew is a commonly exhibited phenomenon by distributed data processing systems and implies that data cannot be solely analyzed within the context of when they are observed if their event times are of interest. To fully appreciate this statement, consider windowing, a concept that will be discussed thoroughly in Section 5.7. Windowing is an approach taken to slice unbounded datasets into finite chunks along temporal boundaries. Elements belonging to the same chunk are processed collectively, analogous to the reduce phase in MapReduce [34]. Defining these boundaries in processing-time may
lead to incorrect results, due to event-time data being potentially assigned to incorrect processing-time windows. This is a consequence of the inconsistent correlation between the involved time domains. Consider the following example.

**Example 17.** A global-scale e-retailer reports the total revenue acquired on a per daily basis. As such, the purchases dataset is divided in fixed windows of a 24-hour duration. Assume for the sake of simplicity that the actual revenue is constant over the observed days. Due to a restart of the stream processing system, purchase data is not processed until the following day. As the system becomes available again, the system processes the pile of uncompleted work. When taking solely the processing-time of events into consideration, the revenue gathered by these out-of-order arriving purchases is included in the revenue aggregate for the incorrect day. When visualizing this data in a graph, it might thus appear that there was a sudden anomalous dip followed by a spike of purchases whilst processing the uncompleted work, when in fact the real rate was steady. This is illustrated by Figure 5.7.

![Figure 5.7: Windowing by processing time, in the context of a network partition.](image)

Not only are the results produced in Example 17 incorrect, they are also nondeterministic. This implies that repeated runs of the pipeline, on identical input, may produce distinct results. For many applications it is, however, critical to be able to replay the event stream to correct errors, try new analysis methods or perform audits [46, 56].

While resorting to event time solves the aforementioned correctness issue, it imposes another fundamental complication. Defining window boundaries in this domain leads to the problem that is referred to as *input completeness*. More concretely, how to determine when all the data for a given event-time have been observed in the context of unbounded,
out-of-order arriving data with no predictable correlation between processing- and event-time.

Example 18. Refer back to Example 17 in which the total revenue is reported on a daily basis. Let us now shift our attention to event-time based windowing. Because the event-time for a specific event is always static, the systems is able to account for the out-of-order arriving data correctly. However, the completeness issue now becomes apparent, namely, how to declare the results for a particular window finished? Declaring the results for a window finished is a process that has to be handled with extreme caution. Prematurely declaring the results for a window finished has the effect that incorrect results are produced, while keeping windows opened for an extended period of time introduces latency.

This problem is studied comprehensively in references [24, 22] and will be discussed thoroughly in depth in Section 6.5. Unfortunately, this problem is intractable for many real-world data sources [61]. To overcome this problem, Akidau et al. [14] propose a fundamental shift in approach to data processing. Instead of attempting to groom unbounded data into finite batches of information that eventually become complete, we should live and breathe under the assumption that we will never know if or when we have seen all our data, only that new data will arrive and old data may be retracted or updated. As such, tools should be designed that allow us to live in this world of uncertainty imposed by these complex datasets. The systems that we build should be able to cope with these facts on its own, with notions of completeness being a convenient optimization for specific and appropriate use cases rather than a semantic necessity across all of them [61].

5.7 Windowing

Windowing, as briefly introduced in the preceding section, is the notion of taking a data source and segmenting it into finite chunks for processing. Elements belonging to the same window are processed collectively. Windowing can be applied to either bounded or unbounded data sources. In the context of unbounded data, windowing is a concept that is indispensable to operations that involve grouping. This is because grouping operations typically require a complete input collection to be able to provide accurate results. For example, computing the number of events in an infinite set of data does not produce an actionable result. Computing the number of events over the last hour, on the other hand, leads to a result that is semantically well defined. When dealing with bounded data windowing is an optional, but nevertheless, semantically useful concept [22].
5.7.1 Window Alignment

Windows may either be applied uniformly across all the data for the window in question, or applied solely for a specific subset of data for the given time window. Windows that are applied uniformly are referred to as *aligned*, while non-uniform windows are referred to as *unaligned*. For instance, unaligned windows may be employed to spread window completion load more evenly over time. This is achieved by phase-shifting the windows for different subsets of the data, e.g. per key, by some random value [61].

5.7.2 Windowing Strategies

There exists many different approaches to windowing a dataset, wherein the applicability of such an approach is highly dependent on the use-case. The following paragraphs introduce the most commonly encountered approaches together with a representative usage scenario. An overview of the windowing strategies is presented in Figure 5.8. Most of the definitions of this section were adapted from the work of Akidau et al. [22].

![Fixed, Sliding, Sessions](image)

**Figure 5.8:** Overview of windowing strategies [61].

**Fixed Windows**

The simplest and most intuitive form of windowing slices time into segments of a fixed-size, temporal length. *Fixed windows*, also referred to as *tumbling windows* by Martin Kleppman [42], are thus defined by a static window size. Frequently encountered fixed windows are hourly windows or daily windows. When this windowing technique is leveraged, every event belongs to exactly one window [42]. Fixed windows are usually aligned, unless spreading of the window completion load is required.
CHAPTER 5. DATA PROCESSING

Figure 5.9: *Concrete example of fixed windowing.*

**Example 19.** Consider fixed windowing with a five-minute duration, as illustrated by Figure 5.9. All the events observed with timestamps between 10:00:00 and 10:04:59 are grouped into one window, events between 10:05:00 and 10:09:59 into the next window, and so on.

**Sliding Windows**

Sliding windows are a generalization of fixed windows. These windows are defined by a fixed length and a fixed period, for instance, hourly windows starting every minute. Assigning the period a value smaller than the size yields overlapping windows. As a consequence, events may be assigned to multiple windows. Fixed windows are essentially the special case of sliding windows where the period equals the length. A period greater than the length yields some sort of a sampling window that considers certain subsets of the data over time. In this case, some data may not be assigned to a window and is therefore dropped. Analogous to fixed windows, sliding windows are typically aligned.

**Example 20.** Consider sliding windows with a 10-minutes window size and a five-minute period. In this scenario, all the events with timestamps between 10:00:00 and 10:09:59 are grouped into one window, events between 10:05:00 and 11:14:59 into the next window, and so on. The resulting windows are visualized in Figure 5.10.

Martin Kleppman [42] used the term hopping windows to define the windows as described in this section. Furthermore, he established a different definition for sliding windows in his work. Herein are sliding windows defined as a means to group all the events that occur within some fixed interval of each other. This definition will not be used throughout this work and is mentioned solely for completeness.

**Session Windows**

Session windowing is a sophisticated, dynamic and data-driven windowing approach. Sessions are a special type of window that capture a period of inactivity over a subset of
the data, specifically per key. Sessions are, as opposed to the other windowing strategies, not defined by a length but rather by a timeout gap. Events occurring within a span of time less than the timeout are grouped together as a session. Session windows are particularly useful in data analysis because they can provide a view of the activities for a specific user over a specific period of time during which they were engaged in some activity [61]. Sessions are the canonical example of unaligned windows, because sessions are never identical across different subsets of data in practice.

**Example 21.** Consider session windowing with a gap duration of 5-minutes, computed per consumer. Incoming events are now colored according to the consumer that issued them. The resulting windows are visualized by Figure 5.11. A window is started once an event for the consumer is received, and the keeps window accepting new events until the five minutes of inactivity have passed.

It is noteworthy that session windows, by definition, always introduce latency proportional to the period of inactivity. Sessions with a longer gap duration may be able to encapsulate more events, while at the same time, they translate to longer latency.
5.7.3 Event Time and Processing Time Windowing

The windowing strategies defined in the preceding sections can be applied to a dataset in either the event time or processing time paradigm. The applicability of windowing in either domain is strongly connected to the use case at hand.

Let us first consider the most commonly encountered approach, that is, windowing within the processing time paradigm. Conceptually, this windowing strategy is implemented by buffering incoming data until some predefined amount of processing time has elapsed. Processing time windowing has been particularly attractive due to its simplicity. As illustrated by Figure 5.12, the incoming data is windowed based solely on the order of arrival within the pipeline. In this light, data is never re-organized within time, thereby simplifying the process of reasoning about the windows sent to downstream components of the pipeline. During processing, judging whether all input for a specific window has been received becomes trivial due to the fact that the window boundaries are defined within the same time domain. In the context of processing-time windowing, there is thus no notion of late data, i.e., data arriving in a specific window after it was assumed to be complete. Processing time windowing is thus most suited for use-cases where information is derived from the source as it is observed, e.g., server monitoring scenarios.

![Figure 5.12: Processing time windowing](image)

While processing time windowing might seem attractive for a wide variety of use-cases at first, it has a major shortcoming. If the data in question has event times associated with them, those data must arrive in event-time order if the processing-time windows are to reflect the reality of when those events actually occurred [61]. However, due to the distributed nature of many real-world, unbounded input sources, event-time ordered datasets are rarely encountered in practice.

Event-time windowing is required for use-cases where the resulting windows need to reflect the times at which the events actually occurred. In contrast to processing time windowing, events are now re-organized in time to yield windows that reflect their event-time ordering. This is illustrated by Figure 5.13 where the black arrows indicate data that arrived in processing-time windows different from the event-time windows to which...
they originally belonged. If instead, processing-time windowing had been employed, it was exactly due to these events that the calculated results would have been incorrect. While event time windowing yields event-time correctness, it introduces the problem of input completeness. Due to the fact that window boundaries are now defined in event time, as opposed to the processing time paradigm wherein events are processed, judging whether all input for a specific window has been observed becomes non-trivial. Specifically, given an event time $t_E$, how judge whether all inputs with event-time $t'_E$, where $t'_E < t_E$ have been observed, lacking a consistent mapping between the involved time domains. Determining exactly when all the data for a given event-time have been observed is proved to be intractable \cite{22}. Fortunately, for many types of input sources, streaming systems can provide reasonable accurate heuristic estimates of window completion through a system called watermarks as encountered in MillWheel \cite{24}, Cloud Dataflow \cite{14} and Flink \cite{29}. Watermarks are a comprehensively studied topic in the context of streaming systems and their discussion is deferred until Section \ref{sec:watermarks}.

![Figure 5.13: Event time windowing \cite{67}](image)

**Example 22.** To appreciate the fundamental difference between windowing in the different time paradigms, Figure 5.14 presents a time-domain mapping of a real world system. The dataset was segmented according to the fixed windowing strategy as introduced in Section \ref{sec:windowing}. Figure 5.14a presents windowing in processing time, while Figure 5.14b visualizes windowing in event time. In an idealized world the events contained within windows 1,2,3 respectively A,B,C should match. This is clearly not the case, window 1 clearly misses event $e_2$ due to being delayed as a result of skew. Moreover, window 2 contains events $e_2,e_3$ and $e_4$ while window B contains events $e_3,e_4$ and $e_5$. Similarly, window 3 contains $e_5$ and $e_6$ while window A contains solely $e_6$.  

\[ 62 \]
(a) Processing time windowing.

(b) Event time windowing.

Figure 5.14: Windowing in processing time versus windowing in event time.
Chapter 6

Apache Beam

This chapter zooms in on the specifications of the Beam model. Apache Beam [1] is in essence a portability framework for big data pipelines. The project originated from the Dataflow Model presented by Akidau et al. [22] and was initially backed by Google, who donated it to the Apache Software Foundation afterwards. It is a very powerful programming model that unifies batch and stream processing. The notions of windows, triggers and watermarks are studied. We examen how these fundamental concepts allow for explicit trade-offs in the dimensions of latency, completeness and cost.

6.1 Introduction

Beam provides an abstraction from the distributed systems logic and consequently allows parallel data computations to be expressed as a sequence of logical transformations on input collections. Depending on the type of input collection, that is bounded or unbounded, the program is referred to as a batch respectively streaming computation. While batch computations are supported by the Apache Beam programming model, this work focusses particularly on streaming computations. That is, processing data originating from an unbounded source where events may arrive delayed and out-of-order. Beam comes with a very broad landscape of supporting execution engines, also referred to as runners. These runners execute Beam programs in a fault-tolerant manner. Portability refers to the fact that a beam pipeline has to be written once, and can be ran anywhere desired, thereby overcoming the oft-recurring vendor lock-in issue.

Figure 6.2 presents the extensive landscape of Beam. Beam programs are designed using a domain-specific language. After implementing the pipeline, the author decides which runner to use. The runner determines on which back-end the pipeline will execute. The framework comes with a multitude of runners including, but not limited to, Apache
Flink \[29\], Apache Spark \[65\] and Cloud Dataflow \[14\]. Moreover, the framework comes with a local runner that allows quick iteration and prototyping. Apache Beam is an open-source initiative that is under continuous development. While the runners are maintained by independent organizations, the functionality of the Beam model is not limited to the lowest common denominator thereof. Rather, the project attempts to incentivize the various runners to implement the latests standards into their execution engines. As such, functionality of the model may or may not be supported by the various execution engines. The functionality offered by the runners, with respect to the Beam model, is summarized in the Beam Capability Matrix (Figure 6.1).

### 6.2 Completeness, Latency and Cost

The Beam model is particularly attractive because it empowers the programmer to balance between the dimensions of completeness, latency and cost. Completeness refers to the fact that the computed result should take all the input data into account. The general idea is that the longer we wait for the data, the closer we get to a complete input collection. Latency hence refers to the time spent waiting for these results. In one extreme, results can either be produced immediately when new data comes in. This has a high risk of producing inaccurate results and may require reprocessing when new data arrives. In the other extreme, results are computed after all the input data was received. Waiting explicitly until all data has been observed naturally translates to increased time to answer. Cost refers to the total amount of work done, and hence resources used, to acquire to the final result. To the end of getting more intuition between the interplay of these dimensions, consider the following use-cases:

<table>
<thead>
<tr>
<th>Monthly billing</th>
<th>A company has a billing procedure that produces invoices for the usage of a service and forwards them to their clients. In this scenario, completeness is paramount because money is involved. Conversely, low latency and low cost are of minor importance given the use-case definition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monthly billing estimate</td>
<td>The service comes with a dashboard that provides the clients with a running estimate during the course of the billing period. Since the interest is in producing an estimate, the dimension of completeness becomes less important. Low latency now turns into the primary priority, while having a low cost is preferable.</td>
</tr>
<tr>
<td>Anomaly detection</td>
<td>Parallel to the use-cases outlined above, the company is also interested in identifying fraudulent users and deny their access to the service. This use-cases stresses low latency, while having a low cost or a complete input collection is certainly not important.</td>
</tr>
</tbody>
</table>
Figure 6.1: Beam capability-matrix [7].
Notice in particular how the monthly billing and the monthly billing estimate compute identical results. However, due to their goals they fundamentally differ in their balance between the dimensions of completeness, latency and cost. Figure 6.3 visualizes the balance for each of the use-cases outlined above in the form of a radar chart.

### 6.3 Pipeline Topology

A Beam pipeline encapsulates an end-to-end data processing task. This includes fetching data from input sources, applying transformations, and writing data to output sinks. Moreover, operational parameters, such as the execution engine to use, are also specified within the pipeline. A data processing pipeline can modeled by a directed, acyclic graph as presented in Figure 6.4. In this graph, edges represent data streams and nodes represent data transformations on these streams. In the Beam domain-specific language, pipelines are composed of the following building blocks:

1. **PCollection**: A PCollection represents a distributed dataset to which parallel transformations can be performed. This dataset can either be of bounded or unbounded nature. Bounded datasets originate from a fixed source, such as a text file, while unbounded datasets come from a continuously updating source, for example a messaging queue. PCollections are the input and outputs of each step in
Figure 6.3: Radar chart visualizing the interplay of completeness, latency and cost for different use-cases. Importance is measured from 0 to 3, where 0 is the least important and 3 is the most important.

the pipeline.

2. **PTransform**: A PTransform represents a massively parallel data transformation within the pipeline. PTransform are applied to input PCollections to yield new PCollections.

PCollections thus correspond to the edges in the graph, while PTransforms correspond to the nodes in the graph. The cylinders in Figure 6.4 correspond to data sources and sinks. Data is most commonly ingested in the pipeline through an external data source and the last step in the pipeline writes the output collection to a data sink. To the end of simplifying the communication with external services, Beam’s source APIs contain a multitude of adaptors to read from respectively write to various external data sources including cloud-based filesystems and databases.

### 6.3.1 PCollections

While a PCollection resembles close affinity to collection libraries as encountered in traditional programming languages. It is noteworthy that these are fundamentally different. First of all, runners have to be able to serialize and deserialize each element as a byte string to the end of supporting intermachine communication. As such, Beam provides built-in mechanisms for encoding and decoding commonly-used types in the form of a so-called coder. Conversely, for custom datatypes, Beam offers the ability to define customized coders as required \[3\]. Secondly, the elements contained within the

---

1. Alternatively, PCollections can be created from in-memory pipeline-graph data within the pipeline.
PCollection may be of any type, but all records are required to be of identical type. Moreover, PCollections are immutable, meaning that adding, removing or altering individual elements is prohibited \[2\]. To obtain a modified PCollection, a PTransform has to be invoked to produce a distinct output collection\[3\]. The input collection hence remains unaltered. PCollections, in contrast to most collection classes, do not provide random access. This implies that a transform in the pipeline processes each record in the collection individually.

6.3.2 PTransforms

PTransforms are applied to PCollections in a parallel fashion. As stated earlier, the desired processing logic is applied to the records in the input collection. In particular, a processing function is applied to each record in the input collection, resulting in zero, one or more output records. These output records are added to the respective output collection. Beam distinguishes between three primary types of transforms, namely element-wise, grouping and composite transforms, as illustrated by Figure 6.5. Composite transforms essentially encapsulate a sequence of element-wise and grouping transforms into a single, logical unit. Nesting multiple transforms inside a single composite transform stimulates reusability and simplifies the code.

\[2\] PTransforms may optionally produce multiple output PCollections through the TupleTag construct \[3\].
The Beam API comes with a plethora of built-in primitives which can be used out-of-the-box. Conversely, users have the ability to provide their own processing logic through the parallel-do transform. An exhaustive list of transforms can be found in the Beam Programming guide [3]. The core transformations, corresponding to different processing paradigms, are summarized below. While it is not strictly necessarily, this work assumes that input collections are composed of key/value pairs.

**ParDo**  ParDo is an element-wise transform for generic parallel programming [22]. This transform is equivalent with the Mapper function encountered in MapReduce programs. User-defined processing logic is passed to a parallel-do invocation and transparently applied to each element in the input collection in a parallel fashion. Example use-cases include filtering, formatting and extracting specific attributes from the input data.

**Example 23.** Consider the user-defined operation that expands all the prefixes for the input key. Specifically, given an input element, the operation outputs an element for each prefix and duplicates the associate value.

\[
\begin{align*}
[(\text{fix}, 1), ([\text{fit}, 2])] \\
\text{ParDo}(\text{ExpandPrefixes}) \\
[(f, 1), (\text{fi}, 1), (\text{fix}, 1), (f, 2), (\text{fi}, 2), (\text{fit}, 2)]
\end{align*}
\]

**GroupByKey**  GroupByKey is a grouping transform that is defined solely in the context of processing collections consisting of key/value pairs. It is a reduction operator, analogous to the Shuffle phase in MapReduce environments, that collects all the values
CHAPTER 6. APACHE BEAM

associated with each unique key. An example use-case, when dealing with interaction
data, is collecting all the interactions for each unique user.

Example 24. Applying the grouping transform on the output produced by the Expand-
Prefixes transform from Example 23 yields the following collection.

\[
\begin{align*}
\text{GroupByKey} & \quad \left[ \begin{array}{c}
(f, 1), (fi, 1), (fix, 1), (f, 2), (fi, 2), (fit, 2)
\end{array} \right] \\
\end{align*}
\]

\[
\begin{align*}
\text{GroupByKey} & \quad \left[ \begin{array}{c}
(f, [1, 2]), (fi, [1, 2]), (fix, [1]), (fit, [2])
\end{array} \right] \\
\end{align*}
\]

CoGroupByKey  CoGroupByKey implements a streaming join, as encountered in the
relational model. Analogous to GroupByKey, CoGroupByGrey operates on collections
containing key/value pairs and performs a relational join of these input collections. An
example use-case is enriching click events with continuously updating user information.

Combine  The combine transformation is leveraged to combine collections of elements,
specifically key/value pairs. Combine transforms apply an associative, commutative
accumulating operation across all the elements with a common key. The general idea is
that a combine function is applied on a per-key basis to build an accumulator. Every
time a record arrives at the transform, the \textit{addInput} method is invoked to alter the state
of the accumulator. When all data is processed, the \textit{extractOutput} method is called to
obtain the final output. The relationship of the accumulator, the \textit{addInput} method and
\textit{extractOutput} method is presented graphically by Figure 6.6. Incoming elements are
visualized by the squares, the accumulator is visualized by the triangle and the final
output by the circle. To be able to recover from failures, the accumulator is periodically
persisted to storage.

Example 25. One of the built-in primitives provided by Beam is the \textit{CountPerKey}
transform. The transform produces a collection of (key, count) pairs, where count re-
resents the number of elements associated with key. For this specific example, the
transform maintains an accumulator that records the number of elements seen for each
key. The \textit{addInput} method increments the count upon receiving a new element.

\[
\begin{align*}
\text{CountPerKey} & \quad \left[ \begin{array}{c}
(f, 1), (fi, 1), (fix, 1), (f, 2), (fi, 2), (fit, 2)
\end{array} \right] \\
\end{align*}
\]

\[
\begin{align*}
\text{CountPerKey} & \quad \left[ \begin{array}{c}
(f, 2), (fi, 2), (fix, 1), (fit, 1)
\end{array} \right] \\
\end{align*}
\]
Beam allows for an end-to-end data processing task to be described as a sequence of logical transformations on a data source to yield the required result. To get more acquainted with the concepts above and their syntax in the Beam DSL, consider the following running example.

**Example 26.** The pipeline is defined within the context of the billing procedure. The input dataset contains interactions of consumers with a digital service. The dataset is stored in the form of a text file, where each line is a JSON serialized representation of an interaction. The information recorded for an interaction includes, among others, the timestamp of the event and the corresponding consumer. Let us start by describing the set of logical transformations to yield the number of billable events per individual consumer. Figure 6.7 visualizes the transformations involved in this pipeline. It is noteworthy that, according to the Beam model, this pipeline performs a batch computation due to the fact that data from a bounded input source is being processed.

![Pipeline Diagram](image)

**Figure 6.7:** *Pipeline to count the number of billable events per individual consumer.*

Initially, the pipeline reads the lines from the input file and parses them into their object
representation. The parsed events are subsequently filtered such that only billable events remain. The Format stage ensures that the downstream transform performs on a key/-value pairs. Specifically, Format promotes the consumer attribute of the event to emit a \((\text{consumer}, \text{event})\) pair. Next up is the built-in primitive CountPerKey that counts the number of events per individual consumer. Prior to persisting the computed counts to storage, these are formatted according to the \text{consumer: count} pattern. Listing 6.1 presents a simplified version of the code used to implement this pipeline in the Beam DSL in Java. Notice in particular that there is a line of code for each of the transforms presented in Figure 6.7.

```java
Pipeline p = Pipeline.create();
p.apply(TextIO.Read.from("Interactions"))
  .apply(MapElements.via(line -> ParseJson(line)))
  .apply(Filter.by(isBillable))
  .apply(MapElements.via(event -> KV.of(event.consumer, event)))
  .apply(Count.PerKey())
  .apply(MapElements.via(count -> count.key() + ": " + count.value()))
  .apply(TextIO.Write.to("BillablePerConsumer"))
p.run();
```

Listing 6.1: Collecting the number of interactions per consumer.

Batch computations do, by definition, not produce any output until they have processed all the data in the input collection. How pipelines are executed is discussed thoroughly in Chapter 7. Let us for the moment gain at-least some high-level insights on how the pipeline from Example 26 operates. Figure 6.8 presents a snapshot of the pipeline’s state at three successive processing-time timestamps. Elements in the collection, obtained after the filter step, are visualized by the squares. The elements are organized according to their event-time and processing-time timestamps. For simplicity, we consider only the events originating from a single consumer, as indicated by the blue color. As the pipeline observes new values, they are incorporated in its intermediate state and eventually materialized as output. For the use-case defined in Example 26, solely the number of events for the consumer is accumulated. The green area represents the the portion of event time and processing time accumulated in the result, with the aggregate value displayed near the top. Conclude that as processing-time elapses, as indicated by the green horizontal line, more and more elements are processed. After the entire input collection is processed, the result is complete and can be emitted.
CHAPTER 6. APACHE BEAM

Figure 6.8: Operation of the billable count pipeline on a fixed input collection.

While the batch Pipeline introduced in Example 26 successfully covers the use-case, it requires the input collection to be complete. In reality, however, most datasets are of unbounded nature. Common examples of unbounded datasets include social network interaction data and sensor data. When operating on unbounded datasets, it makes much more sense to feed the pipeline through an unbounded source. Prevalent examples of unbounded sources are Google Cloud Pub/Sub [9] and Kafka [46]. Details of these systems are discussed in-depth in Chapter 4.5, but for our purposes it suffices to conceptualize these as messaging queues into which a potential infinite number of events can be ingested.

Unbounded input collections render classic batch processing useless, due to the fact that the input collection is never effectively known to be complete. The shift from a bounded to an unbounded input source entails certain complications due to the fact that most operations do not produce a well-defined result over an infinite dataset. The concept of windowing, as described in Section 5.7, is therefore employed to slice the dataset into finite chunks for processing. The Beam model supports the various techniques presented in Section 5.7.2. How these windows manifest under the hood will be examined in Section 6.4. For the moment being, think of windows as simply being a collection of buckets. When new data is ingested in the pipeline, it is added to a bucket according to its event-time. Consider the refined version of the billing pipeline in Example 27.

Example 27. Instead of adding interactions to a text file, an event is produced to Google Cloud Pub/Sub every time a consumer invokes the digital service. The approach taken in Example 26 is thus no longer feasible, particularly due to the fact that counting the number of events over an infinite dataset does not yield an actionable result. Redefine the pipeline to count the number of billable events, per user, per hour. Fixed windowing in event time is used to produce results that respect the event time ordering of the events.
6.4 Windowing Model

The concept of windowing can be conceptualized by buckets that buffer incoming elements. When new data is received, it is added to a bucket according to its event time. It is important to note that, under the hood, windows are never explicitly constructed. Windows instead manifest implicitly at grouping-time. In this light, grouping is internally redefined to become a GroupByKeyAndWindow operation. Apache Beam provides a framework that seamlessly accommodates the windowing strategies described in Section 5.7. Instead of operating on plain key/value pairs, the following quadruples are propagated through the system [22]:

\[(\text{key, value, timestamp, window})\]

Where timestamp corresponds to the event time of the event, and window represents the window to which the event belongs. Windows are denoted \([w_{\text{start}}, w_{\text{end}}]\), where \(w_{\text{start}}\) and \(w_{\text{end}}\) are timestamps representing the window boundaries. The key and value attributes are accessible to the user directly, whereas the timestamp and window are changed by applying specific transforms on the element. All incoming elements are initially assigned to a default global window that spans the entire event-time domain. The default global window is denoted \([0, \infty)\). A key insight is that both the key and window are considered first-class citizens of the element [61]. There are essentially two intimately related primitives that underpin the windowing framework in Apache Beam [22]:

1. **Window assignment**, an operation that assigns the element to zero, one or more windows. Conceptually, this function returns the set of windows for a given input element.

2. **Window merging**, which collocates windows at grouping time. This allows windows to evolve over time as new data arrives.

Defining a windowing strategy thus requires both assignment and merging logic to be defined. In many cases it is perfectly valid for an element to be associated with multiple windows. The fact that windows are stored as part of the element itself has an important implication. Whenever an element is assigned to multiple windows, the assignment operation creates a duplicate of the element for each of the windows to which it was assigned [22].

**Example 28.** Consider windowing a dataset according to the sliding windows strategy. Specifically, define a two minute duration and a one minute period. Note that, by definition, there is overlap between subsequent windows. This implies that an element may potentially be assigned to multiple windows.
Example 29. To gain further insights on the interplay between the windowing primitives, let us consider the end-to-end process of applying session windowing. Sessions may be employed to capture bursts of activity that are followed by a gap of inactivity. Assume that activity, and hence a session, ends when no events have been observed for a duration of 30 minutes. The input data is visualized in Figure 6.9. Keys $k_1$ and $k_2$ represent the users that triggered events $v_1$, $v_3$, $v_4$ and $v_2$ respectively.

The first step in the process is applying the window assignment operation. For session windowing in particular, the primitive accommodates each record in its own window, these windows are called *proto-sessions* in [61]. The proto-sessions each have a duration of 30 minutes and start at the event time of the associated record. The window assignment primitive, applied to our input collection, is visualized in Figure 6.10.
Thereafter, the merging operation is invoked. For session windowing, this operator merges overlapping windows, with identical keys, together. The windows, for key $k_1$, containing $v_1$ and $v_3$ respectively overlap and are therefore merged to yield a larger session window containing the two records. The other windows remain unaltered. The situation is visualized in Figure 6.11.

The figures above serve merely for the sake of intuition. As mentioned earlier, windows are never explicitly constructed. Rather, window construction occurs implicitly at grouping time. Figure 6.12 presents session windowing by means of the core primitives.
The input records are accommodated in their own proto-session. The proto-session starts at the event time timestamp of the corresponding record and has a length of 30 minutes.

The timestamps of the records are dropped, as they are no longer relevant. If subsequent stages of the pipeline require the timestamp, it can be materialized as part of the record value.

The records are grouped based on their key to yield a collection of \((value, window)\) pairs for each key.

The corresponding windows, for a particular key, are merged according to the merge logic defined by the windowing strategy. For session windowing in particular, overlapping windows are merged in a single new, larger session. In the input data, only the proto-sessions for \(v_1\) and \(v_4\) overlap.

For each key, the resulting \((value, window)\) pairs are merged according to their
window to yield a collection of values per window, per key. Values \( v_1 \) and \( v_4 \) have the same window, due to the merge in the prior step, and are therefore grouped.

6 The resulting groups are now expanded to \((key, value, timestamp, window)\) tuples with new per-window timestamps. In this example, \( timestamp \) is set to the end of the window. This is certainly not the only option, as discussed in Section 6.8.1.

6.5 Watermarks

Section 5.7.3 highlighted the differences between windowing in event time or processing time. Windowing in event time poses another fundamental complication during processing. Consider the colored portions, representing event time windows, in the graph presented in Figure 6.13. Recall that events are never processed instantaneously and essentially always arrive with a highly variable amount of delay due to time-domain skew (Section 5.6.1). As a result, events can arrive just anywhere in processing time.

![Figure 6.13: Windowing the events in the interactions dataset in event time.](image)

The colored bands of Figure 6.13 thus essentially extend forever upward and computation is expressed in terms of the entire contents of these infinite, vertical bands. This implies that, during processing, the pipeline stores all the data for all windows and waits forever. This is indeed not feasible and a naive solution consists in defining a cutoff. Specifically, define a maximum allowed delay to which events are accepted and drop the events that arrive after this delay. Figure 6.14 presents a visual representation of this cutoff. Events colored grey are discarded as they exceed the maximum delay allowed.
Figure 6.14: Visualizing the notion of a cutoff, elements arriving after a maximum allowed delay are discarded.

Defining the cutoff has two major advantages. First off all, the top of each band is truncated and thus no longer extends forever upwards. Secondly, the cutoff limits the number of bands that have to be dealt with simultaneously. This greatly reduces the amount of state that the pipeline needs to maintain in its intermediate state. The disadvantage of the cutoff is that it is insensitive to the data, thereby resulting in a potentially large amount of discarded data. In essence, the cutoff gives a first approximation for a so called watermark.

Watermarks are temporal notions of input completeness in the event-time domain [22]. The general idea is that as processing time elapses, the estimate of progression in event time also proceeds. For real-world systems, this is an ever-changing function of time due to the distributed nature of the execution [61]. Figure 6.15 represents a time domain mapping for a real-world system. The green line represents advancement in the processing domain, whereas the red line visualizes the watermark. The point where the current processing time and the watermark intersect, represents completeness. For instance, assume that intersection occurs at 11:30 in processing time and 10:20 in event time. This indicates that, at 11:30 during processing, all the events with event time smaller than 10:20 have been observed.

A watermark is considered to be perfect when it manages to enclose all the data in the dataset [22]. Conversely, heuristic watermarks do not enclose all the data within the dataset. Figure 6.16a presents an example of a perfect watermark, while Figure 6.16b visualizes an heuristic watermark. How these watermarks are effectively computed is discussed in Section 6.8. For now we assume that it is determined transparently by the pipeline.
Figure 6.15: Progress in the event time domain as processing time elapses. The green line represents processing time advancement, whereas the red line visualizes the watermark.

Data that arrives after the watermark is referred to as late data [61]. By definition, the notion of late data is only relevant in the context of heuristic watermarks. The grey squares in Figure 6.16b call out late data.
CHAPTER 6. APACHE BEAM

Watermarks provide a measure of completeness with respect to the input collection. This is especially relevant for use-cases that prioritize completeness, for example, the billing pipeline. This is not a complete picture, as watermarks do not allow use-cases that stress low-latency to be addressed.

6.6 Triggers and Panes

Triggers determine where in processing time results are materialized[22]. Specifically, they are the primitive used to control the balance between the dimensions of latency and cost. It is noteworthy that triggers are always defined within the context of a specific window. Akidau et al. [61] introduce three classes of triggers.

1. Completeness triggers, triggers that materialize a result only when the contents of the window are believed to be complete. Watermarks, as introduced in Section 6.5, are the underlaying primitives that drive this class of triggers.

2. Repeated update triggers, triggers that produce results as the contents of the window evolve over time. This class is further divided in two distinct classes [3].

   (a) Data driven update triggers, triggers that examine the elements that arrive in the window and fire when the data meets a certain condition. For example, a trigger that generates an update after 5 elements have been observed.
(b) **Processing time update triggers**, trigger that fires after a certain amount of processing time has elapsed. For example, a trigger that fires after 10 seconds of processing time.

3. **Composite triggers**, the triggers described above may be combined to form more complex triggering mechanisms. For example, emitting a speculative result for every record that arrives, and additionally materializing a final result when the contents of the window are believed to be complete.

**Example 30.** With the notions of watermarks and triggers established, let us implement the refined version of the pipeline as described in Example 27. The simplified Java code of this pipeline, using the Beam DSL, is presented in Listing 6.2. Since the monthly billing pipeline requires complete results, a completeness trigger is used to materialize results. Assume for the moment that the pipeline is given near-perfect watermark information, late data is thus possible.

```java
apply( Window.into(FixedWindows.of(ONE_HOUR)))
  .triggering(AfterWatermark.PastEndOfWindow())
  .apply(Count.PerKey())
```

**Listing 6.2:** Collecting the number of interactions, per hour, per consumer.

To gain understanding on how the pipeline operates, consider the snapshots visualized in Figure 6.17. The green line represents progress in the processing time domain while the red line visualizes the watermark. To keep the figure concise, the accumulated state was omitted. Draw specific attention to the points where the watermark intersects the end of the window boundary. At these locations, the contents of the window are believed to be complete, the trigger is therefore activated and the aggregate is emitted. The emitted results are displayed in the triangles. By default, windows are closed when the completeness trigger fires. This has the effect that late data is discarded, as indicated by the grey squares in the figure.
CHAPTER 6. APACHE BEAM

Figure 6.17: Visualizing the mechanics of a completeness trigger. Panes are materialized when the watermark surpasses the boundary of the window.

This pipeline provides a high completeness and high latency solution, i.e., we wait until the input is believed to be complete before producing a result. Moreover, this approach is low cost due to the fact that there is a sole aggregate emitted for each window. To satisfy the low-latency requirement of the monthly billing estimate, we resort to a repeated update trigger. The repeated update trigger materializes a new result each time a record arrives. The code of this version presented in Listing 6.3.

```java
1... 
2   .apply(Window.into(FixedWindows.of(ONE_HOUR)))
3       .triggering(Repeatedly.forever
4         (elementCountAtLeast(1))
5   .apply(Count.PerKey())
6...```

Listing 6.3: Producing a new result after a new record arrives.

Figure 6.18 visualizes the conceptual pipeline operation on a slightly altered input col-
lection. For every newly arriving event, a new result is emitted. Notice in particular that no ordering is implied, due to the early firing mechanism. For instance, window [11:00, 12:00) now fires a result before window [10:00, 11:00). The multiple results produced by a window, as indicated by the boxes in Figure 6.18, are referred to as panes. Conclude that in this solution, cost was essentially traded for latency and completeness. The cost of this solution is considered high due to the fact that it is driven by the input.

![Figure 6.18: Visualizing the mechanics of a repeated update trigger. Panes are emitted instantaneously as a new record arrives.](image)

The drawback of the pipeline presented in Listing 6.3 is that its cost is proportional to the size of the input collection. Moreover, the fact that late data is discarded in the pipeline of Listing 6.2 is not feasible for many real-world applications. Listing 6.4 demonstrates how composite triggers can fine-tune the balance between completeness, latency and cost.

```java
1 ... 
2 .apply(Window.into(FixedWindows.of(ONE_HOUR)))
3 .triggering(
4     AfterWatermark.PastEndOfWindow()
```
The composite trigger presented above essentially combines three distinct triggering types. The mechanism materializes a complete result after the watermark passes the end of the window boundary. To the end of reducing latency, early triggers emit speculative results. Specifically, upon receiving new data, the trigger buffers incoming elements for a duration of two minutes before emitting a result. This has the effect that the trigger is no longer driven by the input, but rather by time. Finally, when late data is observed, the pipeline produces an immediate correction of the result. Figure 6.19 visualizes the trigger in action. Panes are colored according to the type of the trigger that fired it. Yellow triangles represent firings due to an early trigger, green triangles represent firings due to the completeness trigger and red triangles represent firings due to late trigger. Akidau et al. [61] refer to these panes as respectively early, on-time and late. Subsequent panes emitted by a specific window are subject to certain restrictions. First, a variable number of early panes is emitted, followed by at most one on time pane, followed by a variable number of late panes [40].

6.6.1 Global Windowing

The notion of windowing was introduced to the end of slicing an unbounded collection into finite chunks for processing. To produce results when processing unbounded collections, windowing has so far been a requisite. The notion of triggers essentially relaxes this requirement, as they allow for individual panes to be fired over time. A GlobalWindow in Beam can be conceptualized as a window spanning the entire event-time domain. Due to the unbounded nature of the dataset, the window is never complete. However, using a repeated update trigger, results can be produced as the contents of the windows evolve over time.

6.6.2 Allowed Lateness

To handle late data consistently, the pipeline presented in Listing 6.4 essentially accumulates state for windows until the end of time. Especially in the context of unbounded input sources, persisting state for all the windows indefinitely is impractical due to re-
Figure 6.19: Visualizing the mechanics of a composite trigger. The pipeline produces a speculative result after a certain amount of processing time, and a complete result when the watermark passes the end of the window. These firings are encoded by the yellow and green background color respectively. Late data is handled by producing refinements of the original result, as visualized by the red triangles.
source limitations. Beam provides a way to bound the lifespan of the windows, by means of an allowed lateness parameter $\text{[3]}$. The idea is similar to the cut-off introduced at the start of Section $\text{6.5}$. The main difference is that the cutoff is defined in processing time, whereas allowed lateness is defined in the event time domain. Define the *lateness horizon* for a window as the event time timestamp representing the end of the window, plus the allowed latency. Once the watermark reaches the lateness horizon for a particular window, the resources used to store the state of the window can be reclaimed. The region above the lateness horizon is referred to as the droppable zone, data arriving in this zone is simply discarded.

**Example 31.** Figure $\text{6.20}$ visualizes the lateness horizon and the droppable zone for a situation where the allowed latency set to one hour. For window $[10:00, 11:00]$, the lateness horizon is established at 12:00. Once the watermark reaches the lateness horizon, as visualized by the dotted line, the window is closed. Elements arriving in the droppable zone are discarded, as illustrated by the grey square.
6.7 Accumulation

Depending on the triggering mechanism used, multiple results for a single window may be emitted over time. Accumulation provides a system to control how multiple panes for the same window relate to each other. The Dataflow Model \cite{dataflow_model} introduces three different refinement modes:

1. **Discarding**, whenever a pane is emitted, any stored state is essentially discarded. This implies that subsequent panes bear no relation to one another. In essence, the trigger thus emits deltas over time. This is particularly useful in cases where downstream transforms expect the values from different panes to be independent, i.e., it is accumulating results.

2. **Accumulating**, upon materializing a pane, the state of the window remains intact. Successive panes thus re-emit a total, revised answer. This is particularly suited for use-cases where old results are overwritten when a new result is received.

3. **Accumulating and Retracting**, in addition to retaining the state for the window upon firing a pane, a snapshot of the state is persisted. Prior to materializing new output, the system produces a retraction for the previous pane. This mode is useful in the context of grouping operations, where it is generally not possible to overwrite the previous result. The retraction may be used to the end of reversing the original result.

It is noteworthy that in all of the examples previously presented, the accumulation mode was used. To the end of visualizing the difference between the various techniques, consider the following example.

**Example 32.** Figure \ref{fig:accumulation_example} presents the emitted panes for each refinement mode on an input collection. The retractions that are part of the Accumulating and Retracting mode are visualized by the purple triangles. Recall that, when using discarding mode, the final pane does not present a total sum. Rather, the sum can be obtain by summing the values of the independent panes. This particularly highlights the fact that discarding mode is useful when the downstream transforms are accumulating results. Conversely, for the accumulating mode, the final value does capture a total sum. However, when summing over the individual panes produced by window $[10:00, 11:00)$ over time, this yields the incorrect value of 6. This is exactly the issue that the accumulating and retracting mode addresses. Specifically, both the last value emitted and the sum of all materialized panes yield the correct answer of 3. Table \ref{table:accumulation_modes} summarizes these findings for the window representing event time interval $[10:00, 11:00)$.

\footnote{At the time of writing, solely the discarding and accumulation modes are supported by Beam according to the Beam Programming Guide \cite{beam_programming_guide}.}

89
Figure 6.21: Visualizing the different refinement modes.

Table 6.1: Relationship of the different accumulation modes.
6.8 Tracking Pipeline Progress Through Watermarks

Section 6.5 introduced the concept of watermarks at high level. Understanding how watermarks operate under the hood is key to understanding completeness and latency of the results produced by the pipeline. An end-to-end data processing pipeline can be conceptualized by a block box. To characterize the progress of the pipeline relative to its unbounded input collection, streaming systems observe the distribution of event times of the events entering the system. Events received by the pipeline, but not yet completed, are referred to as in-flight. After processing for a particular event ends, it is referred to as completed. Figure 6.22 presents the distribution of in-flight and completed events for a real-world system. Conclude that, as time elapses, more of the event time domain is completed and the graph thus moves gradually to the right.

![Figure 6.22: Visualizing a pipeline as a black box.](image)

Now, define a watermark to be a monotonically increasing timestamp of the oldest, unprocessed event in the pipeline. Akidau et al. [24] refer to this definition of watermarks as low watermarks. While this work focuses exclusively on low watermarks, it is noteworthy that there exist other definitions in literature. For instance, the percentile watermarks described in Streaming Systems [61]. This definition of watermarks exhibits two important characteristics, referred to as visibility and completeness. Visibility refers to the idea that if an event is stuck in the pipeline, the watermark will not be able to advance. Completeness, on the other hand, captures the intuition that when the watermark advances past timestamp \( t \), all the on-time events before \( t \) have been observed.

This characteristic drives the completeness trigger as described in Section 6.6.

**Example 33.** Figure 6.23 presents a snapshot of the events in a system at successive processing time timestamps. Advancement of the watermark is particularly highlighted.

---

4If the watermark was able to advance, it would eventually render the element droppable which is highly undesirable.
Blocks placed above the event time axis represent in-flight events, whereas blocks positioned under the axis represent completed events. Figure 6.23c calls out the importance of defining the watermark monotonically. The hatched event arrives late and if not defined monotonically, the watermark would jump back in time upon its arrival. Recall that, when surpassing a window boundary, watermarks are used to trigger the process of materializing panes. If the watermark was able to move back in time, this would potentially imply that the emitted pane has to be withdrawn. For this reason, elements arriving behind the watermark are considered late and windows may optionally produce late panes to refine earlier results.

In reality, pipelines are composed of a series of isolated transforms as illustrated by Figure 6.24. The pipeline is formalized by the directed, acyclic graph $G = (V, E)$, with $V$ a finite set of vertices and $E \subseteq \{ (x, y) \mid (x, y) \in V^2 \land x \neq y \}$ a set of edges over these vertices. The vertices in the graph represent collections, whereas the nodes represent transforms on these collections.

When the watermark is not advancing, the process of tracking down the exact problem source is complicated given its global definition. Instead, watermarks are defined at the boundaries of each step. More precisely, for each transform $t \in V$ there exists an input watermark and an output watermark, denoted $W_{in}(t)$ and $W_{out}(t)$ respectively. Consider the following recursive definitions, where $\mu_{\min}(t)$ is used to denote the mono-
tonically increasing timestamp of the oldest, unprocessed element at transform \( t \).

\[
W_{\text{out}}(t) = \min(W_{\text{in}}(t), \mu_{\text{min}}(t)) \tag{6.1}
\]

\[
W_{\text{in}}(t) = \min\{W_{\text{out}}(t') \mid (t', t) \in E\} \tag{6.2}
\]

The input watermark for a transform \( t \) can be conceptualized as the timestamp of the oldest work not yet sent to \( t \). The output watermark, on the other hand, is the oldest work not yet completed by \( t \). The main takeaway is that each transform has its own, independent view on reality. To get acquainted with these recursive definitions, consider the following example.

**Example 34.** Figure 6.24 presents a schematic overview of the input and output watermarks for each transform within the pipeline. Explicit indications for the input watermarks have been omitted for brevity of the figure. The output watermark of transform \( D \) equals the minimum of the input watermark and the timestamp of the oldest, unprocessed element within \( D \). More precisely, \( W_{\text{out}}(D) = \min(W_{\text{in}}(D), \mu_{\text{min}}(D)) \). The input watermark of transform \( D \) is the minimum of the upstream output watermarks, denoted \( W_{\text{in}}(D) = \min(W_{\text{out}}(B), W_{\text{out}}(C)) \). Similar reasoning can be used to yield \( W_{\text{out}}(C) = \min(W_{\text{in}}(C), \mu_{\text{min}}(C)) \) respectively \( W_{\text{in}}(C) = W_{\text{out}}(A) \) for the output and input watermark of transform \( C \), and so on. Seeing how the watermarks are propagated through the system, raises the natural question of how \( W_{\text{in}}(A) \) is obtained.

\[\diamond\]

In essence, this recursive definition has pushed the problem of determining watermarks to the data source. This makes perfect sense, as knowledge specific to the source can now
be exploited to provide the pipeline with watermark information. Section 6.5 introduced two classes of watermarks, perfect and heuristic. Perfect watermarks come, by definition, with the strict guarantee that no data with event times less then the watermark will ever arrive again. This class of watermarks requires perfect knowledge of the source, which is not feasible for many real-world distributed input sources. To fully appreciate why, consider the following example.

**Example 35.** A system processes a stream of click-events originating from mobile devices. However, devices can lose signal due to a plethora of reasons. One such reason is that the user might be on a plane. This implies that the click-events are accumulated and send to the system when the flight arrives. To have a watermark account for this events, it has to know that the user was taking a plane in the first place.

Heuristic watermarks provide no such guarantee, it is thus possible that data arrives behind the watermark. The term late data was reserved to refer to data that arrives behind the watermark. For the time being, assume that the watermark propagation mechanism does not introduce additional late data. In particular, this means that it is not possible for an element to be on time in one component of the pipeline, but late in another downstream component. If the source gives perfect watermark information, this information will thus remain perfect throughout the system. Conversely, if the watermark gives imperfect information, the information will remain imperfect but the pipeline will not introduce additional late data. Consider the specific example of watermark propagation below.

**Example 36.** Refer back to the billing pipeline described in Example 27. The events are subdivided in windows with a one-hour duration and a completeness trigger is used to materialize the results. Assume, for the sake of simplicity, that we are able to construct a perfect watermark for the input source. Figure 6.25 visualizes watermark initialization for the most prevalent steps in the pipeline. In particular, two special initialization values were used. The value *unknown* indicates lack of any sort of knowledge and therefore corresponds to the value of $-\infty$. The value *empty* is used to explicitly state that a transform is quiescent, the watermark should not be delayed, and hence maps to $+\infty$.

![Figure 6.25: Initializing watermark information for the billing pipeline.](image)

After watermark initialization, the input source reports that it wont sent messages with

---

94

---

5In fact, the other way around is permitted, i.e., turning late data into on-time data during processing.
event times before 1:00:00. This information is propagated to subsequent transforms using the recursive definitions presented in Equations (6.1) and (6.2). Note, in particular, that the choice of initialization values is essential to the proper conduct of the propagation procedure. Figure 6.26 presents a complete view of the watermark information after this step.

Figure 6.26: Propagation the watermark received by the input source.

Next, a new event \( e = (\text{key}, \text{value}, \text{timestamp}, \text{window}) \) is ingested in the pipeline. Based on the information provided by the perfect watermark, we can confidently say that \( \text{timestamp} \) is after 1:00:00. Figure 6.27 visualizes how the various transforms in the pipeline interact with the event. A dark background color is used to indicate that processing for \( e \) completed, whereas a light background color represents that \( e \) is being processed. The event is initially added to the global window that spans the entire event time domain. Prior to forwarding \( e \) downstream, Transform \( B \) inspects the value of \( \text{timestamp} \) and changes \( \text{window} \) accordingly. The event is subsequently buffered inside the Combine transform. More precisely, it is added to the accumulator that corresponds to the resulting \( \text{key} \) and \( \text{window} \) for \( e \). Buffering the event in Transform \( C \) has the effect that the value for \( \mu_{\min}(C) \) is defined, namely \( \mu_{\min}(C) = \text{timestamp} \).

Figure 6.27: Pushing an element through the pipeline.

As events keep arriving gradually over time, the source updates the input watermark accordingly. For simplicity of discussion, the events are all assumed to be associated
with the same key and window. A snapshot of the pipeline is presented in Figure 6.28. Recall that a completeness trigger is used to materialize results. Usage of this trigger implies that events remain buffered in the Combine stage until the watermark moves beyond the window boundary. Notice in particular that \( w_{out}(C) \) is now substantially delayed, due to the value of \( \mu_{min}(C) \).

The source will eventually increase the input watermark to 2:00:00, which in turn will be propagated towards Transform \( C \). The watermark has now surpassed the window boundary and the completeness trigger for the window [1:00, 2:00) fires. This has the effect that a pane containing the key with its respective count is materialized to the output collection. The perfect watermark guarantees that no more data with timestamps before 2:00:00 will be ingested in the pipeline. The window is complete and its resources can hence be reclaimed. This process is illustrated by Figure 6.29, the aggregate element is visualized by the circle.

The source updates the input watermark accordingly.
The key takeaway is that any delay in a transform will cause delay in the watermarks of downstream transforms. Thanks to the perfect watermark, the pipeline presented in the example did not have to address late data. Unfortunately the world of distributed systems is not so rosy, in fact, perfect watermarks are rarely encountered. Detecting late data, and providing specific semantics to deal with it, is thus of uttermost importance. To what extent late data is accepted is controlled by allowed lateness parameter, as described in Section 6.6.2. In reality, the resources used by the window are hence not reclaimed when the watermark passes the windows’ boundary, but instead when it passes the lateness horizon. Late data arriving before the lateness horizon is handled consistently, and data arriving after it is explicitly dropped.

Source watermark creation is highly specific to the input source, and is therefore not be considered in this work. Akidau et al. present an excellent case study that describes source watermark creation for Google Cloud Pub/Sub in Streaming Systems [61]. The characteristics of the service make it impossible to create a perfect watermark. The authors therefore introduce heuristic watermark for the source. The general idea is that the source maintains two distinct subscriptions to the source topic, i.e., a base subscription and a tracking subscription. The base subscription is used to read the data to be processed, whereas the tracking subscription is used to perform the watermark estimation. The tracking subscription effectively inspects the backlog of the base subscription and uses the minimum of the event timestamps in the backlog to construct the watermark estimate. Due to the fact that the tracking subscription does not perform any heavy lifting, it can be assumed to be sufficiently far ahead of the base subscription thereby producing a reasonable watermark.

### 6.8.1 Output Timestamps

Having described how individual elements are combined when presented to a grouping transform, the question on how to combine the associated timestamps arises. More concretely, what output timestamp should be used for an aggregate, given the timestamps of the elements contained in the pane. The value of this timestamp has a major impact on watermark progression in downstream transforms. Given the restriction that a watermark should never move backwards in time, there is a wide range of valid output timestamps for the pane. However, in practice, the following are most commonly encountered [61]:

1. **End of the window**, is the only safe choice when the output timestamp should represent the window boundaries. This is the default option implemented by most runners [40].

2. **Timestamp of first non-late element**, results in the most conservative watermarks possible. Watermark progression will likely be hindered.
3. **Timestamp of a specific element**, under the condition that the element did not arrive late. Depending on the use-case, it makes more sense to use another arbitrary timestamp.

While the Example 36 provides a thorough discussion on the propagation mechanism, propagating the watermark in this fashion introduces significant latency in the face of a grouping transform. This is especially highlighted by Figure 6.28 where the watermark is delayed for nearly an hour. Downstream components are thus substantially delayed, as the watermark is the primary driver for progress in the system. Real-world implementations therefore allow the watermark to advance at a grouping transform, even if the output was not yet materialized. Recall that, as element are processed by the components in the pipeline, it is not allowed to make late data out on non-late data[6]. If this was allowed, it would have the highly undesirable effect that a perfect watermark is able to into a heuristic watermark during processing. Allowing the watermark to advance at a grouping transform, imposes a high risk on the subject of creating such illegal late data. To appreciate this statement, consider the example below.

**Example 37.** For the concrete setting visualized in Figure 6.29, assume that the output is materialized with a timestamp that equals the timestamp of first non-late element received by the transform. For this particular scenario, the watermark is held back correctly. Though, if instead the watermark was able to advance before the output was materialized, this would render the output as late in the downstream component.  

To refrain from making late data, out of non-late data[7] the system places explicit holds on the watermark based on the output timestamp and the allowed latency[40]. These holds can be used to prove more formally that the watermark propagation mechanism is correct, i.e., no additional late data or droppable data is introduced during computation.

### 6.9 Data Latency Versus System Lag

Watermarks have provided a system to measure progress and completeness in the event time domain. In a sense, the event time watermarks relate only to the event time timestamps that are flowing through the system. This is not yet a complete picture, as this does not allow data latency to be distinguished from system latency. The idea is that, during processing, the systems introduces additional latency. For example, while communicating with an external service, when a message is stuck during delivery or when a user-defined function throws exceptions[8]. Tracking completeness solely in the

[6] Similarly, it is not allowed to make droppable data out of non-droppable data during processing.

[7] Moreover, the holds are also used to ensure that non-droppable data does not become droppable throughout the pipeline.

[8] In a streaming context, functions that throw exceptions are re-executed indefinitely as to ensure exactly-once (Section 7.5)
event time domain, does not allow a differentiation between a system that is processing data from an hour ago without delay, or a system that processes data in real-time but was delayed for an hour while doing so [61].

The concepts introduced in Section 6.8 can be adopted to the end of quantifying progress in the processing time domain. Instead of operating on event time timestamps, the ingestion timestamp is tracked. This yields a notion of processing delay, separate from data delay [61].

### 6.10 Verifying Pipeline Correctness

Testing the end-to-end operation of a pipeline is key to developing an effective data processing solution. Designing tests that cover all possible scenarios is not feasible for pipelines that use sophisticated triggering mechanisms and handle out-of-order data. Therefore, Beam provides an extensive framework that allows automated testing of pipelines. Pipelines can be tested thoroughly using the `TestStream` transform. The transform allows fine-grained control over both the advancement of the watermark and the pipeline’s processing time clock [39]. In particular, tests evaluate the effects of triggers and the respective output that they produce.

**Example 38.** Listing 6.5 presents an example of a concrete unit test for the pipeline described in Example 39. However, the implementation uses a somewhat differing triggering mechanism. First, a speculative result is emitted for each element that arrives. When the watermark surpasses the window boundary, the pipeline materializes a complete result. Late data is handled by producing an instantaneous refinement. In the test, three events are ingested in the pipeline before the watermark reaches the end of the window. The watermark is subsequently shifted beyond the window boundary, and an explicit late element is ingested.

```java
public class SampleTest {

    private static final Duration WINDOW_LENGTH = Duration.standardMinutes(2);
    private static final Duration LATENESS_HORIZON = Duration.standardDays(1);

    @Rule
    public TestPipeline p = TestPipeline.create();

    @Test
    @Category(ValidatesRunner.class)
    public void simpleTest() throws Exception {
        Instant baseTime = new Instant(0L);
        Duration one_min = Duration.standardMinutes(1);
        TestStream<KV<String, Long>> events = TestStream.create()
```
.advanceWatermarkTo(baseTime)

// First element arrives
.addElements(
  TimestampedValue.of(KV.of("laurens", 0L), baseTime.plus(one_min))
)
.advanceProcessingTime(Duration.standardMinutes(5))

// Second element arrives
.addElements(
  TimestampedValue.of(KV.of("laurens", 0L), baseTime.plus(one_min))
)
.advanceProcessingTime(Duration.standardMinutes(5))

// Third element arrives
.addElements(
  TimestampedValue.of(KV.of("laurens", 0L), baseTime.plus(one_min))
)
.advanceProcessingTime(Duration.standardMinutes(5))

// Window ends
.advanceWatermarkTo(baseTime.plus(WINDOW_LENGTH))

// Late element arrives
.addElements(
  TimestampedValue.of(KV.of("laurens", 0L), baseTime.plus(one_min))
)

// Fire all
.advanceWatermarkToInfinity();

PCollection<KV<String, Long>> userCount = p
  .apply(events)
  .apply(new CountPipeline());

IntervalWindow window = new IntervalWindow(baseTime, WINDOW_LENGTH);

PAssert.that(userCount)
  .inEarlyPane(window)
  .containsInAnyOrder(
    KV.of("laurens", 1L), // First firing
    KV.of("laurens", 2L), // Second firing
    KV.of("laurens", 3L) // Third firing
  );

PAssert.that(userCount)
  .inOnTimePane(window)
  .containsInAnyOrder(
    KV.of("laurens", 3L) // On time firing
Listing 6.5: Verifying pipeline correctness.

Apache Beam is thus an extremely powerful for both streaming and batch use cases. The fact that it allows for explicit tradeoffs between the dimensions of latency, completeness and cost allows for a wide variety of use cases to be addressed. Moreover, the fact that it exposes watermarks as a first class citizen allows for straightforward reasoning about the correctness of the results.
Chapter 7

Pipeline Execution

This chapter aims to provide insights in the pipeline lifecycle, that is, how the code describing the logical transformations in the pipeline is compiled to yield an executable, distributed program. The main problem is that this logical description does not naturally translate to an optimal execution plan. In this light, we describe a series of optimizations to increase efficiency of the program. Moreover, we define the notion of exactly-once processing and study how records are propagated through the pipeline to achieve such semantics. Understanding how exactly-once is achieved is essential to characterizing pipeline correctness.

7.1 Google Cloud Dataflow

Section 6.1 introduced the broad landscape of Apache Beam. In particular, the widespread availability of execution engines. Google Cloud Dataflow [14] is a fully-managed, cloud-based service for deploying and executing Beam pipelines. Within the sections to come, pipeline execution is studied exclusively in the context of Google Cloud Dataflow. Dataflow is responsible for parallelization, distribution and execution of the code. To this end, the service relies on several other cloud services such as Compute Engine ¹, Cloud Storage ² and Bigtable. Cloud Dataflow includes features that provide on-the-fly adjustment of resource allocation and data partitioning, through features called Autoscaling and Dynamic Work Rebalancing [10]. Moreover, another feature in favor of the service is the ability to do in-flight updates of the pipeline [18].

¹https://cloud.google.com/compute/
²https://cloud.google.com/storage/
7.2 Fault Tolerance

In the physical world it is impossible to create systems that do never break, i.e., components may often fail or break. While hardware failures are rarely encountered, distributed systems introduce exponentially more variables than a single machine does. Specifically, machines may crash or the network might become unavailable for an extended period of time. The more components involved in a distributed context, the larger the possibility that something will fail. Similarly to MapReduce, Google Cloud Dataflow executes pipelines in a fault-tolerant manner. This implies that the system transparently copes with failures, without manual intervention. Awareness of this failure model, and its implications, are key to creating pipelines that produce correct results even in the face of failures.

7.3 Processing guarantees

When streaming-systems make no guarantees about their record processing, they are referred to as best effort. Other systems offer at-least-once guarantees, thereby ensuring that records are always processed at-least-once by every step in the pipeline. When processing records with a system that provides such at-least-once semantics, duplicates might be introduced during pipeline execution. This has the effect that downstream aggregates are distorted and thereby correctness is violated.

The notion of exactly-once processing refers to at-least-once and at-most-once processing at the same time [61, 51]. Systems that guarantee exactly-once-processing essentially ensure that a record is never dropped nor duplicated [51]. However, mainly due to shared resource limitations, data that shows up late is a common phenomenon in the world of distributed systems. Consequently, the user specifies a maximum lateness for which records are accepted (Section 6.6.2). Records that arrive on time for processing are subject to the exactly-once semantics, while records that exceed this latency threshold are explicitly dropped. To illustrate all the related concepts, we introduce the following running example.

Example 39. To concretize our study in the notion of exactly-once processing, consider the pipeline defined in Listing 7.1. The pipeline computes the number of interactions received for each individual consumer. The pipeline initializes the key for each event by means of a map transform\(^3\). The events, now accompanied by a key, are subsequently collected in their respective windows. The window is provided with a trigger that fires a minute after a new event for the consumer observed. This delay allows multiple events for the same consumer to be buffered, thereby reducing the cost incurred by downstream

\(^3\)The MapElements transform is syntactic sugar for a ParDo that performs a simple function, i.e., a function that performs a single input to single output mapping.
transforms. Finally, the events are aggregated to yield the total number of interactions for each consumer.

```
PCollection<KV<String,Long>> perConsumerCount = events
    .apply(MapElements.via((Event e) -> KV.of(e.getConsumerId(), e)))
    .apply(Window.into(new GlobalWindows()))
    .triggering(
        Repeatedly.forever(
            AfterProcessingTime.pastFirstElementInPane()
                .plusDelayOf(Duration.standardMinutes(1))
        )
    ).accumulatingFiredPanes()
    .apply(Count.perKey());
```

Listing 7.1: Collecting the number of interactions per consumer.

### 7.4 Parallelism

PCollections are parallelized based on their key value, and thus form the main unit of distribution. All records with the same key are processed by the same worker instance, as result of a process called a shuffle. This process is also referred to as a group by key (GBK) in literature, as such, these terms will be used interchangeably. While this may seem limiting at first, real-world application applications generally perform computations over millions of keys and thus the parallelism that can be achieved is still massive. Data is shuffled between physical worker nodes using RPCs to ensure that records with the same key end up collectively at a worker instance [51].

**Example 40.** The pipeline described in Example 39 keys the distributed input collection according to the identifier of the consumer. The collection is subsequently re-shuffled to ensure that the records with the same keys are processed collectively. Figure 7.1 visualizes the input collection for each logical step of the pipeline. Elements in the collections are colored according to their key value.

Using RPCs for inter-machine communication may seem to be convenient at first, but this approach introduces numerous pitfalls. Martin Kleppmann elaborates on the fundamental flaws when relying on RPCs in Designing Data Intensive Applications [42]. Local function calls either succeed or fail, whereas a network request may additionally return without a result, due to a timeout. These timeouts are a manifestation of the highly unpredictable nature of network requests. Specifically, either the request or response may be lost due to network problems. Network problems are so commonly encountered that anticipation is required to secure data integrity. Google Dataflow will essentially retry the RPC until it receives a positive acknowledgement, this technique is referred
to as *upstream backup* [51]. Upstream backup guarantees at-least-once delivery of each record.

Unfortunately, network requests may fail even-though they succeeded at the receiving side. Retrying such requests introduces duplicate elements in the respective input collection. For example, when a timeout is triggered as a result of the acknowledgment being delayed due to network congestion, this scenario illustrated by Figure 7.2. The image visualizes the collections as seen by the sender and receiver respectively. A timeout is triggered due to the acknowledgment being lost, upon retrying a duplicate is added in the resulting collection. Our exactly-once semantics are clearly violated, we thus require mechanisms to detect and remove such duplicate records.
In the interest of deduplicating the elements introduced as a result of faulty RPC re-attempts, elements are augmented with a unique identifier. Moreover, the receiving side is equipped with a catalog in which the identifiers of newly arriving records are maintained. Prior to adding a new element to its respective collection, the receiver performs a lookup in the catalog to ensure the record is unique. If the element appears to be a duplicate, it is simply dropped. Reuven Lax reports that Google Bigtable is leveraged to hold the deduplication catalog \[51\]. The catalog is maintained by an highly-available, external service to maintain exactly-once semantics even in the face of worker crashes.

Having to communicate with an external service to deduplicate every record is a very costly endeavor and will lead to increased latency due to network round trip times. Moreover, the fraction of unique records arriving is generally much larger than the number of duplicates. We can greatly reduce the number of requests to the deduplication catalog by relying on a probabilistic data-structure that allows quick set membership verification.

The idea of a *Bloom filter* was fist introduced in the work by Burton H. Bloom \[27\]. At the heart of the data-structure lies an \( m \)-bit vector, denoted \( B \), and a sequence of \( k \) random hash function denoted \( h_i \) for \( i \leq k \). Each hash function \( h_i \) maps the input \( x \) to an element in the set \( \{e_1, \ldots, e_m\} \) where \( e_j \) is an \( m \)-bit vector with only its \( j^{th} \) set to 1. The output \( h(x) \), also referred to as the signature, is defined to be the logical and of \( h_1(x), \ldots, h_k(x) \). Moreover, for two \( m \)-bit vectors, \( a \) and \( b \), the notation \( a \ll b \) is used to denote that \( b \) has a 1 in each location where \( a \) has a 1. Elements are added to
the bloom filter by first computing the signature, \( h(x) \) and by subsequently setting the corresponding bits in the bit vector \( B \). Similarly, to determine whether an element \( y \) is present in the Bloom filter \( S \), compute \( h(y) \). If \( h(y) \ll B \) the element might be present in the filter, otherwise \( y \notin S \). Note that, by design, this data-structure may give false positives, but never false negatives. This is a very convenient property for our particular use-case.

**Example 41.** Figure 7.3 presents the process of adding elements to the Bloom filter graphically. The Bloom filter maintains an 8-bit vector \( B \) and uses three distinct hash functions \( h_1, h_2 \) and \( h_3 \). When adding an element, it is presented independently to each hash function. This process yields three different bit vectors, the bit vectors are combined using the logical and operations. Conversely, to verify whether an element \( y \) is present, compute its signature \( h(y) = h_1(y) \land h_2(y) \land h_3(y) \). The element might be present if and only if \( h(y) \ll B \).  

![Figure 7.3: Inserting elements in a bloom filter, where \( m = 8 \) and \( k = 3 \).](image)

The size of \( B \), the bit vector, is generally based on the expected number of elements to store and the desired false positive rate [27]. However, for many applications, it is impossible to define the number of elements a priori. To this end, there exist approaches that dynamically adapt to the number of elements while assuring a false positive rate. A prevalent example is the Scalable Bloom Filter introduced by Almeida et al [25].

Each worker instance thus maintains one of these bloom filters in memory. Bloom filters used by Google Dataflow use a static bit vector, mainly motivated by the fact that these are computationally more efficient. The catalog functions to reduce the rate at which expensive communication with the external datastore is required. When the filter reports that \( x \notin S \), the element is certainly unique. When the filter reports \( x \in S \), there is a very high chance that the element is a duplicate and this is verified using the external service.
However as time elapses, and elements are being added to the filter, the overall false positive rate starts to increase considerably. This phenomenon is also exhibited by the Bloom filter presented in Figure 7.3. To remedy this issue, Google Dataflow workers maintain a distinct Bloom filter for each 10-minute interval [49]. Figure 7.4 presents this approach graphically. To relate incoming events with their respective filter, the sender assigns a deterministic processing-time timestamp to each element. The element, along with its processing time timestamp, is persisted as part of the checkpointing mechanism. This mechanism is introduced in Section 7.5. Using distinct Bloom filters has the additional benefit that less data needs to be scanned when a new worker replaces a failed worker.

![Figure 7.4: Maintaining distinct bloom filters for every 10-minute interval.](image)

Maintains all these bloom filters forever is certainly not feasible in terms of memory usage. For this reason, a conceptual garbage collection watermark is computed based on the distribution of the processing time timestamps of the events (Section 6.9). When this watermark surpasses the interval for which the filter is maintained, its resources can be reclaimed by a garbage collection procedure.

7.5 Side Effects

Google Dataflow does not guarantee that code is run only exactly-once per record, as part of its fault-tolerant execution model [3, 51]. In this light, a user transform might be invoked multiple times for a given record [10]. Moreover, it is even possible that multiple workers might operate on the same record simultaneously [8]. This is paramount to ensure at-least once processing in the face of individual worker failures.

\[\text{Recall that, by definition, elements arriving after the allowed lateness are dropped. Keeping Bloom filter stored after the garbage collection watermark has surpassed the boundaries of the filter thus has no more use.}\]
This imposes complications in regard to side-effects that are executed as part of the user-defined processing logic. Examples of such side-effects are sending a verification mail or communicating with an external database service. These side effects might potentially be executed multiple times, which is highly undesirable. To overcome this issue, it is encouraged to make the processing logic idempotent [3, 61]. An idempotent operation is an operation that can be performed multiple times, yet having the same effect as if it was performed only once [42]. An example of an idempotent operation is writing a fixed value to a key-value store. Certainly not all operations are of idempotent nature though most operations can be made idempotent by adding additional metadata, as illustrated by the following example.

Example 42. Subtracting a defined amount from someone’s bank account is certainly not a idempotent operation. The operation can be made idempotent by adding an identifier to each subtraction request. Moreover, a record is maintained for the identifiers of the updates done to each bank account. Upon receiving a new request, the record is used to determine uniqueness of the request.

While the duplication problem from the preceding section was solved by means of a catalog, our exactly-once semantics are still not ensured. This is due to the fact that user-defined processing logic might produce non-deterministic output. More concretely, this implies that repeatedly processing the same input record may yield different output. The desired behavior is that only one of these outputs is committed to the resulting output collection. However, this requires that the output must be assigned stable identifiers. Assuring stable identifiers is exceptionally troublesome for non-deterministic transforms that produce a variable number of outputs. The following example introduces a real-world example of a non-deterministic transformation that produces output of variable size.

Example 43. Consider the setting of a social network, which offers the functionality to create status updates. When users create a new status update, their followers receive an appropriate notification. To achieve this, a pipeline has been designed that operates on the status update input stream, as visualized by Figure 7.5. Note that, the shuffling steps in the pipeline have been omitted for brevity, these exist between each of the presented steps. Upon receiving a new status update for a particular user, the Create notifications transform communicates with an external datastore to fetch the set of followers for the user. The transform subsequently creates a notification for each follower of the user in question. It is obvious that the set of followers for a specific user is dynamic, that is, changes rather quickly over time. Re-processing of records may therefore yield completely different output collections.
CHAPTER 7. PIPELINE EXECUTION

Google Dataflow addresses this issue by using a checkpointing mechanism that makes non-deterministic processing effectively deterministic \[51\]. In this light, the output for a transform, accompanied by its unique identifier, is persisted to storage prior to being emitted to downstream components of the pipeline. The upstream backup mechanism, as introduced in Section 7.3, thus replays the output that has been checkpointed upon shuffle re-attempts rather than re-running the user-defined processing logic. Note that, the user’s code might be run multiple times, but in the end, only the output of a single invocation is produced towards downstream components.

**Example 44.** To gain more insights on how these mechanisms relate to each other, let us consider a concrete example. The setting is that a sender transform receives an input element. The sender applies its transform on the element and forwards the result to the remote receiver. The incoming element is visualized by a square, whereas a circle is used to visualize the element after the transform was applied. Conform the exactly-once mechanisms, the sender is equipped with storage to maintain snapshots\[5\] The receiver, on the other hand, maintains a Bloom filter for each 10-minute interval and has an active connection with Bigtable. This setup is visualized in Figure 7.6. Before the sender forwards the element downstream, it materializes a snapshot of the element. The element is initially unacknowledged as indicated by its red background color. As discussed in Section 7.4, the sender attaches the processing time the element. This timestamp is used to relate the element to the corresponding Bloom filter at the receiver.

\[5\]Snapshots are also persisted to Bigtable to the end of maintaining exactly-once even if the sender crashes.
As the receiver receives the element, it queries the Bloom filter to see whether the element is present. The timestamp, attached to the element by the sender, is used to choose the appropriate Bloom filter for deduplication. As the Bloom filter is currently empty, the element is considered unique. It is both added to the Bloom filter and persisted in Bigtable. Persisting the element in Bigtable ensures exactly-once even if the receiver crashes. Finally, the receiver sends an acknowledgment to the sender and adds the element to the input collection. Figure 7.7 visualizes the process graphically.

Now, consider the case where the acknowledgment sent by the receiver gets lost in the unreliable network. Recall in particular the transform applied by the sender may be of non-deterministic nature. Upon detecting a timeout, the sender does not re-apply the transform but rather replays the snapshot of the element. This particular scenario is illustrated by Figure 7.8.
This has the effect that identifiers remain stable in the face of failures. Upon receiving the element, the receiver queries the Bloom filter to determine the uniqueness of the element. The Bloom filter reports that the element is non-unique. Relaying solely on the Bloom filter to determine uniqueness is unfeasible, due to the fact that it might produce false positives. Consequently, the receiver queries Bigtable to be sure that the element is indeed a duplicate and flags it accordingly. While the element is a duplicate, the receiver sends an explicit acknowledgment. This is to refrain the sender from continuing to send the element forever as to assure exactly once.

Having introduced the various measures taken to ensure exactly-once processing in the face of failures, let us now form a complete picture. Figure 7.9 presents two separate finite-state machine (FSM) descriptions for the sender and the receiver respectively. The arrows in the FSM description indicate the transition from one state to another. The event causing the transition is shown above the horizontal line labeling the transition, and the actions taken when the event occurs are shown below the horizontal line. When no action is taken on an event, or no event occurs and an action is taken, the symbol Λ is used to explicitly denote the lack of an action or event. The initial state of the FSM is indicated by the dashed arrow.

These diagrams resemble close affinity to the way the TCP-protocol operates. While Dataflow uses a custom protocol to exchange information between the sender and receiver, Apache Flink leverages TCP-connections to enable communication between different worker instances [29].
Figure 7.9: Independent state diagrams of the sender and the receiver.
CHAPTER 7. PIPELINE EXECUTION

7.6 Graph Construction

Dataflow creates an execution graph from the code that describes the pipeline as a sequence of logical steps. This includes all of the transforms and their associated processing functions. This phase is called graph construction time. During this phase, the service also performs validation and ensures that the pipeline does not contain illegal operations. The first step in composing the corresponding execution graph is expanding the various transforms. Recall that Beam offers the ability to encapsulate multiple transforms into a single, composite transform. At graph construction time, these composite transforms are essentially inlined to yield a flat execution graph. It is noteworthy that a number of SDK-provided transforms are in reality composed of multiple underlaying primitives. The graph obtained after flattening these composite transforms is referred to as the execution graph.

Example 45. Figure 7.10 visualizes the graph construction process of the pipeline introduced in Example 39. The pipeline contains a series of transformations to ingest, map, count, and write the number of interactions for each consumer. This example illustrates that the SDK-provided transform used to count the number of elements is composed using multiple primitives.

![Graph Construction Diagram](image)

(a) User-defined pipeline  
(b) Execution graph

Figure 7.10: Per consumer count execution graph.

While describing a pipeline as a sequence of logical transformations increases ease-of-development, it has the downside that it does not naturally translate to an efficient execution plan. To the end of increasing the efficiency of the resulting execution graph, let us consider some optimizations in the next section.
CHAPTER 7. PIPELINE EXECUTION

7.7 Optimizing the Execution Plan

The final order of transformations in the execution graph usually differs from the order used when specifying the pipeline. Dataflow performs various optimizations on the graph to ensure efficient execution of the pipeline. Dependencies between subsequent transforms in the execution graph are maintained during optimization. However, steps without dependencies can be executed in any order or as part of a larger, composite transformation. The most prevalent optimization techniques are fusion and multi-level combining. The ultimate goal is to minimize the amount of serialization and shuffling of data.

7.7.1 Fusion Optimization

Fusion is the simplest and most intuitive optimization, it was first introduced in the work by Craig Chambers et al [30]. The optimization essentially fuses multiple, logical steps in the execution graph into a single execution stage [49, 10]. Fusing eliminates the need of materializing every intermediate PCollection, which is a very costly operation, both in terms of memory and processing overhead. This is due to the fact that fused execution stages are run as an in-process unit and therefore data maintained to ensure exactly once is no longer required for each individual step [49]. Fusion generally reduces even the most complex pipelines to a limited number of physical execution stages. This reduces the amount of data transfer required significantly.

There are essentially two different classes of fusion as described in FlumeJava [30], namely producer-consumer fusion and sibling fusion. Figure 7.11 presents a visual representation of these classes. If one ParDo operation performs function \( f \), and its result is consumed by another ParDo operation that performs function \( g \), the two ParDo operations are replaced by a single ParDo that computes \( g \circ f \) (read: \( g \) after \( f \)).

Sibling fusion applies when two or more ParDo operations read the same input collection. These operations are fused into a single ParDo operation that computes the result of all the fused operations in a single pass over the output.

7.7.2 Combine Optimization

Grouping transforms are an important concept in the context of distributed data processing. Apache Beam offers various aggregating primitives such as the GroupByKey, CoGroupByKey and Combine transforms [3]. Additionally, there is a number of SDK-provided transforms that are composed of these aggregating primitives, such as the CombinePerKey transform. All such primitives combine data over the entire input collection, including data that may be distributed across different workers. During these
aggregations, it is much more efficient to perform partial combining locally prior to sending data to the main grouping operation [49]. This optimization was first introduced in MapReduce [34]. Multi-level combining considerably reduces the number of messages for delivery, and thereby the number of reads and writes to persistent storage.

**Example 46.** The relevance of multi-level combining is best perceived in the context of a concrete example. Analogous to the pipeline in the running example, the pipelines from Figure 7.12 perform a counting operation. To simplify the discussion, assume that we are dealing with a single key. Figure 7.12a visualizes execution without the combine optimization. This has the implication that each element has to be send explicitly to the combine transform. Instead of sending each element individually, the pipeline in Figure 7.12b performs the count locally and sends the local result. The commutative and associative nature of the transform enables these local results to be combined into a single, global result.

The optimization techniques presented above are combined to yield an efficient execution plan. In particular, this execution plan minimizes serialization and the costly process of shuffling data. Let us apply both techniques on the consumer count pipeline, as introduced in the running example.

**Example 47.** Figure 7.13 presents the graph at the successive stages during graph construction and optimization. Relations between individual stages explicitly annotated using dashed lines. The leftmost column of the figure represents the execution graph obtained after the graph construction phase. The middle column presents the steps added due to ensure exactly once. In particular, data is materialized, shuffled and read from state in the face of a failures. The rightmost column presents the graph after optimization. Recall that the pipeline of the running example performs a combine.
(a) Without multi-level combining.

(b) With multi-level combining.

Figure 7.12: Relevance of using multi-level combining on a combine transform.

transform, the multi-level combine optimization is hence applicable. The steps before respectively after the GKB-step are fused into single, physical execution phases.
Note in particular that it is not possible to fuse the group by key transform. As discussed, this requires the data to be re-shuffled across physical worker instances. Generally speaking, most transforms in between GKB transforms are merged into a single execution phase. This is also illustrated by Figure 7.13. The optimizations presented may fuse an arbitrary number of non-deterministic transforms into a single execution phase. The fact that it is impossible to fuse the GKB operation, provides a notion of a non-deterministic barrier. In particular, only one of the outputs from the upstream, non-deterministic stages is allowed to pass this non-deterministic barrier. This is due to the fact that the snapshotting mechanism is used at the boundaries of each execution stage. Thus instead of reprocessing elements in upstream transforms, the elements are replayed from the snapshot in the face of failures. For downstream stages, this creates the illusion as if the pipeline is processing deterministically.

7.8 Bundling of Elements

The inter-machine communication is a very costly operation in the distributed execution of the pipeline. The operation requires the elements to be serialized, persisted and transported between different physical worker instances. Avoiding this serialization may require re-processing in upstream transforms in the face of failures, which is not feasible.
for unbounded input sources. Moreover, checkpointing the progress of the pipeline is paramount to ensure exactly-once processing. As described in Section 7.4, inter-machine communication happens as part of the shuffle process.

Processing each element in parallel has some noticeable drawbacks. It makes it impossible to batch pipeline operations, for example writing elements to a sink or checkpointing progress during processing. To overcome this issue, the elements in the collection are accommodated in bundles, which form the unit of serialization and communication. There is an inherent trade off between the number of elements that are persisted collectively and the number of elements that have to be re-processed if there is a failure. The way the element collection is subdivided in bundles is arbitrary and runner-specific. Streaming runners commonly prefer low-latency and thus persist small bundles while batch runners, on the contrary, prefer high-throughput and choose to operate on larger bundles. Note that bundles are not stable, bundles are subject to change throughout the execution of the pipeline.

**Example 48.** Figure 7.14 presents a bundled version of the consumer count pipeline. Observe how elements are accommodated in bundles. These bundles are not static and vary highly during execution of the pipeline.

![Figure 7.14: Bundling in the consumer count pipeline.](image)

To the end of minimizing inter-machine communication, consumer-producer fusion was introduced in Section 7.7.1. The different transforms that are fused in this process are referred to as dependently parallel. Recall that execution stage are executed as an in-process unit. Fusing two transforms in a single execution stage thus implies that the output of the producing transform, for a given element, must be processed on the same worker.
Example 49. Figure 7.15 visualizes two dependently parallel transforms. The element color encodes a parent-child relationship in consecutive collections. Worker 1 initially executes Transform A on its input collection and subsequently executes Transform B on the resulting output collection. Worker 2 behaves in a similar fashion.

Figure 7.15: Handling elements in dependently parallel transforms.

Individual elements form a indivisible unit. In theory, the maximum level of parallelism is thus limited by the number of elements in the collection. However, fusing transforms into a single execution stage constrains the parallelism that can be achieved. This is because the parallelism is now limited to at most the number of items in the initial input collection. This can be exceptionally troublesome for high fan-out operations, i.e., operations the produce thousands of outputs for a single input element. The pipeline described in Section 43 presents a real-world example of an high-fan operation. Whenever a celebrity updates their status, a single element in the input collection may potentially produce thousands or even millions of output elements.

7.9 Handling Failures

Recall that user-defined processing logic is passed to a ParDo invocation. This user-defined logic may fail due to many reasons, e.g., communication with an external service that is temporarily unavailable, attempting to parse an invalid element, and so on. When processing elements in bundles, the general idea is that if a single element in a bundle
fails, the entire bundle fails. This implies that all elements within the bundle have to be re-computed. To guarantee completeness of the pipeline, bundles have to be retried or the entire pipeline fails. Upon re-executing a bundle, bundles are not guaranteed to be stable, nor are the bundles necessarily re-processed on the same worker. Figure 7.16 presents this graphically. Initially, a bundle is scheduled for processing on Worker 2, however due to the failure of a single element the entire bundle fails. The failed bundle is thereafter scheduled for re-execution on Worker 1.

When executing a pipeline using Dataflow backend, the re-execution mechanism of bundles depends on the mode of execution. When running in batch mode, bundles are retried for four times before the pipeline fails completely. For stream pipelines, on the other hand, bundles are retried indefinitely. This may result in permanent stall of the pipeline [10]. This is an implication of the at-least-once semantics offered by the runner and can be observed by examining the system lag (Section 6.9) of the pipeline.

Recall that, in the context of fused transforms, data to ensure exactly-once is no longer maintained at the boundaries of each individual transform. Instead, exactly-once data is stored at the boundaries of the fused stage. In particular, this implies that elements are not snapshotted in between transforms, thereby requiring the entire phase to be re-executed in the face of a failure. The Apache Beam Execution Model refers to the transforms that are part of the fused stage as co-failing [2].

### 7.10 Sinks

The output of large-scale data processing is eventually sent to another system, for example for consumption by external services. Delivering data to such an external service is essentially a side effect. The exactly-once semantics do not hold for such side effects, as explored in Section 7.5. The best approach is to ensure that the side-effect operation is idempotent and therefore robust in the face of replay [50]. This approach will fail
if there is non-determinism involved. The solution of this problem is to re-shuffle the data prior to invoking the sink transform. As discussed, re-shuffling the data creates an implicit non-deterministic barrier due to the snapshotting mechanism. In particular, the output from step-to-step is check-pointed to storage to ensure that the generated records are stable. In the face of failures, data replayed from the checkpoint as illustrated by Example 44.

Example 50. Figure 7.17 presents a pipeline that writes output to a Pub/Sub topic. Transforms preceding the Format transform are assumed to be non-deterministic. The explicit re-shuffle assures that data is check-pointed in the format stage. If elements in the Write transform fail, the elements materialized in the snapshot are replayed.

![Figure 7.17: Exactly once when producing to Pub/Sub.](image)

7.11 Sources

The study of how exactly-once is maintained during pipeline execution does not yet form a complete picture. The question naturally arises how exactly-once is achieved within the sources and sinks that are part of the pipeline. For sources in particular, this implies that the processing backend needs to ensure that every record produced by the source is processed exactly-once. However, to execute pipelines in a fault-tolerant manner, the engine might retry reading from a source in the face of failures. This is a trivial problem when reading from a deterministic input source, e.g., a file or a Kafka topic. When reading from such source the service can infer deterministic record identifiers using a combination of the name and offset.

For non-deterministic input sources, for example a Google Pub/Sub subscription, this is not feasible. This non-determinism originates form the fact that Pub/Sub will re-
deliver messages to different worker nodes in a completely different order. Inferring message identifiers for records delivered by this service is thus impossible, as these are highly unstable. The only way the execution engine is able to deduplicate records from a non-deterministic input source, is when the source itself provides stable identifiers upon ingestion in the system. Luckily, Cloud Pub/Sub augments the message with a stable identifier, this identifier remains unaltered upon redelivery. Analogous to ensuring exactly-once at sinks, an explicit re-shuffle of the data is performed to ensure that records with the Pub/Sub identifier are processed by the same worker instance. This re-shuffle is performed based on the Pub/Sub identifier of the message.

Example 51. Figure 7.18 visualizes this graphically, only components relevant to this discussion are displayed. The annotations on the elements represent the Pub/Sub message identifier. An element is delivered twice to different worker instances of the pipeline, however, due to the re-shuffle the Parse transform is able to detect the duplicate element.

![Figure 7.18: Deduplicating Pub/Sub Messages on ingestion.](image)

As an additional remark, consider the case where a pipeline \( A \) produces to a Pub/Sub topic and another pipeline \( B \) reads from the topic. In this case, it is still possible that exactly-once is not maintained. Indeed, while using an explicit reshuffle near the sink assures element stability, it is still possible that the same element is produced to Pub/Sub twice. The second pipeline is unable to detect this, given the fact that the Pub/Sub message identifier is assigned upon ingestion in Pub/Sub, and these identifiers will thus be unique. It is possible to perform the shuffle near the source based on another attribute,
rather than the message identifier. To yield an end-to-end reduplication procedure, assign a custom Pub/Sub message attribute at the Format stage and use the custom attribute to perform the deduplication at the source.

This still does not yield any guarantees, as Dataflow maintains a Bloom filter for each 10 minute interval as part of the deduplication mechanism introduced in Section 7.4. Recall, in particular, that a deterministic timestamp is used to correlate elements with their respective Bloom filter. The Pub/Sub ingestion timestamp for the duplicate element differs, and it is thus perfectly possible that each element ends up in a different Bloom filter.

The system could make temporal guarantees about the deduplication procedure by forcing the Bloom filters to overlap. For instance, if the Bloom filters have a 10-minute overlap, the system can guarantee that duplicate elements, within a 10-minute interval, are detected and eliminated. It is noteworthy that, at the time of writing, it is uncertain whether this overlap is implemented in Dataflow.

7.12 Dynamic Autoscaling

Autoscaling is a feature of Dataflow that automatically chooses the number of workers required to execute a pipeline. This feature allows the number of workers to be dynamically reallocated during runtime. Recall that interaction data is subject to high variations in input rate. Selecting a fixed number of workers has the implication that the job is either over-provisioned or under-provisioned at a particular moment in time. This is illustrated by Figure 7.19b and Figure 7.19a respectively. The red line on the figure represents the actual workload, and the available resources are visualized by the grey area. Provisioning too many works incurs unnecessary cost, whereas under-provisioning yields higher latency of results.

By using autoscaling, the pipeline can adaptively change the number of workers to changes in load and resource utilization. Scaling is done based on several metrics, such as CPU utilization, throughput and the amount of work remaining. For our concrete setting, the work remaining is the number of unprocessed messages in Pub/Sub. The general idea is to minimize the work remaining, while maximizing worker utilization and throughput.

Another closely related problem is configuring the right number of partitions for a given dataset. Traditional systems require the number of partitions to be specified prior to executing the pipeline. Excess in partitions will result in partition overhead, whereas

---

*Note that, assigning a timestamp for deduplication at the sink is not possible because Bloom filters are not maintained indefinitely, i.e., their state is automatically reclaimed by a garbage collection procedure.*

124
using too little partitions will limit the level of parallelism that can be achieved. In addition to dynamically adding workers to a given job, Dataflow also adjusts the number of tasks to maximize worker utilization \cite{30}. This concept is referred to as dynamic work rebalancing \cite{10, 41}.

### 7.13 Conclusion

While the Beam Model offers exactly-once semantics, the pipeline designer has to be aware of how the pipeline is executed in order to design a pipeline that produces correct results. The most prevalent issue is that the pipeline does not guarantee exactly-once execution of the user-defined processing logic, which has the result that side effects have to be idempotent. This certainly impacts the complexity of the processing logic, as discussed in Chapter 8.
Chapter 8

Send Time Optimization

The moment in time at which the audience is approached has a major effect on the impact of the marketing campaign. While reporting and analysis may assist the marketeer in defining general guidelines on when to approach the audience, the process of choosing when to approach the consumer is especially tedious. The optimal send time is highly consumer specific, and prone to rapid change. As such, for most audiences, it is often intractable to infer a set of rules to determine the optimal send time. This chapter discusses the implementation of send time optimization, that is, a feature devoted to establish the most appreciate moment in time to approach each individual consumer automatically.

**Disclaimer** We stress that the focus of this chapter is not placed on the complexity of the algorithm, but rather on the implementation of the pipeline. This work therefore uses a relatively simple model to implement send-time optimization. The pipeline described in this chapter is by no means associated with the send-time optimization provided by the Selligent Marketing Cloud.

8.1 Introduction

Selligent Marketing Cloud endeavors to maximize the overall open-rate of outbound marketing campaigns. One of the most prevalent tools used for this purpose is deferred launch, as discussed in Section 2.3.4. By analyzing the interaction data, the organization can make certain predictions about the optimal send time for the audience. Deferred launch is leveraged to send the content to the users at the predicted optimal send time. Establishing the send time for an audience manually is both time consuming, and yields highly inaccurate results. Recall that the interaction data can also be leveraged to
Create entirely new features that support the marketeer in creating even more valuable interactions.

Send time optimization (STO) attempts to deliver content to consumers at the exact time of the day that it is most likely to be opened. The time of the day that maximizes the open-rate is highly specific to each individual consumer. Past behavior is used to the end of predicting the optimal send time. Specifically, the decision on when the content should be delivered to a consumer, is based on the most frequent open time observed for that particular consumer. Send time optimization can be used to send content to audiences containing millions of consumers, each receiving the content at their optimal time of day, through a single click.

Recall that the mission of Selligent Marketing Cloud is to provide omnichannel marketing solutions. Given this mission, the functionality of STO should not be restricted to predicting solely the optimal point in time. Additionally, the feature should be able to determine the most appropriate communication channel to approach the consumer at hand. Here too, interaction data gathered by the OptiExt module is leveraged. From here on, send-time data refers to the combination of the optimal point in time and the most appropriate channel to approach a consumer.

### 8.2 Consumer Profiles

To determine send-time data for a consumer, examining all historical records is not feasible due to the unbounded nature of the interaction dataset. Instead, a detailed profile is maintained for each individual consumer. An example of such profile, in JSON format, is presented in Listing 8.1. For brevity, only the most relevant attributes are presented. The firstEventTimestamp and lastEventTimestamp attributes refer to the event time timestamps of the first respectively last event observed for the consumer. The lastUpdateTimeStamp attribute indicates the processing time timestamp of the last profile update. Past behavior of the consumer is tracked in the activity dictionary. The dictionary maintains 24 buckets, one for each hour of the day, for each relevant communication channel. Each bucket records the number of interactions, for the specific channel, on the respective hour. Upon receiving a new event, the count within the corresponding bucket is incremented.

```json
{
  "firstEventTimestamp": 1557670359047,
  "lastEventTimestamp": 1557785479771,
  "lastUpdateTimeStamp": 1557785579747,
  "activity": {
    "mob": [5, 8, 5, 5, 9, 8, 6, 9, 9, 5, 6, 2, 3, 3, 4, 3, 2, 4, 3, 4, 3, 3, 3, 7]
  }
}
```
When launching a marketing campaign with send time optimization enabled, the execution engine consults the consumer profile to establish the send-time data. The send-time data is subsequently used to schedule the specified content for delivery.

### 8.3 Implementation

Streaming systems provide an excellent candidate to approach the computation of consumer profiles. These systems are particularly attractive due to their inherent ability to deal with unbounded datasets, and their means to provide continuous refinements of results. Providing timely refinements of results is especially relevant in the concrete setting of STO, where a newly arriving event may potentially lead to different send-time data for a particular user. Another alternative would be to schedule a periodical batch job, i.e., a daily job to update consumer profiles. The batch job processes the dataset by slicing it into finite chunks for processing. The latency incurred by batch jobs, together with their inability to cope transparently with unbounded datasources, makes them less attractive. For this reason, we focus exclusively on streaming pipelines. The following sections delve into possible implementations of the STO pipeline, elaborates on the correctness and evaluates the dimensions of performance, latency and cost.

#### 8.3.1 Ingestion

Events are ingested in the pipeline through Google Cloud Pub/Sub. Recall that the services guarantees at least-once delivery semantics, implying that duplicates might potentially enter the system. Duplicates may result in less valuable output produced by the pipeline and are thus best eliminated. The measures taken to ensure exactly-once during processing, as introduced in Chapter [7](#) can be exploited to yield an automatic deduplication procedure. The Pub/Sub adaptor enables this functionality through the `withIdAttribute` property. More concretely, when producing a message to Pub/Sub, add a unique identifier to the attributes of the message. When ingesting the elements in the

---

1. Alternatively, the internal message identifier of Pub/Sub can be used to the end of deduplicating the events.
pipeline, specify the attribute that uniquely identifies the message, this identifier is subsequently used to deduplicate incoming elements. The Beam code in Java is presented in Listing 8.2.

Listing 8.2: Deduplicating messages from Pub/Sub upon ingesting.

8.3.2 Global Windows

The first, and most straightforward, implementation of the pipeline employs global windowing as described in Section 6.6.1. The windowing technique is combined with a repeated update trigger (Section 6.6) that emits a pane upon receiving a new element. Consecutive panes are merged according to the accumulation strategy described in Section 6.7. The implementation is conceptualized by Figure 8.1. The idea is that the global window maintains the profile for each consumer indefinitely, as a new event is observed it is added to the profile and a refinement of the original result is produced. An individual event is represented by a square, whereas a consumer profile is visualized by a circle.

Figure 8.1: Conceptualizing the global window approach.

Judging pipeline correctness is trivial as a complete result of the consumer profile is persisted to storage, i.e., the side effect is idempotent. Given the broad range of consumer identifiers and the bursty nature of the workload, maintaining the state for all profiles in-memory is not feasible nor cost effective. Moreover, upon restarting the pipeline, data residing in memory is lost and profiles have to be recalculated.
8.3.3 Internal State

Maintaining every profile in-memory indefinitely is not feasible for the expected workload. Instead, a possible solution is to leverage the stateful processing feature of Beam. When employing stateful processing, a ParDo invocation may optionally claim durable, consistent state [45]. Stateful processing occurs within the context of a single key and window. It is the responsibility of the runner to persist state in a timely manner and to control the access to it. Specifically, the runner performs additional data shuffling and synchronization in order to avoid concurrent access to state cells [45]. Figure 8.2 presents the pipeline when relying on internal state.

![Figure 8.2: Conceptualizing internal state.](image)

Conceptually, this method resembles close affinity to the global window approach. Incoming events are passed to the `MergeWithInternal` stage. The stage reads the profile from durable storage and augments it with the event before persisting it again. The stage also emits the resulting profile in the interest of persisting it in Datastore for consumption by external services, e.g., the execution engine. The code of the stateful ParDo is Listed in Listing 8.3. State cells are declared and configured through the attributes of the `StateSpec` type. During element processing, access to state is granted through the read and write methods respectively.

```java
public static class MergeWithInternalState extends DoFn {

    @StateId("consumerProfile")
    private final StateSpec consumerProfileSpec = StateSpecs.value();

    @ProcessElement
    public void processElement(ProcessContext c,
                               @StateId("consumerProfile") ValueState profileState) {
        ConsumerProfile consumerProfile = profileState.read();
        if(consumerProfile == null) {
            consumerProfile = new ConsumerProfile();
        }
    }
```

130
15 ConsumerProfile newProfile =
16    consumerProfile.merge(c.element().getValue());
17
18    c.output(KV.of(c.element().getKey(), newProfile));
19
20    profileState.write(newProfile);
21 }
22 }

Listing 8.3: Using internal state.

The main drawback of using internal state is the fact that it is scoped to the lifetime of the pipeline. If the pipeline is ever to be restarted, the durable state becomes unavailable.

8.3.4 External State

Instead of attempting to store entire consumer profiles internally, one could alternatively opt for a load-and-store approach using Datastore. Specifically, the pipeline combines solely the most recent events for a consumer in a delta profile and a downstream transform merges the delta profile with the profile residing in Datastore. The pipeline is conceptualized in Figure 8.3. Light-colored circles represent delta profiles, whereas the dark circle represents a full profile. Recall that interaction data is a highly spiky workload, i.e., consumers issue a limited number of events before going inactive for an extended period of time. Doing a fetch-merge-store cycle for each individual event that arrives is an expensive endeavor, as it involves high-latency network communication. Therefore, in contrast to the techniques outlined above, this pipeline employs session windowing (Section 5.7.2). Session windows seamlessly encapsulate such spikes, for a particular consumer, in a single window. Grouping activity bursts in a partial profile limits the number of Datastore requests, thereby reducing cost.

Figure 8.3: Conceptualizing external state.

The MergeWithExternal is an example of a non-idempotent operation, i.e., re-applying the operation does not yield the same result. Judging pipeline correctness now becomes very non-trivial, due the combination of the fault-tolerant execution model and the non-idempotent nature of the side effect. This is illustrated by the following example.
**Example 52.** Consider the case where a session that contains five events is produced downstream in the pipeline. The `MergeWithInternal` transform retrieves the old profile, merges the result and persists the resulting profile to Datastore. Now, due to unfortunate events, a new session is produced for the same consumer just before the new profile is materialized. A distinct worker instance fetches the outdated profile and merges it with the newly arrived session. The worker subsequently persists their version of the profile to Datastore, yielding an inconsistent state. In particular, the five events from the first session for the consumer are lost. Figure 8.4 concretizes this setting by means of a system sequence diagram.

![System sequence diagram visualizing how concurrent access to Datastore may result in an inconsistent state.](image)

The race condition described in Example 52 can be addressed by using transactions. Specifically, both the fetch and write calls are encapsulated within a single database transaction. Using a transaction essentially locks the profile during the merge procedure, thereby rejecting other entities from accessing it. There is still a pitfall that is not covered by employing transactions, namely re-execution in the face of failures. Specifically, while Beam guarantees exactly-once-processing for elements, it does not guarantee that user-defined processing logic is run only once per record. Consider the example below.

**Example 53.** Recall that, for efficiency considerations, elements are accommodated in

---

\textsuperscript{2}It is noteworthy that in earlier designs of the pipeline, the read and write steps were implemented in isolated transforms, if these transforms are not fused (Section 7.7.1), this leads to similar issues.
CHAPTER 8. SEND TIME OPTIMIZATION

bundles when transferred between subsequent transforms. Section 7.8 provides a more in-depth discussion on bundles and their implications. The crucial takeaway is that, when a single element in the bundle fails, the entire bundle fails. The failed bundle is rescheduled, meaning that all elements in the bundle are reprocessed. This also applies to the delta profiles, a bundle containing an arbitrary number of elements is passed to the \texttt{MergeWithInternal} Function. If processing for one of these profiles fails, the entire bundle is retried and already merged profiles are thus accounted twice\footnote{Even when the bundle does not fail, there is still a risk of accounting profiles twice. This is due to the fact that the runner might invoke multiple DoFn instances to process the same bundle simultaneously as part of its fault-tolerant execution model \cite{8}.}.  

The issue outlined above is best addressed by making the operation idempotent. As discussed in Section 7.5, the majority of operations can be made idempotent by adding additional metadata. This is also true for our particular use case. The timestamp representing the end of the session window, denoted \( w_{end} \), can be used for this purpose. More concretely, when merging a profile, the transform stores \( w_{end} \) inside the \texttt{lastUpdateTimestamp} attribute of the profile. To deduplicate future profile updates, the transforms compares the current \( w_{end} \) with the value of \texttt{lastUpdateTimestamp}. Updates attempting to alter profiles where \( w_{end} \leq \texttt{lastUpdateTimestamp} \) are dropped.

Using the end of the window to deduplicate profile updates is correct only when the timestamp is monotonically increasing. This is trivial when processing in the processing time domain. For event-time windowing, however, this might not always be the case due to late data. If the allowed lateness (Section 6.6.2) is non-zero, late panes may produce refinements of original results. If the window in question already produced an on-time pane, the refinement will be dropped incorrectly by the described mechanism\footnote{By definition, the same issue arises when using early firings.}. Consider the simplified scenario below.

\textbf{Example 54.} For simplicity, consider fixed windowing, in the event time domain, with a duration of two hours. Panes are emitted by means of a completeness trigger and refinements are emitted instantaneously. Moreover, the allowed lateness property is set to one hour. For a concrete setting, this yields a time domain mapping as presented in Figure 8.5. Events \( e_1, e_2 \) and \( e_3 \) are received in time and are emitted as part of the on-time pane \( A \) for window \([10:00, 12:0)\). As processing time elapses, late elements \( e_4 \) and \( e_5 \) arrive within the lateness horizon for window \([10:00, 12:0]\). The elements have the effect that the late panes \( B \) and \( C \), with identical window boundaries, are materialized. Panes \( B \) and \( C \) are dropped by the deduplication mechanism and their respective events are therefore lost.

Since our implementation uses processing time windowing, there is no notion of late data. If event-time windowing is required for a particular use-case, one could instead rely on the current time, according to the system clock of the worker instance, to de-duplicate...
CHAPTER 8. SEND TIME OPTIMIZATION

Figure 8.5: Successive panes yield non-monotonically increasing timestamps due to late firings.
profile updates. This, however, requires an explicit re-shuffle to yield stable timestamps in the face of failures. Section 7.10 discusses how and why re-shuffling ensures stability of these timestamps.

### 8.3.5 Cloud Functions

A *Cloud Function* is a small, independent unit of processing logic focused on a single responsibility [15]. Cloud Functions are serverless, highly available and scale automatically. A Cloud Function is commonly invoked in response to specific events. For our particular use-case, it suffices to know that a Cloud Function can be configured to runs whenever an event is produced to a particular Pub/Sub topic. These functions are attractive due to their pay for what you use model, i.e., cost is incurred only when the function runs.

The measures taken to ensure correct execution of the former pipeline may translate to a noticeable performance bottleneck. A possible solution is to push the expensive merging process out of the pipeline and make a Cloud Function responsible for performing the heavy lifting. Instead of merging the profiles as part of the pipeline, the partial profile— together with its corresponding metadata—is produced to a distinct Pub/Sub topic. In response to this event, a Cloud Function is invoked that merges the partial profile with the profile housed in Datastore. Figure 8.6 conceptualized the topology of this pipeline.

![Figure 8.6: Conceptualizing the Cloud Functions approach.](image)

The at-least-once, out-of-order delivery model offered by Pub/Sub makes the process of merging profiles tedious. Specifically, Pub/Sub might deliver duplicate delta profiles and, additionally does not guarantee that messages are delivered in order. This implies that the deduplication mechanism introduced in Section 8.3.4 is no longer feasible. Example 55 provides a concrete example.

**Example 55.** The pipeline produces a delta profile, along with the timestamp that denotes the end of the window, to Pub/Sub. At roughly the same time, a series of new
events are observed for the same consumer. Once again, the pipeline combines the events in a delta profile, adds the associated timestamp, and delivers it to Pub/Sub. If the second profile is processed first, the first profile update is rejected by the deduplication profile. This implies that the events from the first profile are lost.

To correctly deduplicate profiles updates in such environment, explicit bookkeeping of the Pub/Sub message identifiers is required. Moreover, the Cloud Function needs to be maintained, the execution has to be monitored and they increase the overall cost.

8.3.6 Hybrid Approach

Internal state, as introduced in Section 8.3.3, is scoped to the lifetime of the pipeline. External state, on the other hand, requires thorough coordination to yield correct results. The notions of internal state and external state can be combined to yield a more sophisticated implementation. This approach is conceptualized by Figure 8.7. When the MergeWithInternal transform receives a delta profile from a consumer, it checks the internal state to verify whether the profile from the consumer is present. If the profile is not available, the transform fetches the remote profile from Datastore and caches it in its internal state. When the profile is available, the transform merges it with the delta and overwrites the internal copy. Additionally, the transform persists the full profile in Datastore. Note, in particular, that writing the full profile to Datastore is an idempotent side-effect.

This version of the pipeline relies heavily on the fact that the runner performs additional re-shuffling and synchronization to avoid concurrent access to state cells. Moreover, caching profiles internally works only under the assumption that no external systems modify the version residing in Datastore. If other systems do alter the profiles, this approach will result in inconsistencies and should not be used. Moreover, keeping the
profiles cached during the entire lifetime of the pipeline might not always be feasible as the internal resources are subject to limits.

### 8.3.7 Hybrid Approach with Cache Invalidation

The internal resources of the pipeline are not infinite, as discussed in Section 8.3.6. In complement to stateful processing, Beam also provides a notion of timers. Timers can be used to allow more fine-grained control over the computation, as they empower the programmer to set a timer at a particular point in the future. Upon expiring, the timer triggers a user-defined callback [44]. For example, these timers can be used to author invalidation workflows and to construct even more sophisticated triggering mechanisms. In the context of send-time-optimization, they can be used to allow even more control over the dimensions of completeness, latency and cost. Figure 8.8 visualizes the pipeline with state and timers conceptually.

The approach does no longer rely on session windowing, but instead employs global windows. When an element arrives at the `MergeWithInternal` transform, the profile is fetched from Datastore and cached in the internal state. Upon persisting the profile in internal state, the pipeline also schedules a timer that is used to the end of evicting the profile out of memory. The profile is materialized in Datastore only when the callback is invoked, and resources are subsequently reclaimed. Moreover, in addition to persisting the profile in a state cell, the transform also keeps explicit track of the number of updates done to the profile. The pipeline writes an intermediate version of the profile to Datastore when the number of modifications reaches a user-defined threshold.

![Figure 8.8: Conceptualizing the hybrid approach with cache invalidation.](image-url)
While this approach bounds the duration that profiles are cached in memory, it is still not assured that the internal state does not eventually fill up. At the time of writing, it is still unknown what the implications are when the internal state becomes saturated. This requires further investigation and this implementation is therefore purely experimental. Note that, analogous to the approach described in Section 8.3.6, this version assumes that no external systems modify the version residing in Datastore.
Chapter 9

Experimental Results

To gain insights on the performance of the pipelines under specific and anticipated loads, a series of load tests were performed. Each of the pipelines described in Section 8.3 have been implemented and tested thoroughly. This chapter provides a exhaustive overview on the test cases performed, along with the most significant findings.

9.1 Setup

The Google Cloud Stackdriver environment was leveraged to visualize the test results. This is a hybrid cloud solution that provides visibility in performance, uptime and overall health of cloud-powered applications. Moreover, the Stackdriver Monitoring agent was installed on the worker instances to gather run-time statistics. The agent provides us with metrics such as CPU utilization and memory usage. Key performance indicators used are, among others, CPU utilization, CPU utilization, system lag, the number of Datastore operations and the number of unacknowledged messages in Pub/Sub.

To test the systems under realistic workloads, Selligent developed a tool that is able to simulate random audiences. The tool is named Kuick and can be configured by specifying the number of tenants to simulate, along with the number of consumers per tenant. Kuick also provides controls to limit both the number of events per second and the total number of events to produce. Moreover, to reach the desired rate of events, Kuick was ran via the Cloud Run environment.

[1]https://cloud.google.com/monitoring/
CHAPTER 9. EXPERIMENTAL RESULTS

9.2 Metrics

The run-time statistics of the pipelines are visualized in a dashboard, for which structure is presented in Figure 9.1. The choice of a dashboard was done to aid the process of identifying correlations between the metrics. The defined metrics are all sampled every 60 seconds and their semantics are as follows:

A. The number of vCPUs currently allocated to the Dataflow Pipeline. Specifically, it is the current number of workers times the number of vCPUs per worker.

B. The current maximum duration that an item of data has been awaiting processing, in seconds (Section 6.9).

C. The age (in event time) of the most recent item of data that has been fully processed by the pipeline (Section 6.5).

D. Number of unacknowledged messages in the subscription used to feed the pipeline.

E. The percentage of CPU currently in use.

Figure 9.1: Structure of the load testing dashboard.

---

https://cloud.google.com/monitoring/api
CHAPTER 9. EXPERIMENTAL RESULTS

F. Percentage of memory used by each vCPU.

G. Count of Datastore API calls, grouped by API method (lookup, commit, abort).

H. Number of elements added to each PCollection of the pipeline so far.

I. Cumulative count of publish requests on the load test topic, grouped by result.

J. Time Spent In Disk Operations, averaged.

K. The total number of collections that have occurred.

L. The accumulated collection elapsed time in milliseconds.

The metrics above the dashed line are presented for each test, whereas the metrics below the line are included only when the are believed to be relevant.

9.3 Throughput Testing

In the first series of tests, the test scenario consisted of pushing a predefined number of events in the pipeline and observe how they effect the aforementioned metrics. In these tests, both the audience size and the rate of events have been increased. Experiments were also ran using different machine types. Finally, the autoscaling was enabled to investigate how efficiently Google Dataflow re-allocates the number of workers dynamically. Note that, the pipelines were tested isolated from each-other, to avoid concurrent Datastore requests.

Pipeline specific settings were fixed for each test performed. For pipelines that employ session windows, the gap duration was set to 2 minutes. Moreover, the pipeline with cache invalidation produces a speculate result after each 100 updates to the profile, and the invalidation parameter was set to 5 minutes.

9.3.1 Scenario 1

In the first test scenario, a number of 200 000 events were generated at a rate of approximately 2000 events per second. Kuick was configured to simulate an audience containing 10 000 consumers of a single tenant. The n1-standard-1 machine type was used, that is, hardware with a single vCPU and 3.75 GB of memory. The maximum number of workers was constrained to 1.

https://cloud.google.com/compute/docs/machine-types
CHAPTER 9. EXPERIMENTAL RESULTS

Test Summary

Kuick Configuration

1. Events: 200 000
2. Rate: 2000 events/second
3. Tenants: 1
4. Audience: 10 000 consumers/tenant

Pipeline Configuration

1. Machine type: n1-standard-1
2. MaxWorkers: 1

Figure 9.2: Load test results global windows.

In the pipeline the employs global windowing (Figure 9.2), the number of unacknowledged messages ramps up rather quickly. With approximately 170 000 unacknowledged messages at the maximum. Moreover, the lag introduced by the system is about 10 minutes at peak time.

The system lag measured when using internal state (Figure 9.3) is significantly lower,
CHAPTER 9. EXPERIMENTAL RESULTS

Figure 9.3: Load test results internal storage.

Figure 9.4: Load test results external storage.
CHAPTER 9. EXPERIMENTAL RESULTS

Figure 9.5: Load test results cloud function.

Figure 9.6: Load test results hybrid approach.
with 2.5 minutes at peak. To process the 200,000 elements, the pipeline took solely 3 minutes in contrast to the 10 minutes when relaying on global windows. When examining the elements in the input collections, the number seems to stagnate for some time before going up again. This might be due to the scheduling overhead to avoid concurrent access to state cells. The other implementations exhibit similar behavior, with the hybrid approach giving the best results in terms of system lag and backlog size.

The dashboard for external state is presented on Figure 9.4. The peak on the Datastore requests is now shifted 2 minutes to the right, corresponding to the gap duration of the session windows. As the pipeline materializes the windows in Datastore, the System lag drops and the CPU usage increases.

Figure 9.5 visualizes the tests results when pushing the expensive merging logic out of the pipeline. The Datastore requests are now delayed, due to the delay of the Cloud Function invocation. The peak on the Datastore requests is remarkably symmetric.

For the Hybrid approach with timers (Figure 9.7), one can clearly observe a long tail on the number of unacknowledged messages in the Pub/Sub subscription. Investigating the operations on Datastore shows, that during this tail, the pipeline ceases to fetch new profiles. Only after the invalidation timers fire, and the pipeline starts evicting profiles, new profiles are being fetched. While examining the logs did not back this theory, the
might be an indication of saturated internal state.

9.4 Scenario 2

In the second test case, both the number of events and the audience size have been increased. The pipeline configuration was left unaltered, that is, the same machine type was used and the number of workers is constrained to 1.

<table>
<thead>
<tr>
<th>Test Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kuick Configuration</strong></td>
</tr>
<tr>
<td>1. <strong>Events</strong>: 1 000 000</td>
</tr>
<tr>
<td>2. <strong>Rate</strong>: 2000 events/second</td>
</tr>
<tr>
<td>3. <strong>Tenants</strong>: 500</td>
</tr>
<tr>
<td>4. <strong>Audience</strong>: 1 000 000 consumers/tenant</td>
</tr>
<tr>
<td><strong>Pipeline Configuration</strong></td>
</tr>
<tr>
<td>1. <strong>Machine type</strong>: n1-standard-1</td>
</tr>
<tr>
<td>2. <strong>MaxWorkers</strong>: 1</td>
</tr>
</tbody>
</table>

The resources are now heavily under-provisioned and translate to a substantial increase in processing duration. The execution of the pipelines was manually terminated after three hours.

The test results when employing global windows is presented on Figure 9.8. Draw specific attention to the graph that visualizes the number of elements in the PCollections of the pipeline. This graph shows a rather interesting pattern, i.e., the number increases slightly, stagnates for an extended period of time, and increases again. There is a clear correlation between the stagnation in the graph representing element count and the graph that represents garbage collection time. Specifically, when the number of elements in the PCollections stagnates, the garbage collector kicks in. After the garbage collection process terminates, memory utilization drops and new elements are accepted. The results for internal, external storage and cloud function approaches are very similar.

It is noteworthy that for the Cloud Functions approach (Figure 9.11) the function invocations have been disabled. This was done to the end of reducing costs and is justified by the fact that the invocation of the function itself does not influence the behavior of the pipeline. For this reason, there is no data for the Datastore requests available in the
Figure 9.8: Large test global windows.
CHAPTER 9. EXPERIMENTAL RESULTS

Figure 9.9: Large test internal storage.
Figure 9.10: Large test external storage.
Figure 9.11: Large test cloud function storage.
CHAPTER 9. EXPERIMENTAL RESULTS

Figure 9.12: Large test hybrid storage.
Figure 9.13: Large test hybrid storage with cache invalidation.
Results of the implementation of hybrid state, without invalidation, are visualized by Figure 9.12. The results clearly show that the internal state was used up. This can be observed by the fact that the number of unacknowledged messages remains perfectly stable for about two hours. Even after the garbage collection process takes place, no advancement is made in the backlog.

With invalidation in the picture, the pipeline is able to make progress at a relatively slow pace. Figure 9.13 shows a correlation between the reads and writes to Datastore. More concretely, after profiles are invalidated, and hence wiped out of internal storage, new profiles are fetched from Datastore.

### 9.5 Scenario 3

In this test, Kuick was configured analogous to the test scenario introduced in Section 9.4, with a slightly increase in the rate at which events are produced. Instead of using the n1-standard-4 machine, n1-standard-4 worker machines are now used to executed the pipeline. These machines have four vCPUs and 15 GB of memory.

<table>
<thead>
<tr>
<th>Test Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kuick Configuration</strong></td>
</tr>
<tr>
<td>1. <strong>Events</strong>: 1,000,000</td>
</tr>
<tr>
<td>2. <strong>Rate</strong>: 8,000 events/second</td>
</tr>
<tr>
<td>3. <strong>Tenants</strong>: 500</td>
</tr>
<tr>
<td>4. <strong>Audience</strong>: 1,000,000 consumers/tenant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Pipeline Configuration</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>Machine type</strong>: n1-standard-4</td>
</tr>
<tr>
<td>2. <strong>MaxWorkers</strong>: 1</td>
</tr>
</tbody>
</table>

The pipelines now process the events again in a reasonable amount of time.

The graph visualizing the number of unacknowledged messages shows clear plateaus for the pipeline that uses global windows (Figure 9.14). Recall that, by definition, this pipeline accumulates state for each key in the dataset. As such, these plateaus might call our shortage in memory to account for all these keys. Though, during pipeline
CHAPTER 9. EXPERIMENTAL RESULTS

Figure 9.14: Load test results global windows.

Figure 9.15: Load test results internal storage.
CHAPTER 9. EXPERIMENTAL RESULTS

Figure 9.16: Load test results external storage.

Figure 9.17: Load test results cloud function.
Figure 9.18: Load test results hybrid approach.

Figure 9.19: Load test results hybrid approach with cache invalidation.
CHAPTER 9. EXPERIMENTAL RESULTS

execution, Kuick is still generating events. An alternative explanation hence is that Pub/Sub had delay in propagating the messages. The pipeline processed all the events in approximately 15 minutes. Internal state exhibits similar characteristics.

The backlog for the external state (Figure 9.16) is considerably smaller, with around 250,000 at peak time. Recall that the external state employs session windowing. Once again, one can observe a tail in the number of unacknowledged messages. A possible explanation of this tail is the bulk of unclosed session windows in-memory.

When relying on a function to merge the delta profile with the profile in datastore, the system lag is considerably smaller. Tests results are visualized by Figure 9.17. The spike in the unacknowledged messages gives a distorted view, as Pub/Sub is still generating events at this time. Recall that the steps performed by this pipeline are exactly equal to the steps performed by the external storage. The only difference is that external storage merges the delta profile as part of the pipeline. This operation is by definition an expensive operation, as it requires four round-trips to Datastore. Specifically, the transform creates a transaction and fetches the Datastore profile. Subsequently the transform merges the profiles in-memory, writes the result to Datastore, and finally commits the transaction. In worst case, this has to happen for each individual element that arrives in the pipeline. This gives an alternative explanation to the tail observed in the unacknowledged messages in Figure 9.16. Namely, the operation introduces too much latency to keep up with the rate of incoming events. Note that, invocation of the cloud functions have explicitly been disabled to reduce costs.

Results produced by the hybrid approach with invalidation resemble close affinity to the results observed in Figure 9.7. Once again, the number of unacknowledged messages drops at a relatively slow pace. The datastore requests are once again very symmetrical, the pipeline is able to fetch new profiles only after profiles are invalidated.

9.5.1 Autoscaling Test

So far, testing was done using a fixed set of workers. Google Dataflow provides an autoscaling feature that re-allocates the number of workers dynamically according to the runtime conditions of the pipeline (Section 7.12). The previous test cases were repeated with the autoscaling feature enabled.

<table>
<thead>
<tr>
<th>Test Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kuick Configuration</strong></td>
</tr>
<tr>
<td>1. <strong>Events</strong>: 1,000,000</td>
</tr>
<tr>
<td>2. <strong>Rate</strong>: 8000 events/second</td>
</tr>
</tbody>
</table>
3. Tenants: 500
4. Audience: 1 000 000 consumers/tenant

Pipeline Configuration
1. Machine type: n1-standard-4
2. MaxWorkers: 4

Figure 9.20: Load test results global windows with autoscaling enabled.

For the implementation using global windows, Dataflow scales up to four workers as the number of unacknowledged messages increases. The time elapsed to process 1 000 000 messages is now reduce to 10 minutes. This duration includes the time that Dataflow needed to diagnose that the pipeline was under-provisioned and to re-allocate the workers accordingly. This test also highlights the scalability of Datastore, at peak the service is processing 4 000 writes per second. For internal state, on the other hand, Dataflow scales up the workers relatively late. Most remarkable is the fact that Dataflow does not bother to scale up the pipeline that leverages cloud functions. Once again, cloud function invocations have been disabled to reduce costs.

The key takeaway from these tests is that downscaling is pretty conservative. Whereas workers are scaled up from one to four instantaneously, downscaling occurs one by one.
CHAPTER 9. EXPERIMENTAL RESULTS

Figure 9.21: Load test results internal storage with autoscaling enabled.

Figure 9.22: Load test results external storage with autoscaling enabled.
CHAPTER 9. EXPERIMENTAL RESULTS

Figure 9.23: Load test results cloud function with autoscaling enabled.

Figure 9.24: Load test results hybrid approach with autoscaling enabled.
Moreover, the scale of the test performed was too small to really appreciate the advantage of the autoscaling feature.

### 9.6 Streaming Tests

Test cases so far have been limited to pushing a batch of events in the pipeline. This is not the expected workload, as we expect records to arrive continuously. As such, additional tests were performed by ingesting events with a continuous rate.

#### 9.6.1 Scenario 1

In the first test, Kuick was configured to push a continuous stream of events with a rate of approximately 5000 events per second. For the streaming tests `n1-standard-4` worker machines were used and the maximum number of workers was constrained to 32. Tests were restricted to the implementations with external storage, cloud functions and hybrid storage with cache invalidation.

---

**Figure 9.25:** Load test results hybrid approach with cache invalidation and autoscaling enabled.
As there is inherent latency involved in scaling the pipeline, pipelines were initialized to use four workers. This is done to reduce the initial backlog accumulated and to avoid the pipeline from scaling up to the maximum number of workers instantaneously.

### Test Summary

#### Kuick Configuration

1. **Rate**: 5000 events/second  
2. **Audience**: 10 000 000 consumers

#### Pipeline Configuration

1. **Machine type**: n1-standard-4  
2. **MaxWorkers**: 32

First up is external storage, as visualized in Figure 9.26. This test highlights the latency incurred by merging the profiles as part of the pipeline. Due to the enormous backlog, Dataflow eventually scales up to 32 worker instances. The CPU utilization on these workers is rather low, as they spend most of their time communicating with Datastore.

The implementation that uses cloud functions performs surprisingly well (Figure 9.27). Both CPU and memory utilization are generally high and the system latency is 12 seconds on average. For the streaming tests, the cloud function invocations have been enabled. As a consequence, delta profiles are now being merged with the profiles in Datastore. This can be observed in the chart that visualizes Datastore requests. To the dashboard of Figure 9.27, cloud function specific charts were added. In particular, these charts visualize execution time of the function, the number of executions and the number of unacknowledged messages in the delta topic. Interestingly enough, the number of invocations is not raising above 1800 per second. This might require further investigation.

As the backlog ramps up for the hybrid approach, Dataflow scales up to 32 worker instances. As the backlog is processed, and the CPU utilization on the workers decreases, Dataflow scales down to 16 workers. The chart that visualizes Datastore requests exhibits an interesting pattern as well. Profiles are read in bulk, cached in memory and evicted as they are invalidated.

### 9.6.2 Scenario 2

Previous result for the external state pipeline was rather unsatisfactory. As Selligent leverages this implementation, another test was performed with the rate of events re-
CHAPTER 9. EXPERIMENTAL RESULTS

Figure 9.26: Streaming test results external storage.

duced to 2000 events per second. Figure 9.29 visualizes an improvement in both CPU utilization and memory usage.

Test Summary

Kuick Configuration

1. Rate: 2000 events/second
2. Audience: 10 000 000 consumers

Pipeline Configuration

1. Machine type: n1-standard-4
2. MaxWorkers: 32
Figure 9.27: Streaming test results cloud function.
CHAPTER 9. EXPERIMENTAL RESULTS

Figure 9.28: Streaming test hybrid with cache invalidation.

Figure 9.29: Streaming test results external storage.
9.7 Conclusion

The tests of the pipelines can be considered to be successful. Adding extra compute resources to the pipeline increases the events that can be processed simultaneously. We have explicitly shown that processing at the rate of 2000 events per second is doable with relatively few computing resources.

For external storage, we can conclude that the cost paid for correct, real-time results is certainly high. The pipeline relies on Datastore transactions, which is considered to be an expensive operation. As such, adding resources to the pipeline may not necessarily increase the rate at which events can be processed.

While pushing the expensive merging operation significantly increases the efficiency of the pipeline, the scalability of the Cloud Functions are pretty limited. At the time of writing, it is unknown what is causing this. If Google manages to fix this issue, the pipeline is able to scale to 5000 events per second, processed in real-time.

While the test results for the hybrid approach with cache invalidation can be considered successful, it is still unknown to what extent the pipeline will lag once the internal state becomes saturated. This requires additional experiments.
Chapter 10

End-to-end Reporting Application

This chapter proposes a highly scalable end-to-end reporting application. In particular, the different components that are part of the application are discussed.

| Disclaimer | This application serves merely as a proof-of-concept and is by no means associated with the reporting functionalities that are provided by Selligent Marketing Cloud. |

10.1 Reporting Use-cases

This chapter describes a number of reporting scenarios that will form the backbone of this work. Each use-case is centered around a specific reporting dimension and was shaped in collaboration with the engineering team at Selligent. For each scenario, we describe the relevance and propose appropriate visualizations to be able to deliver valuable insights.

10.1.1 Overview

The most prevalent data source for this work is the dataset consisting of events that record consumer interactions. Such events may be recorded as a consequence of an action taken by a consumer contextualized within a journey, i.e. opening an e-mail, clicking call-to-action links, visiting a web page, purchasing an order, and so on. While this dataset contains solely interaction data, recall that information can be extracted in regard to many different dimensions. The dimensions are described in Section 2.6.2.
These different dimensions are reflected in the events recorded in the interaction dataset. An example of such event, in JSON format for brevity, is presented in Listing 10.1. The tenant and organization refer to customer and the sub-brand of the customer respectively. The actor block captures the entity that caused the event to occur, similarly the audience dictates the audience segment to which the entity belongs. The time is represented by means of a timestamp in UTC epoch format. The journey block delivers information in regard to the journey, it identifies both the journey and the action in which the interaction was established. As described in Section 2.3.5, the action maintains a reference to the message. This message, along with the communication used, is stored in the message block.

```json
{
   "meta": {
      "tenant_id": "BC49286F-C6FF-4F69-9BDB",
      "organization_id": "bca68afb-1d7d-4659-a3e3"
   },
   "actor": {
      "type": "consumer",
      "audience_id": 4,
      "consumer_id": 2
   },
   "journey": {
      "journey_id": 842174,
      "action_id": 1000300
   },
   "message": {
      "message_id": 843270,
      "message_type": "MAIL"
   },
   "request": {
      "timestamp_utc": "20181026090138",
      "event_type": "CLICK"
   }
}
```

**Listing 10.1:** JSON serialized version of an event as encountered in the dataset.

Within the following sections, we present three valuable reporting use-cases that will form the backbone of this work. Each of these use-cases is centered around one of the aforementioned dimensions and contains a problem that can be solved by leveraging the power of streaming systems. Moreover, for each of these use-cases a series of building blocks, along with their business value, will be described.

---

1 This is a measure for describing a point in time, expressed in the number of seconds that have elapsed since 00:00:00 Thursday, 1 January 1970 UTC.
10.1.2 Consumer Profile

The consumer is the central entity in the end-to-end marketing process. The interaction data can greatly benefit the extent by which the customer experience can be personalized. The most prevalent problem in customized marketing is sending relevant content to specific consumer segments at the correct moment in time. Sending emails at an inappropriate moment in time has the effect that the email is less likely to be opened, while sending irrelevant content might have the effect that users unsubscribe from the service. Moreover, the communication channel greatly influences the reaction of the user in regard to the delivered content. The process of tracking interactions provides insights in the interests and the behavior of the customer at hand. Visualizing this data in such a way that the marketeer can infer when and how to approach a user is paramount to increase the overall consumer experience.

Within our first reporting scenario the consumer dimension is particularly highlighted. Herein, we present a series of metrics and visualizations that capture the behavior of the consumer in regard to messages sent within the context of a journey. The main building blocks for the consumer profile are as follows:

1. **Last event**: the last event or events in which the consumer in question engaged. For these events, the corresponding time, channel, message and journey are displayed. This empowers the marketeer to see their latest interaction with the consumer and act accordingly, i.e. users for which the latest interaction is older than some predefined duration may become subject to a reactivation campaign.

2. **Activity heatmap**: heatmap visualization that represent the consumers’ activity per hour of the day. This graph enables the marketeer to estimate the most appropriate time of the day to approach the consumer at hand.

3. **Consumer metrics**: from the interactions recorded for a specific consumer, a series of derived metrics are displayed. These metrics include the open rate and click-to-open rate (Section 2.6.1). Based on these metrics, the consumer may be added or removed from certain audience segments. As such, future correspondence with the user might be adapted to this end.

4. **Touchpoint graph**: graph visualizing the latest interactions with the consumer. This provides insights in the customers path towards conversion.

Figure 10.1 presents a mock-up of the dashboard composed of the various components previously described. The different components are explicitly highlighted for clarity.
10.1.3 Organization Level Insights

Organizations have a wide variety of possible communication channels at their disposal, which may be employed for various purposes. For instance, to incentivize the customers to take action with the brand, keep them up to date or to point out specific events. However, depending on the characteristics of the audience segment, certain channels may be more or less successful than others. The second reporting scenario focuses on delivering insights on the level of the organization. In this light, a dashboard containing the following components is presented to the marketeer:

1. **Channel usage**: a pie chart visualizing the stake of each distinct communication channel in the overall communication of the brand with its consumers. This graph is meant to support the marketeer in choosing the best suited channel to approach the brands’ customers.

2. **Usage over time**: a graph visualizing channel usage over time. While the visualization described above is able to provide the marketeer with a sense of which channel is most commonly used, it lacks the ability to identify trends. Visualizing usage over time on the other hand, may aid the identification of potentially interesting channels. For example, e-mailing is often historically considered to be the most dominant communication channel, however, new channels such as push notifications are becoming increasingly popular among the younger generations.

3. **Incoming events**: a bar-chart visualizing the type of events that are received via the communication channels over time. Possible event types are clicks, views, deliveries and bounces.
4. **Organization metrics**: further metrics for the organization are tracked. These metrics include the total number of messages sent, the number of consumers reached, and so on.

Figure [10.2] presents these different building blocks graphically.

**Figure 10.2**: *Organization level insights dashboard mockup.*

### 10.1.4 Journey

Journeys function, as mentioned earlier, as a roadmap for moving consumers through interactions with the brand. Journeys consist of a sequence of interactions designed to guide the consumer towards conversion. Our third, and last, reporting scenario is centered around mapping out the success of the journey. The dashboard for this scenario is made up of the following components:

1. **Component heatmap**: graph visualizing the number of consumers reached by each action in the journey. This graph is meant to support the marketeer in the
process of identifying bottleneck actions, which in turn is key to the refinement of the journey.

2. **Component metrics**: for each action in the journey, a number of more tangible metrics are maintained. Including the number of consumer reached, the bounce rate, open rate, click through rate, and so on. These metrics may be employed by the marketeer in further analysis, thereby allowing more concrete assertions to be made about the journey.

The journey dashboard is presented in Figure 10.3. Its ultimate goal is to map out the activity within the individual various actions of the journey. To this end, the different actions are each visualized by their own block. The block visualizes the type of the component, the name, the total number of consumers reached and the trend of the previous hour. Moreover, the background of this block is used to visualize an overall trend. Upon selecting a specific action, fine-grained metrics are displayed to the user.

![Journey dashboard mockup](image)

**Figure 10.3**: Journey dashboard mockup.

### 10.2 Introduction

Recall that the interaction data captured by the Selligent Marketing Cloud serves a dual purpose. First, the data is used to equip the marketeer with predefined reporting and
analysis. These functionalities support the marketeer in making predictions, which in turn can be used to guide future decisions. The value created by reporting and analysis is driven by the marketeer. Secondly, the interaction data is leveraged to design new internal features. Internal features are devoted to support the marketeer in answering the questions of who, what, where and when more precisely and accurately (Section 2.2).

While Selligent has been able to successfully provide insightful reporting procedures, the AS-IS reporting functionalities fall short in a number of ways. Section 2.6.4 provides an in-depth discussion on the shortcomings in the reporting functionalities. The main challenge is that the existing reporting functionalities are journey centric. However, in reality, there exist numerous reporting dimensions (Section 2.6.2) that carry business value. In the existing architecture, insights in other dimensions range from impossible to tedious to setup. In this chapter, we propose a new reporting application. The reporting application is designed to meet the evaluation criteria enumerated in Section 2.6.5.

10.3 High-level Overview

Figure 10.4 presents a high-level visualization of the components that make up the application. Events are ingested in Google Dataflow using Cloud Pub/Sub (Section 4.5.1). The stream processor pre-computes the different views on the data in a streaming fashion and materializes the results in different data stores.

A progressive front-end application was designed in order to allow flexible data visualization. The application communicates with a central API, which in turn consumes the data from the various data stores. The following sections zoom in on each of the individual components, and elaborates on their relevance.

10.4 Ingestion

In our experimental setup, events were generated using Kuick. In a production environment, however, events would instead be produced by the OptiExt module (Section 2.6.3). Cloud Pub/Sub (Section 4.5.1) is leveraged to ingest the events in the stream processing engine. Pub/Sub not only decouples the system, but also allows for the systems to be scaled independently. If the pipeline is ever to be shut down, Pub/Sub can buffer incoming events and deliver them when the pipeline is back up and running. The highly scalable design Google Pub/Sub is a perfect fit for the spiky nature of the interaction data. Moreover, Pub/Sub comes with a pay for what you use model where no costs are incurred. It is fully managed, meaning that there is no work required for the setup except for creating the topic to produce the interaction data.
Figure 10.4: High-level overview of the reporting application.
10.5 Processing

To compute the different views on the data, a number of streaming pipelines were implemented. These pipelines were implemented using the Apache Beam programming model (Chapter 6). While the experiments performed in this work focus exclusively on the Google Dataflow Execution engine, it is noteworthy that these pipelines can be run on different runners as well. This is particularly interesting as it does not incur a vendor lock-in. Chapter 8 provides an in-depth discussion on how the consumer profile are computed, whereas Chapter 9 discusses the results of intense load tests.

10.6 Storage

To store data, three distinct data storage technologies were used. Each of these technologies was used for their strengths.

BigQuery is used as a data warehouse for raw events. For flexibility, the events are inserted in BigQuery through a simple Dataflow pipeline. While it is out of the scope of this thesis, using a pipeline for this purpose offers the flexibility to enrich the data in real-time. Possible use-cases are enriching the events with product or user-specific information. BigQuery also offers the ability to be able to replay the event stream, which is a highly desirable property, e.g., to correct results or new analysis methods. Moreover, BigQuery equips the data scientists with a tool to perform interactive analysis. This is done through an SQL-like query language. The main drawback is that BigQuery is too slow to use to retrieve the data visualized in the dashboards. The latency is inherent with the distributed query execution model that the solution employs.

The primarily data storage technology used in the reporting application is Datastore. The choice for Datastore is motivated by cost effective model and the fact that it is a fully managed service. This fully managed nature of the data solution allows for rapid development and iteration. When using the document-like data model provided by Datastore, the developer does not have to think about the underlaying storage. An additional benefit is the out-of-the-box availability of multi-tenancy. For the touchpoint graph, the entity group feature was leveraged to create a parent-child hierarchy between events and the respective consumer. A caveat in using Datastore is that it does not allow partial updates, thereby constraining the number of use-cases that can be addressed.

To enable more complex reporting use-cases, we have experimented with Bigtable. The main drawback is that, in contrast to Datastore, data in Bigtable needs to be modeled to support the access patterns (Section 4.1). The fact that Bigtable allows partial updates makes it particularly interesting. Modeling time line data is especially tedious in Datastore, due to its document like data-model. In order to present time line graphs, as shown
in Figure [10.6] we have exploited the fact that keys are maintained in lexicographical order. This idea was already introduced in Example [14]. Additionally, the wide column nature of Bigtable was leveraged to be able to count unique events. Specifically, the characteristic was used to implement an adjacency list like data structure. Example [13] provides a concrete example of this approach.

10.7 Back-end API

To decouple the front-end from the databases, a central api was developed. The API is developed according to the REST principles. The Spring Framework[2] was used to implement the front-end, particularly because it offers a lot of features that streamline the process of building a REST API. The choice for Spring was particularly due to prior experience and the fact that it is also Java based. The fact that the back-end follows the REST principles, allows the back-end to be scaled out as well.

10.8 Front-end Application

Precomputing the data makes no sense if the data cannot be visualized as desired. The following paragraphs describe the technology stack used for the front-end application.

Vue.js[3] a progressive, JavaScript based, framework for building user interfaces. It framework offers a HTML-like templating syntax that allows to bind the Document Object Model (DOM) to the underlying Vue data. The front-end application communicates with the back-end API to show the data of interest.

The front-end application used the ElementUI[4] component framework for general components, such as cards and simple select dropdowns. For additional styling, the utility-free CSS framework, Tailwind[5] was leveraged.

To display charts, yet another framework was used. ApexCharts[6] is a charting library that aids the creation of interactive visualizations. It integrates into the Vue.js ecosystem with a simple a wrapper component.

10.9 Conclusion

In this chapter, we discussed the implementation of an end-to-end reporting application. Thanks to the streaming pipelines, the application is able to deliver near real-time insights in different dimensions of the input data. The insights are presented in the form of interactive dashboards, which can support the marketer in the creation and the refinement of their marketing campaigns. While we have not explicitly elaborated on each dimension, we state that grouping data based on another property usually suffices to be able to address another reporting dimension.
CHAPTER 10. END-TO-END REPORTING APPLICATION

Figure 10.5: Implementation of the consumer dashboard.
Figure 10.6: Implementation of the organization dashboard.
Chapter 11

Conclusion and Future Work

This work has shown that the latest advancements in the field of data processing can be leveraged in marketing automation. In particular, a scalable end-to-end reporting application was designed that is able to provide omnichannel insights. Moreover, the proof-of-concept application has explicitly shown the suitability of streaming pipelines to compute correct real-time views on unbounded input data originating from distributed input sources. This ability allows for multiple reporting dimensions to be addressed. These insights allow for increased business value, explicitly driven by the marketeer. Simultaneously, the work has focussed on implementing an internal feature, namely send-time optimization. This feature delivers content to consumers at the exact time of the day that it is most likely to be opened, thereby assisting the marketeer in maximizing the impact of their marketing campaigns. The data pipeline constructed in this work determines the best time to approach each individual consumer in a near real-time fashion.

To assess the performance characteristics of the pipelines, a number of experimental load tests were performed. For the concrete setting of send-time optimization, it became apparent that dissecting a per-user profile over an entire, unbounded input source in a real-time manner comes at a certain cost. The correctness of this implementation relies on expensive transactions. These transactions incur a huge performance bottleneck on the overall implementation. To remedy this issue, additional experimentation was performed with Cloud Functions. While these functions can be used to push computationally expensive operations out of the pipeline. However, our experiments have proven that their scalability is rather limited. At the moment of writing, it is unknown what is driving this limitation and this will be communicated with google.

Future work focusses on two different areas. The focus of the thesis was placed on extracting information from the interaction data with respect to the dimension of the consumer. Other dimensions were only implemented to a limited extent. Additional use-
cases, that focus on other dimensions, require attention. Due to the fact that the current application was designed with the idea of extensibility in mind, implementation of these use-cases is expected to be relatively straightforward. The second area for improvement concerns send-time optimization. Send-time currently tracks user activity with respect to the hour of the day. A logical extension to the feature would be to include the day of the week. Moreover, the model that is currently in use to determine the optimal send-time is relatively simple. In the interest of producing even more valuable results, this model should be replaced by a more sophisticated variant. Moreover, the feature suffers from a cold start, i.e., for it to work correctly, a certain amount of data is required. Enriching a number of representative profiles with personal information might serve as an appropriate input source to train a neural network. This neural network could potentially serve as a means to minimize the cold start of the feature.
Bibliography


Appendix A

Nederlandse Samenvatting

Bedrijven vertonen een alsmaar toenemende interesse in het aanbieden van hun producten en diensten via het internet. In tegenstelling tot traditionele marketing methoden, zoals bijvoorbeeld het fysiek versturen van brieven, biedt online marketing de mogelijkheid om aan één-op-één communicatie te doen. Oorspronkelijk was er echter geen sprake van personalisatie. Het uitsturen van een campagne had als gevolg dat elke consument hetzelfde bericht kreeg toegestuurd. Marketing was als dusdanig uitsluitend gefocust op de concrete aanbieding, zonder enig oog voor de ontvanger. De opkomst van het internet en gerelateerde technologieën maakte het mogelijk dat men doelgroepen kon segmenteren op basis van bepaalde karakteristieken, zoals bijvoorbeeld, geografie en aankoopgedrag. Op basis van de karakteristieken van deze segmenten kon de correspondentie tot op beperkte hoogte worden gepersonaliseerd.

Vervolgens werd de ontvanger centraal geplaatst in dit process. In de plaats van te focussen op een concrete aanbieding, trachtte men relevante aanbiedingen te doen voor de specifieke ontvanger. Het voeren van zulke één-op-één communicatie bevordert de relatie met de consument, verhoogt de klanttevredenheid en genereert concurrentievoordeel. Dankzij de moderne internettechnologie is het mogelijk om een bepaalde doelgroep objectief te bereiken. Een verder voordeel aan het voeren van online campagnes is het feit dat het internet niet onderworpen is aan openingsuren, hetgeen eindeloze mogelijkheden tot inkomsten creëert.

Selligent Marketing Cloud is een business-to-business marketingoplossing met als doel bedrijven te ondersteunen bij het opzetten van online business-to-consumer marketing campagnes op wereldwijd schaal. Het platform ondersteunt 700 bedrijven, actief in een grote verscheidenheid van sectoren, gevestigd in 30 landen. In online marketing, dient de marketeer volgende fundamentele vragen te beantwoorden:

1. **Wie**, de *consument* die de bedrijf tracht te bereiken, deze consumenten zijn
doorgaans deel van een groter doelpubliek.

2. Wat, heeft betrekking op de inhoud van het bericht dat naar de consument verstuurd wordt.

3. Waar, via welk communicatie medium wordt de ontvanger benaderd? Of concreter, welke kanaal wordt gebruikt om het bericht bij de ontvanger te krijgen?

4. Wanneer, het tijdstip waarop de consument bereikt wordt.

5. Waarom, welk doel tracht het bedrijf te bereiken? Wil het bedrijf een product verkopen? Wenst het bedrijf de band met de consument te versterken?

Een concrete invulling van deze concepten, noemt men een interactie. In de moderne samenleving wordt er verwacht dat de inhoud van het bericht zowel persoonlijk als relevant is. Het bericht dient via het juiste kanaal, op een gepast tijdstip, te worden verstuurd teneinde het gewenste effect te bereiken. Bedrijven trachten miljoenen contactpersonen te bereiken; deze omvang maakt het erg moeilijk om een gepersonaliseerde ervaring te bieden voor elk van deze ontvangers. Doorgaans zijn de interesses van de ontvangers erg verschillend en kunnen deze niet allemaal via hetzelfde kanaal benaderd worden. Sommige ontvangers zijn bijvoorbeeld enkel bereikbaar via e-mail, terwijl anderen enkel antwoorden op mobiele push-berichten.

Hoofdstuk 2 van de thesis biedt een overzicht in de functionaliteiten waarmee de marketeer wordt uitgerust om bovenstaande vragen te beantwoorden. Om de efficiëntie van de uitgestuurde campagnes in kaart te brengen maakt Selligent Marketing Cloud gebruik van geavanceerde tracking mechanismen. Deze mechanismen maken het mogelijk om interacties te detecteren. Interacties vinden niet uitsluitend plaats wanneer een campagne uitgestuurd wordt (outbound marketing), maar kunnen ook plaatsvinden wanneer een gebruiker een bepaalde actie uitvoert (inbound marketing). Denk hierbij aan het bezoeken van een website, het openen van een e-mail of het afrekenen van een bestelling. Outbound marketing gebeurt op grote schaal; miljoenen berichten worden tegelijkertijd uitgestuurd. Inbound marketing, daarentegen, is een communicatiestroom die steeds is toegewijd aan één enkele klant. Alle gedetecteerde interacties worden expliciet bijgehouden, dit levert een schat aan informatie die gebruikt kan worden om de marketingstrategie van bedrijven te verfijnen.

Op basis van de interactie data biedt Selligent Marketing Cloud rapportering die de marketeer inzicht geeft inzake de resultaten van hun marketing strategie. Rapporten kunnen bijvoorbeeld gebruikt worden om te identificeren welke communicatie kanalen het best geschikt zijn voor de doelgroep. Verdere analyse kan vervolgens namens de marketeer uitgevoerd worden teneinde patronen te detecteren. Deze patronen kunnen ondersteuning bieden om snellere, betere en concretere beslissingen te maken. De rapporten die door het platform aangeleverd worden zijn gelimiteerd in die zin dat er uitgebreide inzichten geboden worden in een beperkt aantal dimensies van de interactie data, namelijk de
APPENDIX A. NEDERLANDSE SAMENVATTING

invalshoek van de campagne en van het bericht. In de realiteit kan deze data echter gebruikt worden om inzichten te bieden in verschillende andere dimensies. Denk hierbij bijvoorbeeld aan de invalshoecken van de ontvanger, een specifiek kanaal of inzichten in het niveau van de volledige organisatie.

Het veld van moderne, gedistribueerde data verwerking heeft in de laatste jaren een reeks grote vooruitgangen geboekt. De thesis bestudeert hoe big data technologieën gebruikt kunnen worden om de huidige implementatie te verbeteren zodanig dat er nieuwe, of meer uitgebreide, use-cases geadresseerd kunnen worden. Meer concreet willen we na-gaan of het mogelijk is een moderne end-to-end reporting pipeline te bouwen. Onder een moderne pipeline verstaan we een *schaalbare* pipeline, die *low latency* resultaten produceert en tegelijk *meerdere/clienten* faciliteert. Verder moet de pipeline *meerdere dimensies* van de input data, op een *interactive* manier, kunnen belichten. In tegenstelling tot de bestaande pipeline, dient deze moderne variant inherent *omnichannel* te zijn. Een verdere conditie is dat de geproduceerde resultaten *correct* zijn. Onvolledige of onjuiste resultaten kunnen tenslotte als gevolg hebben dat de marketeer het platform niet langer in vertrouwen neemt.


Dataverwerking en gerelateerde concepten vormen de focus van hoofdstuk 5. De fundamentele verschillen tussen batch processing en stream processing worden hier gekaderd en de belangrijkste concepten worden toegelicht. Vervolgens wordt er verder ingezoomd op stream processing. Stream processing is in het bijzonder erg geschikt voor het verwerven van oneindige input collecties, ook wel *event streams* genoemd. Een *event* is een op zichzelf staande eenheid dat meestal is uitgerust met een timestamp. Deze timestamp represents de tijdstip waarop het event heeft plaats gevonden. Event streams en traditionele tabellen, die men in het relationele model tegenkomt, zijn erg nauw met elkaar

\[1\] De interacties, die op de verschillende communicatie kanalen waargenomen worden, moeten op een correcte manier geïntegreerd worden.

\[2\] Deze conditie lijkt op het eerste zicht vanzelfsprekend. Helaas is dit in de context van gedistribueerde systemen een erg niet triviaal aangelegenheid, zoals in de thesis bestudeerd wordt.

190
verbonden. Men kan een event stream bekomen door de individuele wijzigingen die aan een tabel aangebracht worden te observeren. Anderzijds kan een tabel beschouwd worden als een momentopname, van de laatste versie, van de waarden in de stream.

Bij het verwerken van data streams is het van uiterst belang weet te hebben van de verschillende tijdsdomeinen die hiermee gepaard gaan. Deze belangrijkste domeinen zijn event time en processing time. Event time is de tijd waarop het event effectief plaats heeft gevonden, terwijl processing time de tijd is waarop het event verwerkt wordt. In het ideale geval zijn event time en processing time gelijk, en worden events ogenblikkelijk verwerkt wanneer deze plaats vinden. In de realiteit is er echter geen duidelijke correlatie tussen de tijdsdomeinen door de karakteristieken van de onderliggende systemen. Bijvoorbeeld, events kunnen vertraagd worden door overbelasting in het netwerk, of het falen van individuen componenten. Op deze manier kunnen events mogelijk uren, of zelfs dagen, vertraagd worden. Deze notie noemt men de time-domain skew, en heeft als effect dat men niet uitsluitend de processing time van het event mag beschouwen tijdens het verwerken. Het hoofdstuk sluit af met de notie van windowing, en techniek die gebruikt wordt om een oneindige stream op te delen in eindige stukken, bijvoorbeeld per uur of per dag. Het splitsen van deze stream kan zowel op basis van de processing time als event time. Voor vele use-cases is event-time windowing echter een vereiste teneinde correcte resultaten te bekomen, omwille van de eerder geïntroduceerde time-domain skew. Denk hierbij bijvoorbeeld aan een scenario waarbij men het aantal events die op een bepaald uur gebeuren, wil achterhalen. Als men in dit geval de windows in processing time definiert kan dit tot foute resultaten leiden wanneer events vertraagd arriveren.

Hoofdstuk 6 gaat verder in op de specificaties van Apache Beam, een programmeer model om gedistribueerde pipelines te programmeren. Pipelines worden in dit model door cyclische grafen gerepresenteerd, waarbij de bogen in de graaf data collecties voorstellen. Anderzijds stellen de knopen in de graaf transformaties op deze collecties voor. Het model staat toe om expliciete overwegingen te maken inzake de dimensies van latency, completeness en cost van de resultaten. Deze overwegingen gemanipuleerd door het gebruik van windows en triggers. Vervolgens wordt de notie van watermarks onder de loep genomen. Pipelines verwerken events in processing time, echter wenst men vaak resultaten te berekenen in het event time domain. Dit specifiek omwille van het eerder aangehaald voorbeeld. Watermarks bieden de mogelijkheid om voortgang in het event-time domain te detecteren en liggen zodoende aan de basis van event time windowing. Om de pipeline uit te voeren, dient men zich te beroepen op een van de ondersteunde execution engines. Dit is tevens ook het grote voordeel van het gebruik van Beam, men kan de beschrijving van de pipeline op elk van deze engines uitvoeren. Met dit gegeven wordt het alom bekende fenomeen van de vendor lock-in voorkomen.

Vervolgens wordt er onderzocht hoe pipelines worden uitgevoerd, specifiek in de context van Google Cloud Dataflow. De concepten bestudeerd in dit hoofdstuk zijn uiterst belangrijk om de correctheid van de resultaten, die door de pipeline geproduceerd worden,
APPENDIX A. NEDERLANDSE SAMENVATTING

in te schatten. In het bijzonder wordt bestudeerd hoe exactly-once garanties geleverd worden. Deze notie is de combinatie van at-least-once en at-most-once processing. Kortweg wordt hiermee bedoeld dat er gegarandeerd wordt dat elk event slechts één keer in de output wordt opgenomen. In afwezigheid van deze garanties, is het mogelijk dat een event meerdere keren in de output weerspiegeld wordt. Voor de meeste use-cases, zoals bijvoorbeeld bij het berekenen van een som, is dit erg onwenselijk. Het waarborgen van exactly-once is niet triviaal in de context van oneindige input data die op een gedistribueerde manier verwerkt wordt. Het onbetrouwbaar karakter van het internet is slechts een van de vele factoren die deze garanties in het gedrang brengen. Correct implementeren van deze garanties brengt een bepaalde kost met zich mee. Om deze kost te drukken worden optimalisaties besproken die ervoor zorgen dat pipelines op een performante manier uitgevoerd kunnen worden. Deze optimalisaties gebeuren automatisch, echter zijn deze inzichten vereist teneinde correcte resultaten te verzekeren.

Tegelijkertijd kan de interactie data ook worden benut om interne functionaliteiten te bieden die de marketeer ondersteunen bij het beantwoorden van de fundamentele vragen. In deze thesis wordt er voornamelijk ingezoomd op de interactie data die gebruikt kan worden om antwoord te geven op de wanneer vraag. Het moment waarop een doelpubliek bereikt wordt heeft een grote invloed op de efficiëntie van een campagne. Voornamelijk de eerste uur is cruciaal bij het lanceren van bijvoorbeeld een e-mail campagne. Na afloop van dit uur verdwijnen de e-mails vaak in de overvolle mailbox. Het beste moment voor verzending is als gevolg dus ook wanneer de mailbox geraadpleegd wordt. Gebreek makend van de interactie data, kan er voor elke individuele consument bepaald worden wat het meest optimale tijdstip is om hem of haar te benaderen. Hoofdstuk 8 introduceert send-time optimization, een functionaliteit die het mogelijk maakt om een bericht te versturen naar een doelpubliek bestaande uit miljoenen gebruikers, waarbij iedere consument het bericht ontvangt op zijn meest optimale tijdstip van de dag. Het hoofdstuk bevat een diepgaande discussie van verschillende mogelijke implementaties van een dergelijke pipeline, waarbij het bestuderen van de correctheid centraal staat. Dus focus van de thesis ligt niet op de complexiteit van het algoritme, maar op de implementatie van de pipeline. In de thesis wordt als gevolg gebruik gemaakt van een relatief eenvoudige algoritme. De bedoeling is dat dit algoritme later vervangen zal worden door een complexer model.

Om inzichten inzake de performantie te verkrijgen, werden de pipelines onderworpen aan load tests. Deze testen werden uitgevoerd op het Google Cloud Platform; er werden negen run-time metrieken van deze pipelines in kaart gebracht. De load tests zijn ruwweg op te splitsen in twee verschillende categorieën, throughput testing enerzijds en streaming tests anderzijds. Bij de throughput tests werd nagegaan hoe de verschillende implementaties overweg kunnen met een batch van input data. Er werd zowel geëxperimenteerd met batches van verschillende grootte als met verschillende opstellingen inzake hardware. Het cloud-based karakter van Google Dataflow maakt het mogelijke dat het

\footnote{In dit geval staat de email alsook bovenaan in de mailbox.}
APPENDIX A. NEDERLANDSE SAMENVATTING

Aantal worker machines dynamisch aangepast kan worden naargelang de runtime conditions. Dataflow biedt deze functionaliteit aan door middel van een feature genaamd autoscaling. Het is in het bijzonder met deze feature dat Dataflow zich onderscheidt van de andere execution engines. Merk op dat interactie data onderhevig is aan grote pieken in de invoersnelheid, denk hierbij bijvoorbeeld aan Black Friday. Autoscaling zorgt ervoor dat deze pieken nodeloos opgevangen worden door het tijdelijk reserveren van extra worker machines. Na afloop van de piek kunnen de machines vervolgens terug vrij gegeven worden. Bij oplossingen die niet in de cloud draaien, dient men de cluster expliciet te voorzien op deze pieken. De effectiviteit van de autoscaling feature werd onderzocht op basis van de load tests. Deze tests waren erg succesvol, er kwam echter wel naar voor dat het terugschalen van de pipelines bijzonder conservatief is. Bijvoorbeeld, de pipeline schaalt direct omhoog van één worker naar vier workers, maar terug schalen gebeurt één per één.

Op basis van throughput testing kunnen slechts beperkte conclusies gemaakt worden; in de realiteit arriveert de interactie data continue. De realiteit werd nagebootst door testen waarbij events met een continue rate gegenereerd werden. Deze testen kunnen over het algemeen ook als succesvol beschouwd worden, met een beperkte hoeveelheid resources kunnen implementaties tot 2000 events per seconde real-time verwerken. Echter kwamen we tot de conclusie dat het inzetten van meer resources zich niet noodzakelijk vertaalt naar een verhoogde input rate. De thesis biedt discussie inzake dit gegeven. De implementatie die berust op een andere service van Google, namelijk Cloud Functions, kan tot 5000 events per seconde aan. Echter is hier het probleem dat de Cloud Functions niet goed genoeg mee schalen. De pipeline kan de events in real-time verwerken, maar de bijhorende functie loopt sterk achter. De reden hiervoor is op het moment van schrijven onbekend en hiertoe zal communicatie met Google plaatsvinden.

Hoofdstuk 10 vat aan met relevante reporting use-cases die in samenwerking met het software engineering team van Selligent werden uitgewerkt. Hierop volgt een bespreking van de end-to-end reporting applicatie. De applicatie levert de marketeer near real-time interactieve rapportering in de verschillende dimensies van de interactie data. De send-time optimization pipeline is een van de vele onderdelen van deze applicatie. Deze pipeline belicht in het bijzonder de dimensie van de specifieke ontvanger. Omwille van het interessante cost-model werd er gekozen om gebruik te maken van Datastore. Een verder voordeel is het feit dat deze service fully managed is; de ontwikkelaar kan de service out-of-the-box gebruiken en hoeft slechts beperkt na te denken over het onderliggende data model. Dit geeft echter een aantal limitaties met betrekking tot de use-cases die geadresseerd kunnen worden. Teneinde meer geavanceerde use-cases te behandelen, kan Bigtable worden gebruikt. Het gebruik van Bigtable levert een verhoogde flexibiliteit, maar een hogere up-front kost. Ook dient de ontwikkelaar expliciet te definiëren hoe de data gepresenteerd wordt, dit heeft een verhoogde complexiteit als gevolg.

In deze thesis werd expliciet aangetoond dat de laatste vooruitgangen in het veld van moderne, gedistribueerde data verwerking gebruikt kunnen worden in de context van
marketing automatie. Er werden schaalbare pipelines geïmplementeerd, die near real-time inzichten kunnen bieden in meerdere dimensies van de interactie data. De output van deze pipelines werd vervolgens gebruikt om interactive dashboards te presenteren. Deze dashboards kunnen de marketeer ondersteunen bij het verfijnen van hun marketing strategie. Tegelijkertijd werd dezelfde dataset benut als proof-of-concept voor een nieuwe feature, genaamd send-time optimization. Deze feature maakt het mogelijk om door middel van één enkele klik, een bericht te versturen naar miljoenen gebruikers, waarbij het bericht afgeleverd wordt op het meest optimale tijdstip voor elke individuele gebruiker.
Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling: Leveraging Big Data Technologies in Marketing Automation

Richting: master in de informatica
Jaar: 2019

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen -, aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingsen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Vijnck, Laurens

Datum: 10/06/2019