D5.4 – FIELD TRIALS AND EVALUATION V2

<table>
<thead>
<tr>
<th>Project Number</th>
<th>688191</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Acronym</td>
<td>STREAMLINE</td>
</tr>
<tr>
<td>Nature</td>
<td>D: Demonstrator</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>Public</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP5</td>
</tr>
<tr>
<td>Due Delivery Date</td>
<td>30th November 2017</td>
</tr>
<tr>
<td>Actual Delivery Date</td>
<td>30th November 2017</td>
</tr>
<tr>
<td>Lead Beneficiary</td>
<td>Rovio</td>
</tr>
</tbody>
</table>
| Authors | Luis Cortesão (ALB)  
Philippe Rigaux (IMR)  
Henri Heiskanen (Rovio)  
Juho Autio (Rovio)  
Björn Hovstadius (SICS – project coordinator) |
Executive Summary

STREAMLINE aims to improve Apache Flink framework in terms of online stream learning, data mining and fusing data at-rest and data in-motion. We utilise the framework on three major sectors: telco, games and web content.

This public report is the 4th deliverable of the STREAMLINE work package 5 (Industrial Applications and Evaluation). The focus of this work package is on the design, integration, implementation and evaluation of the real world industrial applications deployed by the three industrial partners.

This document reports the second iteration of the field trials and evaluation carried out in task T5.4 of the work plan. Purpose of this deliverable is to measure success in various data intensive businesses. STREAMLINE features are used in real-world business requirements under production loads to prove its effectiveness to replace existing solutions in place.

Three different real-world applications are described in this document: a real-time profiling and recommendation application by Altice Labs (ALB), streaming analytics pipeline use cases by Rovio, and a retail products classification and monitoring by Internet Memory Research (IMR).
# Table of Contents

1. Introduction .................................................................................................................. 10

2. Altice Labs ..................................................................................................................... 11
   2.1 Key Performance Indicators ...................................................................................... 11
       2.1.1 KPI 1: Rate of Recommendations .................................................................. 11
       2.1.2 KPI 2: Customers Rejection ........................................................................ 12
       2.1.3 KPI 3: Customers Engagement ..................................................................... 12
       2.1.4 KPI 4: Recommendation Success Rate ......................................................... 13
       2.1.5 KPI 5: Relative Share .................................................................................... 13
       2.1.6 Experimental Setup ....................................................................................... 14
   2.2 Solution Description .................................................................................................. 15
       2.2.1 Global Architecture Overview ....................................................................... 15
   2.3 Tests and Results ....................................................................................................... 16
       2.3.1 Data Quality Assessment ............................................................................... 16
       2.3.2 Data Processing .............................................................................................. 20
       2.3.3 Profiling and Data Analysis ........................................................................... 22
       2.3.4 Recommender Engine .................................................................................... 27
   2.4 STREAMLINE features .............................................................................................. 46

3. Rovio ............................................................................................................................... 49
   3.1 KPI 1: At least one new use case implemented ......................................................... 52
       3.1.1 Baseline and Target measures ....................................................................... 52
       3.1.2 Results ........................................................................................................... 52
   3.2 KPI 2: Business KPI's and customer profiles have a maximum latency of one hour ........................................................................ 52
       3.2.1 Baseline and Target measures ....................................................................... 52
       3.2.2 Results ........................................................................................................... 52
   3.3 KPI 3: Service Uptime Percentage ............................................................................ 52
       3.3.1 Baseline and Target measures ....................................................................... 53
       3.3.2 Results ........................................................................................................... 53
   3.4 KPI 4: Ease of development ..................................................................................... 54
       3.4.1 Baseline and Target measures ....................................................................... 54
3.4.2 Results ..................................................................................................................... 54
3.5 Solution Description ................................................................................................... 55
  3.5.1 Building FlinkJobs projects .................................................................................. 56
  3.5.2 Deploying Flink jobs using command line ............................................................. 56
  3.5.3 Azkaban Workflow Manager ................................................................................. 56
  3.5.4 Flink Streaming Job Plugin ................................................................................... 58
  3.5.5 Automatic recovery from Savepoints and Checkpoints ......................................... 59
  3.5.6 Singular Integration ............................................................................................... 60
  3.5.7 Configurable Streaming Aggregation Job .............................................................. 63
  3.5.8 Offset Monitoring .................................................................................................. 69
  3.5.9 Nagios Monitoring ................................................................................................ 71
  3.5.10 Flink logs in Kibana ............................................................................................. 71
3.6 STREAMLINE features .............................................................................................. 72
4 Internet Memory Research ............................................................................................ 75
  4.1 KPI 1: Model’s prediction precision .......................................................................... 76
    4.1.1 Current system ...................................................................................................... 76
    4.1.2 Baseline and Target measures ............................................................................. 76
  4.2 KPI 2: maintenance of a high precision .................................................................... 76
    4.2.1 Current system ...................................................................................................... 76
    4.2.2 Baseline and Target measures ............................................................................. 76
    4.2.3 Status in M24 ....................................................................................................... 77
  4.3 KPI 3: Start-over Training Necessity ......................................................................... 77
    4.3.1 Current system ...................................................................................................... 77
    4.3.2 Baseline and Target measures ............................................................................. 77
    4.3.3 Status in M24 ....................................................................................................... 77
  4.4 KPI 4: Training phase time ....................................................................................... 77
    4.4.1 Current system ...................................................................................................... 77
    4.4.2 Baseline and Target measures ............................................................................. 77
    4.4.3 Status in M24 ....................................................................................................... 78
  4.5 KPI 5: Document in training latency ......................................................................... 78
    4.5.1 Current system ...................................................................................................... 78
    4.5.2 Baseline and Target measures ............................................................................. 78
4.5.3 Status in M24 .................................................................................................................. 78
4.6 Solution description ......................................................................................................... 78
4.7 STREAMLINE features ....................................................................................................
5 Gap Analysis ...................................................................................................................... 81
6 Conclusion ......................................................................................................................... 83
List of Figures

Figure 2.1: ALB Global architecture................................................................. 15
Figure 2.2: Log records first airing distribution .................................................. 18
Figure 2.3: Initial timestamp anomaly surge....................................................... 19
Figure 2.4: Records event type categorization.................................................... 19
Figure 2.5: Events 100 delay resubmission ......................................................... 20
Figure 2.6: Repeated events per month............................................................. 20
Figure 2.7: Overview of Flinks’ Steps............................................................... 21
Figure 2.8: Database Model overview............................................................... 21
Figure 2.9: FlinkStorer ..................................................................................... 22
Figure 2.10: FlinkActivityStorer ....................................................................... 22
Figure 2.11 - Data visualization example using Zeppelin .................................... 24
Figure 2.12: Performance of the model (repeated items and new items) ............... 40
Figure 2.13: Scores on different airings (recommending repeated items) .......... 42
Figure 2.14: Average scores on different channels ............................................. 44
Figure 2.15: Number of users with a given average score .................................. 46
Figure 3.1 Service availability from Nagios monitoring system ............................ 53
Figure 3.2: Rovio high level analytics pipeline architecture ............................... 55
Figure 3.3: Flink jobs in Azkaban scheduler ..................................................... 57
Figure 3.4: Singular integration high level architecture ....................................... 60
Figure 3.5: Singular streaming job topology ..................................................... 61
Figure 3.6: Example DAG of custom aggregation job ........................................ 67
Figure 3.7: Example Grafana dashboard ......................................................... 68
Figure 3.8: Grafana dashboard builder example ............................................... 69
Figure 3.9: Kafka Offset Monitor ..................................................................... 70
Figure 3.10: Monitoring lag with Kafka Offset Monitor ..................................... 70
Figure 3.11: Nagios monitoring dashboard with “Flink Streams” monitor ........... 71
Figure 3.12: Flink logs in Kibana .................................................................... 72
Figure 4.1: the new architecture based on Flink ................................................ 72
Figure 4.2: Workflow implementation (1)......................................................... Error! Bookmark not defined.
Figure 4.3: Workflow implementation (2)......................................................... Error! Bookmark not defined.
List of Tables

Table 2.1 – Live events overview ........................................................................................................... 16
Table 2.2 – Field description for EPG - Programs ............................................................................. 17
Table 2.3 - Aggregation storage fields.............................................................................................. 23
Table 2.4 - Show Aggregation example ............................................................................................. 23
Table 2.5 – Ten Main Factors Factorization ...................................................................................... 27
Table 2.6: STREAMLINE features used in ALB use cases ................................................................. 47
Table 2.7: STREAMLINE feature test plans ...................................................................................... 48
Table 2.8: STREAMLINE features tests ............................................................................................. 48
Table 3.1: Flink job configuration parameters ................................................................................... 58
Table 3.2: Singular job configuration parameters .............................................................................. 62
Table 3.3: STREAMLINE features used in Rovio use cases ............................................................... 73
Table 3.4: STREAMLINE features planned to be used in future Rovio use cases .............................. 74
List of Listings

Listing 3.1: Building STREAMLINE projects with maven .............................................................. 56
Listing 3.2: Deploying FlinkJobs using command line ................................................................... 56
Listing 3.3: Flink batch job configuration example .......................................................................... 57
Listing 3.4: Example of streaming job configuration ....................................................................... 59
Listing 3.5: Checkpoint configuration parameters ........................................................................... 59
Listing 3.6: Singular job configuration example ............................................................................... 63
Listing 3.7: Custom aggregation job example configuration ............................................................ 67
## List of Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALS</td>
<td>Alternating Least Squares</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>COPPA</td>
<td>Children's Online Privacy Protection Act</td>
</tr>
<tr>
<td>EMR</td>
<td>Elastic MapReduce</td>
</tr>
<tr>
<td>EPG</td>
<td>Electronic Program Guides</td>
</tr>
<tr>
<td>GDPR</td>
<td>General Data Protection Regulation</td>
</tr>
<tr>
<td>GUID</td>
<td>Global Unique Identifier</td>
</tr>
<tr>
<td>HD</td>
<td>High Definition</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>ID</td>
<td>Identifier</td>
</tr>
<tr>
<td>IMDB</td>
<td>Internet Movie Database</td>
</tr>
<tr>
<td>IPTV</td>
<td>Internet Protocol Television</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>KPI</td>
<td>Key Performance Indicator</td>
</tr>
<tr>
<td>OMDB</td>
<td>Open Movie Database</td>
</tr>
<tr>
<td>QA</td>
<td>Quality Assurance</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>S3</td>
<td>Amazon Simple Storage Service</td>
</tr>
<tr>
<td>SLA</td>
<td>Service-Level Agreement</td>
</tr>
<tr>
<td>TB</td>
<td>Terabyte</td>
</tr>
<tr>
<td>UTC</td>
<td>Coordinated Universal Time</td>
</tr>
<tr>
<td>VoD</td>
<td>Video on Demand</td>
</tr>
</tbody>
</table>
# Introduction

STREAMLINE aims to improve Apache Flink framework in terms of online stream learning, data mining and fusing stream and non-stream data, and apply it to three major sectors: telco (ALB), games (Rovio) and web content (IMR). Use cases of each partner are described in more detail in document “D5.3 - Design and Implementation v2”. This document describes the results of “Pilot” stage of STREAMLINE development cycle. Each partner provides a set of KPIs that have been used to measure the performance of STREAMLINE components in a prototype deployment. We also provide solution descriptions as well as high level gap analysis to get technical insight on each use case.
2 Altice Labs

ALB use case aims to provide targeted and contextualized recommended content to IPTV customers, by connecting very high throughputs of at-rest and in-motion data streams into STREAMLINE Flink framework, which ultimately will allow for new services, performance improvements, cost reduction and business growth. ALB use cases are summarised in the next paragraphs.

Use Case 1: Real-time Analytics and Prediction

Analytics are an essential part of IPTV business, as the most important indicators and actions are calculated and retrieved from this data. Providing real-time analytics on both TV services and applications around IPTV represent a crucial next step to improved customer experience.

Status: On-going development. Some issues regarding data quality and real-time availability to be detailed and tackled in Y3.

Use Case 2: Real-time Profiling

Profiling is important for both users and clients as well as for TV channels and programs. Users’ profiles are the mechanisms that allow a thoughtful characterization of clients, typically in an automatic manner. These profiles are important to allow detailed and targeted recommendations for customers.

Regarding programs/channels profiles, although with different goals, the mechanisms to build them are similar. From the business perspective, these profiles allow a broad set of actions ranging from target campaigns of specific products or services to real-time characterization of TV content.

Status: On-going development.

Use Case 3: Real-time Recommendation

By providing quick and short lists of targeted recommended programs, full screen lists of categorized recommendations or even related programs and channels, the overall goal is always to improve customer satisfaction and engagement by recommending, on a real-time basis, the best and most suitable content according to the users’ preferences and the at-the-time available content options.

Status: On-going development.

2.1 Key Performance Indicators

2.1.1 KPI 1: Rate of Recommendations

This KPI measures the rate of recommendations provided to customers. This is measured using the number of recommendations each customer receives under a particular scenario. The goal of this KPI is to evaluate the capability of the system to provide recommendations to customers, and it does not take into account, at this stage, for the quality of the recommendations. This KPI can be mapped into a typical evaluation metric defined as recall.
2.1.1.1 Current system
ALB current IPTV content recommendation system does not provide automatic recommendations, but rather editorial (manually) chosen ones. For instance, popular TV programs or Soccer games are typically displayed in customers set top boxes are recommendations.

2.1.1.2 Baseline and Target measures
At this stage, it is not possible to define a baseline as ALB do not have a fully automatic recommendation system. Nevertheless, we expect STREAMLINE to be able to provide a minimum of 5 to 10 personalized recommendations to each customer based on his historical and real-time activity together with the TV content availability. This is thus considered as the target measure for KPI 1.

2.1.1.3 Status in M24
No change. It will be tested and evaluated on experimental setup, during Y3.

2.1.2 KPI 2: Customers Rejection
This KPI measures the rate of rejected recommendations provided to customers. This is measured by the number of times each customer premeditatedly removes a particular recommended content or category. This KPI is calculated from the precision of the recommendation system.

2.1.2.1 Current system
As mentioned in previous KPIs, ALB current IPTV content recommendation system does not provide automatic recommendations, but rather editorial (manually) chosen ones. For instance, popular TV programs or Soccer games are typically displayed in customers’ set top boxes are recommendations.

2.1.2.2 Baseline and Target measures
Although at this stage it is not possible neither to define a baseline or target measures, as ALB do not have a fully automatic recommendation system in production, it is expected that this target measure is low and gets lower as the system evolves to pilot and production phases.

2.1.2.3 Status in M24
No change. It will be tested and evaluated on experimental setup, during Y3.

2.1.3 KPI 3: Customers Engagement
This KPI measures the engagement of recommendations on customers. This is measured based on the number of recommendations that each customer followed. Whenever a customer receives a recommendation, either because he specifically looked for by navigating through the set top box menu or because it showed in the screen, it is assumed that the recommendation has a positive impact in the customer – and thus improves engagement – if the customer selects or watch that particular recommended content.

2.1.3.1 Current system
Once again, as mentioned in previous KPIs, ALB current IPTV content recommendation system does not provide automatic recommendations, but rather editorial (manually) chosen ones.

2.1.3.2 Baseline and Target measures
At this stage it is not possible neither to define a baseline nor target measures, as ALB do not have a fully automatic recommendation system in production. Nevertheless, as opposed to defined
target measures for KPI 2 (customers rejection), this target measure is expected to increase as the recommendation system evolves to from prototype to pilot and lastly to production.

2.1.3.3 Status in M24
No change. It will be tested and evaluated on experimental setup, during Y3.

2.1.4 KPI 4: Recommendation Success Rate
This KPI is a combination of Customers Engagement, Rate of Recommendations and Customers Rejection KPIs previously described that aims to assign a success rate to the recommendations provided to the customers under context constrains such as a particular time frame or set of customers’.

2.1.4.1 Current system
ALB current IPTV content recommendation system does not provide automatic recommendations, but rather editorial (manually) chosen ones.

2.1.4.2 Baseline and Target measures
Although without a recommendation system in production or tests it is not possible to defined a baseline, it is expected that with STREAMLINE it will be possible to achieve a minimum of 50% of success rate.

2.1.4.3 Status in M24
No change. It will be tested and evaluated on experimental setup, during Y3.

2.1.5 KPI 5: Relative Share
The share, as an indicator, measures the audience of a particular TV program. It is one of the most common performance indicators for TV providers, and is extremely important to understand the popularity of TV programs and channels. This KPI measures the impact that the recommendation engine has on the program share. This KPI is tested using A/B tests, and it is measured through the ratio between the share of each program watched by customers without recommendation versus customers which program was previously recommended by STREAMLINE framework.

2.1.5.1 Current system
Once again, as mentioned in previous KPIs, ALB current IPTV content recommendation system does not provide automatic recommendations, but rather editorial (manually) chosen ones.

2.1.5.2 Baseline and Target measures
It is not possible to defined a baseline at this stage, neither a single target measure, for two main reasons: First, the impact on the share of a program/channel varies a lot, for instance, with the popularity of the program/channel itself and the time of the day it is screened. And second, this impact is also strongly correlated with the Recommendation Success Rate KPI, as a high success recommendation rate is expected to arise a higher impact on the share.

2.1.5.3 Status in M24
No change. It will be tested and evaluated on experimental setup, during Y3.
2.1.6 Experimental Setup

The experimental setup will mostly be used for the evaluation of the recommendation algorithms and approached implemented in ALB use cases. One of the most challenging steps of recommender systems are the validation and evaluation, mainly due to the difficulty of getting access to real customers. At ALB we defined an evaluation methodology for IPTV recommender systems that serves both research and business purposes. It is gradual, iterative and goes through a set of two different phases during Y3.

1. In phase 1, we set a laboratory test, internal to ALB, serving between 4 and 6 users. The test-bed is used for development, testing and demonstration.
2. For phase 2, we have a community of beta testers (40-60 users). The focus of this trial is to intensively collect feedback from the users in order to validate and evaluate the recommender systems developed.
2.2 Solution Description

2.2.1 Global Architecture Overview

The global architecture of the ALB system currently being implemented in the scope of STREAMLINE project is presented in the Figure 2.1.

![ALB Global architecture](image-url)

Figure 2.1: ALB Global architecture

Inputs and outputs are represented mostly by customers and information extracted from the web, such that the information flow occurs outside ALB premises. The top most block (“web crawling”) represents the contextualization data, which is information that is collected from the web and that can bring new value and improve the profiling and recommendation engines. ALB customers’ data is represented by a house icon, with the activity logs being generated in each set top box and sent to ALB premises for storing and analysis. Additionally, the recommendations received by each customer and the feedback they may provide is part of the inputs for this architecture, whether in this case data providing from recommendations and users’ feedback is sent/received through a REST API interface.

Moving from the data sources to the Data Center, the REST API interface handles all requests from/to ALB customers. Activity logs go through Golias data collection infrastructure, that collects, process and store that data in a HDFS cluster, and subsequently become inputs in the recommendation engine. Contextual data is also used as input for this engine.
The final step to provide real-time recommendation to customers’ is to aggregate the rankings, weights and factors obtained by the recommendation engine for each program with both historic and real-time customers’ feedback collected and processed by Apache Flink (STREAMLINE fork). This information is then processed and stored in a fast, distributed database (defined in the architecture diagram as “Clients & Programs Profiles”), which can be accessed directly from the REST API interface to provide responses to all customers’ requests.

2.3 Tests and Results

The system is being developed as described in D5.3 – Design and Implementation v2 – and its different components are being tested and evaluated. In the scope of this deliverable (Y2), and in accordance to the different processes presented in D5.3, we will present the associated tests, evaluations, conclusions and next steps for Y3. Those conclusions and next steps will be of the utmost importance to D5.5 – Design and Implementation v3 – as well as for the STREAMLINE features under development in WP1, WP2 and WP3.

2.3.1 Data Quality Assessment

The system comprehends several data streams, all with different characteristics and objectives (detailed description in D5.3).

During data processing implementation, several tests were applied to the different data streams, enabling a data quality assessment. Its results, analysis and conclusions are presented in the following sections.

2.3.1.1 Data streams

**Activity Logs:** These logs include the large majority of activities that are performed by IPTV clients and which are recorded as an activity log, and include both TV usage and viewing pattern information. Records of activity logs represent activities executed on each set top box, by one or more viewers. There are dozens of different types of events being recorded on each set top box, but for the sake of our use cases, just part of them are relevant, and those are detailed below. The targeted events regard user activity on live content, detailing Channel Tune and Program watched.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event id</td>
<td>100</td>
</tr>
<tr>
<td>Event Name</td>
<td>Channel Tune</td>
</tr>
<tr>
<td>Event Description</td>
<td>A channel tune event occurs when a subscriber tunes away from a TV channel. A subscriber must remain on the tuned channel for a specified minimum amount of time (the default is 6 seconds) before the event is logged. If a subscriber changes channels before the minimum tune time, the tune is not recorded.</td>
</tr>
</tbody>
</table>

| Event id               | 114                                                                                                                                          |
| Event Name             | Program Watched                                                                                                                                |
| Event Description      | A program watched event occurs when the program that a subscriber is watching changes to another program.                                      |

Table 2.1 – Live events overview
**EPGs:** Electronic Program Guides is another important data stream comprehending all the information about programs and channels, such as scheduled date and time, description, duration, etc. To allow a more agile share of information, the EPG divides Program and Channel to different Files. The table below details the information made available by the EPGs.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title ID</td>
<td>The Programs’ name with season information</td>
</tr>
<tr>
<td>Is Adult Content</td>
<td>Boolean to indicate if the Program has adult content</td>
</tr>
<tr>
<td>Program Id</td>
<td>The Programs’ identifier</td>
</tr>
<tr>
<td>Number of Episodes</td>
<td>The available number of episodes of the corresponding season</td>
</tr>
<tr>
<td>Call Letter</td>
<td>The Channels’ Call letter</td>
</tr>
<tr>
<td>Series Episode Number</td>
<td>The Episode number, for the season</td>
</tr>
<tr>
<td>Title</td>
<td>The Programs’ name with season and episode information</td>
</tr>
<tr>
<td>Program Duration</td>
<td>The Programs’ duration, in seconds</td>
</tr>
<tr>
<td>Participants</td>
<td>A list of known participants, such as actors</td>
</tr>
<tr>
<td>End Date</td>
<td>A datetime field with the end time for this Program airing</td>
</tr>
<tr>
<td>Start Date</td>
<td>A datetime field with the start time for this Program airing</td>
</tr>
<tr>
<td>Synopsis</td>
<td>The Synopsis for this Program</td>
</tr>
<tr>
<td>Series Id</td>
<td>The series identifier</td>
</tr>
</tbody>
</table>

Table 2.2 – Field description for EPG - Programs

### 2.3.1.2 Data Issues

An initial survey of data quality was performed on sample activity logs comprising:

- 10,000 clients (with particular focus on intensive users)
- spanning for 2 months

The most relevant findings are presented below:

**Merging series id-s**

It is reasonable to merge different series episodes under one id before making recommendations. There are 359970 unique program ids in the context file (249459 of these are present in the log file). However, if we merge shows with the same name (news, talk shows, etc) under one id, and also merge shows belonging to the same series (stripping eg. T1 Ep. 13 from the names), this number goes down to 31646 in the context file (24600 in the log file), which is one order of magnitude change.

Also, after analyzing this data (Figure 2.2), we can see that the number log records belonging to first airings of a show is not merely 4.7% of the number of all records.
Figure 2.2: Log records first airing distribution

With this ratio, we can attempt to make collaborative filtering recommendations without problems for the majority of the shows. However, the remaining 4.7% is a biased selection of the shows, and probably has a disproportionately high ratio of e.g. movies. This means that it may be worthwhile to approach this partition of the data with other tools, such as context based filtering.

**Log events starting before the program:**

The number of view events that start at least 1 minute before, but at most 1.5 hours before the program that is being viewed: 2431597, which is about 4% of the raw data (2431597 / 61085135). When counting the number of events similarly, but after some preprocessing steps (dropping faulty records, only considering records assumed to be live views, etc) we get 4.3% (2431367 / 55955253). When counting the number of events similarly, after combining the events from the same user viewing the same show into one, we get 5% (2208706 / 43783460).

After some research, it seems that actually most of the examples are probably caused by the program file misreporting the air time (or by choosing the wrong air time from multiple records reporting different times). It was reported to MEO for fixing.

**Categorizing records into every types:**

Heuristic: if the event starts after a show’s airing has started but before it has ended, it is counted as live (1). If the event happens during the week following a show’s airing, it is counted as watching automatically recorded (2). And if the event happens after that, it is counted as watching manually recorded (3). Also there is an additional type of event, where there hasn't been an airing of the show according the our data yet (0). This heuristic had to be altered however: since the devices report the timestamps themselves, there is a surge of people starting to watch the show right before the show starts (it starts ~1.5 hours before). See Figure 2.3:
The rightmost time on the x axis is the start of the show (unit: seconds). We count these records as watching live. A better solution would be to try to correct the devices’ time errors before categorizing the records. This issue was reported to MEO for fixing.

After the categorization, we can see the distribution of the number of records per event type:
Those issues were reported to MEO for fixing. A initial feedback reported the issue being related to STBs configurations as well as the type of queue implement between the MediaRoom system and the HDFS repository. MEO would analyze it further and work on a fix on a priority basis.

### 2.3.2 Data Processing

#### 2.3.2.1 Data Processing implementation

The Data Processing phase has the goal of relating the entries of the input streams and storing the result in a relational database. This phase requires a solution capable of utilizing the available resources of a distributed system, have a flexible handling of heterogeneous streams and available database clients. Apache Flink satisfies these requirements, therefore, ALBs’ processing flow has been modelled accordingly. The data flow is divided into input sources, data transformations and data sinks.

A high-level overview of Flink data processing steps are shown in Figure 2.7.
The input sources for this phase are Files, stored in a Hadoop File System, comprising the previously described data sources - Channels, Programs and User Activity. This input stream is split to multiple operators, where the Files’ records are collected and transformed.

The transformation operations are responsible for extracting relevant information from the record stream, aggregate and supply the sink functions. The extraction task validates and collects all the incoming records. This task is unique to each stream; therefore, the results are passed to further operators for joining. Record aggregation, on a stream environment, requires an appropriate window on which to apply the join operations. On this window, it is also required to hold events in state while their assigned timestamp is below a certain threshold.

The last transformation steps transform the aggregated records to fit the database model.

Finally, the inputs produce four final streams, on which the sink operations perform batch insertions on a PostgreSQL database. These insertions must be performed sequentially to allow a correct key reference. An overview of the related tables can be seen in Figure 2.8.

The entities depicted in the Database Model may be further detailed as:

- A Program refers to an aired instance of a show, e.g. The Simpsons - Season 24 - Episode 2
- An Exhibition presents the timeslots in which a Program aired on a given day, by Channel
- Entries in Visualizations target only single Exhibitions, therefore, having one associated
Channel
The associated Flink implementations are illustrated below – Flink Storer (Figure 2.9) and Flink Activity Storer (Figure 2.10).

---

2.3.2.2 Data Processing tests
Given the data quality issues previously identified, the data processing tests were mainly focused on validating the functional implementation. This validation is still on-going and will provide additional inputs on data quality, as well as some insights on the subsequent optimization and tuning of the process to be performed on Y3, which will also include the required adaptations to data quality fixing to be provided by MEO (to be reported on D5.5 and D5.6).

2.3.3 Profiling and Data Analysis

2.3.3.1 Profiling and Data Analysis implementation
ALBs’ ability to provide suitable and targeted recommendations relies on understanding the consumers’ preferences. This understanding would be provided by an analysis of STB activity, on content and time.
The activity is expected to comprise multiple Viewers, due to the STB being commonly shared among the household. Each Viewer must have an associated Profile. These Profiles could then be used by both Recommender system as well as Data analysis purposes.
The chosen implementation of identifying personas provides a Profile for each hour, which would ideally produce the following:

- hour 09:00 : Viewer 1 “John” -> mostly Action, keen on the Hollywood Channel;
- hour 10:00 : Viewer 2 “John and Jessica” -> regular series watcher;
- hour 11:00 : Viewer 3 “little John and his Merry Men” -> mostly Comedy;

This method for Profiling requires further evaluation and Recommender system requirements before optimizations and new hypotheses can be presented.

A Profile is, at the moment, described by the aggregated Viewer activity in a time slot. It targets certain aspects of this activity, of which, ALB has identified interest on Profiles that describe Channel, Show and Genre. These Profile are stored in a database, PostgreSQL, as entries in table that relate time, box and the aggregated information, as can be seen in Table 2.3.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>timestamp with time zone</td>
<td>The datetime field with hour precision</td>
</tr>
<tr>
<td>Box</td>
<td>uuid</td>
<td>The STB identifier</td>
</tr>
<tr>
<td>Doc</td>
<td>jsonb</td>
<td>The aggregated activity for the corresponding day and hour.</td>
</tr>
</tbody>
</table>

Table 2.3 - Aggregation storage fields

The aggregated activity is to be stored in PostgreSQL’ `jsonb` format. While its’ outer keys store the targeted results, their inner keys provide the visualization ratio (in seconds). An extract of the Shows’ Aggregation, with altered STB identifiers, is displayed in Table 2.4.

<table>
<thead>
<tr>
<th>Show Aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
</tr>
<tr>
<td>2017-01-30 00:00:00+00</td>
</tr>
<tr>
<td>2017-01-30 01:00:00+00</td>
</tr>
<tr>
<td>2017-01-30 01:00:00+00</td>
</tr>
<tr>
<td>2017-01-30 02:00:00+00</td>
</tr>
<tr>
<td>2017-01-30 00:00:00+00</td>
</tr>
</tbody>
</table>

Table 2.4 - Show Aggregation example
Data Analysis on the aggregated data and stored Activity is performed using Apache Zeppelin. Zeppelin connects to the database to provide an interactive visualization of the Profiling questions. An example of such can be seen in Figure 2.11.

2.3.3.2 Profiling and Data Analysis tests

In parallel, and to evaluate alternatives to the previously presented implementation of profiling, additional tests (based on a factorization output) were performed on the existing dataset. The results of the profiling are presented in Table 2.5, with a list of the ten most relevant profiles identified and associated details – main genre, main channels and sample programs watched.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Importance</th>
<th>Short description</th>
<th>Main genre</th>
<th>Main channel</th>
<th>Sample program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.61</td>
<td>Series, news</td>
<td>Series, Comedy, Drama, Mystery</td>
<td>SIC, JJAM, FOX</td>
<td>Os Mistérios de Miss Fisher</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>House</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Televendas</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Digimon Fusion</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Family Guy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yu-Gi-Oh GX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lei &amp; Ordem: Unidade</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Especial</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jess e os Rapazes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.27</td>
<td>Kids-teen, sport</td>
<td>Kids, Sport</td>
<td>BIGGS, ABOLA, TVIFIC</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.85</td>
<td>Entertainment</td>
<td>Séries, Desporto, Entretenimento</td>
<td>FOX, TVIFIC, SLB, RTPM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Soldados da Fortuna Crime, Disse Ela O Prédio do Vasco Alfred Hitchcock Apresenta C.S.I. Miami Merlin A Lei do Amor Amor E Revolução Benfica</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.64</td>
<td>Comedy, Romance</td>
<td>Series</td>
<td>FOX, DISNY</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Foi Assim Que Aconteceu Jess e os Rapazes Uma Família Muito Moderna Foi Assim Que Aconteceu House The Mindy Project Bob's Burgers Family Guy A Teoria Do Big Bang</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.75</td>
<td>News, Information</td>
<td>Information</td>
<td>SICN, TVI24, RTP3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Boa Cama, Boa Mesa Imagens De Marca Os Europeus TV Shop Espaços &amp; Casas Telejornal Madeira Viagens À Minha Terra Golf Report Contas Poupança As Horas Extraordinárias Manchetes 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.8</td>
<td>Kids: DisnyJ</td>
<td>Kids</td>
<td>DISNYJ, SICK, BIGGS</td>
<td></td>
</tr>
</tbody>
</table>
|   |   |   |   | Lanfeust De Troy  
|   |   |   |   | É Inabungacreditável! |
|   |   |   |   | Nina Já É Crescida! |
|   |   |   |   | Doutora Brinquedos:  
|   |   |   |   | Está Na Tua Mão |
|   |   |   |   | Manter-Te São  
|   |   |   |   | A Casa Do Mickey Mouse |
|   |   |   |   | As Mini Aventuras De Winnie The Pooh  
|   |   |   |   | Liga-Te A Miles |
|   |   |   |   | Minnie Toons  
|   |   |   |   | Tsum Tsum  
|   |   |   |   | A Princesa Sofia |
| 7 | 0.98 | Kids + mother (TV shop, gastro,...) | Kids, Information, Entertainment | JJAM, DISNYJ, BABYT, SICN, TVIFIC |
|   |   |   |   | O Prédio do Vasco  
|   |   |   |   | TV Shop |
|   |   |   |   | Chirp  
|   |   |   |   | Pingu |
|   |   |   |   | Kit e Kate  
|   |   |   |   | Wussywat  
|   |   |   |   | Thomas E Os Seus Amigos  
|   |   |   |   | Masterchef Brasil |
|   |   |   |   | Joe e Jack  
|   |   |   |   | Inspetor Max |
|   |   |   |   | Morangos com Açúcar |
| 8 | 0.78 | Kids JJAM, BABYT | Kids | JJAM, PANDA, BABYT |
|   |   |   |   | Wussywat  
|   |   |   |   | O Meu Universo  
|   |   |   |   | Bob O Construtor |
|   |   |   |   | Vida Selvagem  
|   |   |   |   | Pingu  
|   |   |   |   | Kit e Kate  
|   |   |   |   | Yaya E Zouk  
|   |   |   |   | Joe e Jack  
|   |   |   |   | Os Heróis da Cidade |
| 9 | 0.89 | Kids PANDA | Kids | PANDA, DISNYJ, SICK |
|   |   |   |   | Bing  
|   |   |   |   | O Rato Renato  
|   |   |   |   | Os Doozers  
|   |   |   |   | Zazie & Max  
|   |   |   |   | Um Amigo De Peso  
|   |   |   |   | Missão: Ao Resgate Da Selva  
|   |   |   |   | Wow! Wow! Wubbzy!  
|   |   |   |   | Porquinha Peppa  
|   |   |   |   | Bob O Construtor |
### 2.3.4 Recommender Engine

#### 2.3.4.1 Batch and online recommender algorithms implemented in Flink

The recommender system has to serve in an online environment, which can be highly non-stationary. Traditional recommender algorithms may periodically rebuild their models, but they cannot adjust to quick changes in trends caused by timely information. In our experiments with actual Altice Labs set-top box data, we observed that even a simple, but online trained recommender model can perform significantly better than its batch version. We design online learning based recommender algorithms that can efficiently handle the non-stationary properties of the Altice Labs task.

Detailed description of the implementation is available in D5.3 (section 2.3.6.2 - Overview of batch and online recommender algorithms implemented in Flink)

The implemented developments are based on the batch API based iALS and the streaming API based batch and online DSGD STREAMLINE pull requests:

- **IALS**: [https://github.com/gaborhermann.flink/tree/ials](https://github.com/gaborhermann.flink/tree/ials)
  [https://issues.apache.org/jira/browse/FLINK-4613](https://issues.apache.org/jira/browse/FLINK-4613)
  [https://github.com/apache/flink/pull/2542](https://github.com/apache/flink/pull/2542)
- **DSGD** [https://issues.apache.org/jira/browse/FLINK-4961](https://issues.apache.org/jira/browse/FLINK-4961)
  [https://github.com/apache/flink/pull/2819](https://github.com/apache/flink/pull/2819)

#### 2.3.4.2 Prototyping notebook experiment

Below a short description of the Prototyping notebook implementation and configuration, used to test and evaluate the recommender implementation.

**simulated batch sgd experiment**
We process the data linearly, and train the recommendation system on the data available so far every 24 hours. To simulate a real-world application, the use of models is further delayed by 24 hours, meaning we don’t start using a model until the next training period. The models recommend ranking lists for every user from the currently available programs, and these are evaluated on the viewing events one by one, with the online NDCG metric. The results are plotted averaged over 24 hours.

Default parameter values:
- top K: 10
- dimensions: 10,
- learning rate: 0.05
- regularization rate: 0
- number of iterations: 3
- period length: 86400

```
In [23]:

   experiment = pyrecsys.experiments.PortugalTelekomExperiment(
       learningRate=0.004,
       negativeRate=10,
       regularizationRate=0.001,
       topK=5,
       number_of_iterations=5,
       dimension=20,
       contextData=context_data,
   )

   rankings = experiment.run(data)

   reading data...
   reading data... finished
   logging to memory
   running experiment...

In [24]:

   results = prs.NdcgScore(rankings)

   results.timeFrame(60*60*24).plot()
```

```
Out[24]:
<matplotlib.axes._subplots.AxesSubplot at 0x7f43889e4c88>
```
Parameters of batch sgd

Parameters we test for:
- learning rate
- negative rate
- regularization rate
- number of batch iterations over a batch
- number of latent dimensions

These experiments measure the effectiveness of the parameters on a later part of the data.

In [6]:

```python
parameter_eval_data = data.copy()
parameter_eval_data['eval'] = data['eval'] & (data['time'] > (data['time'].iloc[-1] - 86400))
```

In [8]:

```python
parameter_experiment = pyrecsys.experiments.PortugalTelekomExperiment(    learningRate=0.005,    negativeRate=10,    regularizationRate=0.001,    topK=5,    contextData=context_data,    period_length=(data['time'].iloc[-1] - data['time'].iloc[1] - 2 * 86400)
)```
In [10]:

```python
lr_params = prs.ThreadedParameterSearch(parameter_experiment, prs.NdcgScore, threads=8)
lr_params.setParameterValues('learningRate', [0.002, 0.005, 0.007, 0.01, 0.015, 0.02])
lr_params_result = lr_params.run(parameter_eval_data, verbose=False)
```

In [11]:

```python
nr_params = prs.ThreadedParameterSearch(parameter_experiment, prs.NdcgScore, threads=6)
nr_params.setParameterValues('negativeRate', [0, 5, 10, 15, 20, 25])
nr_params_result = nr_params.run(parameter_eval_data, verbose=False)
```

In [75]:

```python
rr_params = prs.ThreadedParameterSearch(parameter_experiment, prs.NdcgScore, threads=5)
rr_params.setParameterValues('regularizationRate', [0.003, 0.007, 0.01, 0.013, 0.016])
rr_params_result = rr_params.run(parameter_eval_data, verbose=False)
```

In [13]:

```python
it_params = prs.ThreadedParameterSearch(parameter_experiment, prs.NdcgScore, threads=4)
it_params.setParameterValues('number_of_iterations', [1, 3, 5, 7])
it_params_result = it_params.run(parameter_eval_data, verbose=False)
```

In [15]:

```python
dim_params = prs.ThreadedParameterSearch(parameter_experiment, prs.NdcgScore, threads=5)
dim_params.setParameterValues('dimension', [5, 10, 15, 20, 25])
dim_params_result = dim_params.run(parameter_eval_data, verbose=False)
```

In [81]:

```python
plots = [
    ('learningRate', lr_params_result),
    ('negativeRate', nr_params_result),
    ('regularizationRate', rr_params_result),
    ('number_of_iterations', it_params_result),
    ('dimension', dim_params_result)
]
fig = plt.figure()
```
for i, (name, param_results) in enumerate(plots):
    ax = fig.add_subplot(2, 3, i+1, ylabel='ndcg', xlabel=name, xticks=param_results[name])
    ax.plot(*param_results.T.values)
    ax.tick_params(axis='x', labelsize=8)
plt.tight_layout()

baselines¶

For correctness sake, the training of these models is done in batches of 24 hours, and the use of the models is further delayed by 24 hours, similar to batch sgd.

popularity model¶

We recommend the most popular item available every time.

personal popularity model¶

We try to recommend from the available items the one that the user has watched the highest number of times. This model falls back to the global popularity if it's not able to make a recommendation from the available items.

In [25]:

```python
ppopularity = pyrecsys.experiments.PeriodicPopularityModelExperiment(
    seed=254938879,
    topK=5,
    filters=[prs.AvailabilityFilter(context_data)],
)```
ppop_results = ppopularity.run(data_implicit)

reading data...
data reading finished
logging to memory
running experiment...

In [26]:

ppersonalpopularity = pyrecsys.experiments.PeriodicPersonalPopularityModelExperiment(
    seed=254938879,
    topK=5,
    filters=[prs.AvailabilityFilter(context_data)],
)

ppersonalpopularity_results = ppersonalpopularity.run(data_implicit)

reading data...
data reading finished
logging to memory
running experiment...

In [27]:

pd.concat([
    prs.NdcgScore(rankings).timeFrame(60*60*24).rename(columns={'ndcg':'batch sgd'}),
    prs.NdcgScore(ppop_results).timeFrame(60*60*24).rename(columns={'ndcg':'periodic popularity'}),
    prs.NdcgScore(ppersonalpopularity_results).timeFrame(60*60*24).rename(columns={'ndcg':'periodic personal popularity'}),
], axis=1).plot()

Out[27]:
<matplotlib.axes._subplots.AxesSubplot at 0x7f4388d7e9e8>
Online models

In [6]:

```python
one线上_experiment = pyrecsys.experiments.PortugalTelekomOnlineExperiment(
    learningRate=0.14,
    negativeRate=30,
    regularizationRate=0.0,
    topK=5,
    contextData=context_data,
)
```

In [7]:

```python
online_rankings = online_experiment.run(data)
```

reading data...
data reading finished
logging to memory
running experiment...

In [44]:

```python
online_results = prs.NdcgScore(online_rankings)
online_results.timeFrame(60*60*24).plot()
```

Out[44]:
<matplotlib.axes._subplots.AxesSubplot at 0x7f420ccb0208>
online vs batch sgd

In [45]:

    pd.concat(
            
            prs.NdcgScore(rankings).timeFrame(60*60*24).rename(columns={'ndcg':'batch sgd'}),
            
            prs.NdcgScore(online_rankings).timeFrame(60*60*24).rename(columns={'ndcg':'online sgd'}),
            
        axis=1).plot()

Out[45]:
<matplotlib.axes._subplots.AxesSubplot at 0x7f40c20dcef0>
parameters of online sgd

In [35]:

```python
online_lr_params = prs.ThreadedParameterSearch(online_experiment, prs.NdcgScore,
threads=7)
online_lr_params.setParameterValues('learningRate', [0.05, 0.1, 0.15, 0.2, 0.25, 0.3])
online_lr_results = online_lr_params.run(data, verbose=False)
```

In [36]:

```python
online_nr_params = prs.ThreadedParameterSearch(online_experiment, prs.NdcgScore,
threads=6)
online_nr_params.setParameterValues('negativeRate', [10, 50, 100, 150])
online_nr_results = online_nr_params.run(data, verbose=False)
```

In [84]:

```python
online_rr_params = prs.ThreadedParameterSearch(online_experiment, prs.NdcgScore,
threads=6)
online_rr_params.setParameterValues('regularizationRate', [0.003, 0.007, 0.01, 0.013, 0.016])
online_rr_results = online_rr_params.run(data, verbose=False)
```

In [38]:

```python
online_dim_params = prs.ThreadedParameterSearch(online_experiment, prs.NdcgScore,
threads=4)
```
online_dim_params.setParameterValues('dimension', [10, 15, 20, 25])

online_dim_results = online_dim_params.run(data, verbose=False)

In [85]:

plots = [
    ('learningRate', online_lr_results),
    ('negativeRate', online_nr_results),
    ('regularizationRate', online_rr_results),
    ('dimension', online_dim_results)
]

fig = plt.figure()

for i, (name, param_results) in enumerate(plots):
    ax = fig.add_subplot(2, 2, i+1, ylabel="ndcg", xlabel=name, xticks=param_results[name])
    ax.plot(*(param_results.T.values))
plt.tight_layout()

Online baselines

In [40]:

online_popularity = pyrecsys.experiments.PopularityModelExperiment(
    seed=254938879,
    topK=5,
    filters=[prs.AvailabilityFilter(context_data)])
online_popularity_rankings = online_popularity.run(data_implicit)

reading data...
data reading finished
logging to memory
running experiment...

In [41]:

online_ppopularity = pyrecsys.experiments.PersonalPopularityModelExperiment(
    seed=254938879,
    topK=5,
    filters=[prs.AvailabilityFilter(context_data)],
)

online_ppopularity_rankings = online_ppopularity.run(data_implicit)

reading data...
data reading finished
logging to memory
running experiment..

In [46]:

pd.concat([
    prs.NdcgScore(online_rankings).timeFrame(60*60*24).rename(columns={'ndcg ':'online sgd'}),
    prs.NdcgScore(online_popularity_rankings).timeFrame(60*60*24).rename(columns={'ndcg':'online popularity'}),
    prs.NdcgScore(online_ppopularity_rankings).timeFrame(60*60*24).rename(columns={'ndcg':'online personal popularity'}),
], axis=1).plot()

Out[46]:
<matplotlib.axes._subplots.AxesSubplot at 0x7f4187dbd710>
online experiment with small learning rates

In [8]:

```python
online_lr_small_params = prs.ThreadedParameterSearch(online_experiment, prs.NdcgScore, threads=7)

online_lr_small_params.setParameterValues('learningRate', [0.001, 0.01, 0.03, 0.05, 0.07, 0.1])

online_lr_small_results = online_lr_small_params.run(data, verbose=False)
```

In [9]:

```python
online_lr_small_results.set_index('learningRate').plot()
```

Out[9]:
```<matplotlib.axes._subplots.AxesSubplot at 0x7fd913227240>```
2.3.4.3 Tests and results

We ran two different scenarios: recommending from all the available items, and recommending only from the available items that are new to the user. The performance of different models can be seen in the Figure 2.12.

![Scores with repeated items](image1)

![Scores when recommending only new items](image2)

Figure 2.12: Performance of the model (repeated items and new items)

We can see that the new item recommendation task is much harder.
Scores on different airings of the same show (recommending repeated items)
We have also examined the scores’ distribution.
The following charts (Figure 2.13: Scores on different airings (recommending repeated items)show in blue the performance of the recommender system (y axis) when measured only on shows that are played for the nth time (x axis). It also shows the number of such shows (red line).
Figure 2.13: Scores on different airings (recommending repeated items)
Average scores on different channels
The following charts (Figure 2.14) show the average scores (blue bars) when measured only on shows played on various channels (x axis). The charts also show the number of records belonging to the channels (red line).
Figure 2.14: Average scores on different channels
Number of users with a given average score

The following charts (Figure 2.15) show the number of users with a given average score (blue line). The red line shows the average number of records for the given user group.
Relevant findings
- The average activity has lowest values around 500 in a month, which still means more than 8 unique shows watched on average every day.
- Very active users have very high accuracy. Note that above NCGD 0.5, we either predict the exact channel selected in 50% of the time, or we include the selected channel in the top 3 recommendations.
- Very active users might be pubs, nursery schools or other places that run the TV all day - may be worth manually investigated and even dropped from training and evaluation.
- Accuracy above NDGC 0.2-0.4 is already expected to provide good user experience.

2.4 STREAMLINE features
In the scope of the project, and upon requirements identified by the use cases, STREAMLINE functionalities will be developed, mainly extending Flink capabilities. The specific STREAMLINE features (available in the following Flink branch - https://github.com/streamline-eu/streamline-hybrid-engine), used in Altice Labs use cases, are identified in Table 2.6.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Use case</th>
<th>Relevance</th>
<th>Availability</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault tolerance: Managing stateful streaming (T1.2)</td>
<td>Data Processing within all use cases</td>
<td>Critical</td>
<td>Available</td>
<td>The use of a Fault tolerant application is a key requirement in Data Processing. It relies on information to be related and enriched, requiring no data loss and the persistence of unprocessed events.</td>
</tr>
</tbody>
</table>
A High-Level Declarative Language for Machine Learning at Rest and in Motion (T3.2)

Data Processing within all use cases

Important

Available

The envisioned Machine Learning (ML) algorithms should make use of both the stored events as well as aggregated information (eventually costly SQL queries on the stored data). An integration of this logic onto the application would benefit the overall operational cost, improving database availability and readily supplying ML algorithms.

Conjoint Machine Learning (T2.1)

Recommendation

Critical

Available

The recommendation system should fit the model in batch and apply the model in real-time. The recommender system has to serve in an online environment, which can be highly non-stationary.

Table 2.6: STREAMLINE features used in ALB use cases

Table 2.7 will list and describe the plan, associated steps, existing constraints and target deliverable to test Streamline functionalities in each case.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Planned Tests</th>
<th>Test Details /Constraints</th>
<th>Target Deliverable</th>
</tr>
</thead>
</table>
| Fault tolerance: Managing stateful streaming (T1.2) | • validate adequacy (level of automacy, human intervention required, ...) to system failures and maintenance interventions • evaluate impacts on performance | Test Details
Test Details
• tbd
Constraints
• existing data quality issues and its impact on data processing stabilization may delay tests | D5.6                |
A High-Level Declarative Language for Machine Learning at Rest and in Motion (T3.2)

- evaluate overall operational cost benefits (comparing to SQL queries on the stored data)
- evaluate improvement on database availability

Test Details
- tbd

Constraints
- existing data quality issues and its impact on data processing stabilization may delay tests

Conjoint Machine Learning (T2.1)

- evaluate accuracy of recommender results
  - batch API based iALS
  - streaming API based batch and online DSGD
- evaluate recommender performance adequacy
- evaluate recommender in experimental setup

Test Details
- accuracy will be evaluated in several iterations, comprising different datasets (time period, clients, ...)

Constraints
- existing data quality issues and its impact on data processing stabilization may impact test results

Table 2.7: STREAMLINE feature test plans

Table 2.8 will describe the STREAMLINE features executed tests and associated results.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Tests Performed</th>
<th>Test Results</th>
<th>Comments</th>
</tr>
</thead>
</table>
| Conjoint Machine Learning (T2.1) | • evaluate accuracy of recommender results
  - batch API based iALS
  - streaming API based batch and online DSGD | detailed in section 2.3.4 - Recommender Engine | • existing data quality issues and its impact on data processing stabilization may impact test results
  • upon data quality fixing conclusion, new dataset will be provided for tests and model tuning, extending the volume (250,000 clients), diversity (remove focus on top clients) and time span (2 months) |

Table 2.8: STREAMLINE features tests
3 Rovio

Rovio’s big data pipeline collects data from game clients, services and external systems. Data is aggregated and analysed to produce user profiles with features such as registration date, last seen timestamp, average session length, churn score, ads shown and money spent. Profiles are processed further to produce KPIs like daily new and returning users, retention, total in-app purchase and ads revenue, conversion rate and average revenue per user. User profiles are also used for service targeting purposes; for example, we may disable interstitial ads from spenders, prioritize support tickets based on user churn propensity or ban cheaters from game events.

Rovio’s main goal is to utilise STREAMLINE to improve the delivery time and self-service capabilities of player profiling and games business reporting. Rovio’s use cases are summarised in the next paragraphs. Some of the use cases we worked on year two of STREAMLINE did not end up to pilot or production phase because we either did not have time to deploy the solution yet or we decided to use alternative technology. Each use case contains a deployment status information. KPIs and solution overview will focus on solutions that have been deployed.

**Use case 1: Custom Real-Time Reporting Dashboards**

*Deployment status: LIVE.*

Rovio reporting is mostly batch based where profiles, KPIs and raw data in columnar format is available in the next morning. We provide a lot of customisation for this batch data. Users can create their own dashboards and query the data using for example Redshift, Presto or Athena. However, there are scenarios when real-time reporting is required. For example, in ads we want to know that all our placements are working as expected and get the information about failures as soon as it happens so we can fix it before significant revenue loss. Other use case is in app purchases; we want to follow the trends of IAP products per country and application store so we notice potential problems in our catalogue prices or payment services. Analytics data is available in real-time in our Kafka topics, but we provide no means of accessing that data beyond traditional development tools that are quite technical.

Purpose of custom real-time reporting dashboards is to provide game teams the ability to easily create real-time dashboards using declarative language for aggregation rules and then configurable dashboard software to implement the views.

**Use case 2: Kafka Backup to S3**

*Deployment status: Deployed on alternative technology Secor.*

At the start of the STREAMLINE project most of the Rovio data was still using an old version of Kafka (0.7) that was not supported by Apache Flink. We wanted to migrate our games and services to use the latest Kafka versions so we can more easily integrate to Apache Flink and also get the benefits of bug fixes and new features especially around high availability. As most of the Rovio reporting still
happens on the batch data it is really important that the raw data in the Kafka is also made available to S3. The data copying from the old Kafka clusters is handled by modules called Kafka pull jobs. These are in-house built Hadoop cascading jobs running on AWS Elastic MapReduce clusters that are launched on an hourly basis. They read the all the new data since last read from the configured Kafka topics and store the data to S3. This system has many problems:

- they are expensive due to inefficient partition handling and hourly bootstrapping overhead
- they are difficult to operate due to complicated in-house built offset management
- they fail often e.g. due to normal AWS provisioning errors

The replacement system should instead use streaming technology to store the data to S3. System should be always on and write the data to S3 as soon as it arrives. System must have very little maintenance required and should be very reliable.

Backup must implement exactly once semantics; we must get all the data to disk and it must not contain duplicates. System must also be scalable. We must easily scale up when data volumes grow. System must be error tolerant. We must be able to stop and start the system so that no events are lost and no duplicates are generated.

Additionally, we must be able to control the size of the output file and maximum interval between data flush to disk. This way we can try to get optimal file sizes, but also ensure that files are written to disk often enough.

Use case 3: Portfolio ID Graph Analysis

Deployment status: Deployed on alternative technology Spark GraphFrames.

Rovio has many games and we want to track the player journey across the entire portfolio so we can e.g. do profitable user acquisition. None of the Rovio games force creation of a player account or linking to a 3rd party network such as Facebook. Player analysis is done using mainly following hashed IDs:

- Advertiser Identifier - This identifier is global across all apps but can be reset by end user from the handset. You can also disable this altogether using limited ads tracking.
- Device Identifier - This identifier is unique across all the apps from the same publisher (like Rovio). However, this identifier may reset if player uninstalls all Rovio games or changes the devices.
- Player Identifier - This is Rovio’s own auto generated identifier for the user. This is unique only in one application and cannot be directly used across all applications. If user links his account with e.g. Facebook this identifier can be used to identify player across multiple devices.
- Facebook Identifier - This identifier is unique inside one application only and not suitable as such for portfolio analysis

Previously we have used traditional join operations to link different accounts to build a portfolio profile for the player. However, these rules are complicated and do not really give the satisfactory
results. This use case is about using graph analysis (connected components) to build player identifier clusters that help us to track the player journey across Rovio applications. In STREAMLINE project, we compared Apache Flink Gelly library with Apache Spark GraphFrames API to implement a daily process that runs graph analysis over Rovio data.

**Use case 4: Churn prediction pipeline**

*Deployment status: Deployed, but does not have the Flink enrichment included yet.*

User acquisition is expensive so you want to retain your players. Purpose of churn prediction is to identify the players most likely to churn so that we can target them with personalised content. Prior to STREAMLINE, Rovio was running a churn prediction pilot for one game. The pilot was based on Redshift and R. This solution had following issues:

- solution did not scale well for games with larger daily average user count
- additional infrastructure had to be maintained to manage the R components
- feature extraction was using game specific legacy datasets which required us to run an obsolete pipeline for this single purpose

Target of this use case is to replace the R based legacy system with a solution that uses our selected big data stack and our common datasets. We evaluate the available machine learning libraries and select the best tool for the job. Additionally, we look into extracting some of the features in real-time to simplify the migration work if streaming solution is required in the future.

**Use case 5: Integration to 3rd party systems**

*Deployment status: Live deployment on December 2017 with key partner. Discontinued with two other partners.*

Mobile attribution enables Rovio to track all of user acquisition channels and aggregate conversion data for deeper analysis, by ad clicks or ad impressions. Rovio analyses media performance across multiple advertising channels: track and attribute incoming users to sources they came from on a daily basis. Currently Rovio uses multiple vendors: one for user attribution and the other for cost aggregation. This causes following issues:

- Campaign identifiers from attribution data differ from cost, which makes mapping the spend with the revenue complicated and requires additional work
- Troubleshooting requires working with multiple vendors with different levels of support

Rovio wants to consolidate their user acquisition stack and pilot a 3rd party system that offers both user attribution as well as cost aggregation capabilities for performance marketing purposes. However, Rovio does not want to introduce new SDKs into the actual game clients because:

- Rovio takes user privacy very seriously and must be compliant with regulations such as COPPA and GDPR: we want to be able to control which players are profiled and what data is sent to 3rd party systems
• We want to quickly add and remove new 3rd party vendors. Having an SDK would require new version of the game to be submitted when vendors are added and removed.

• We want to control the size and other resources of the game package by not including additional libraries

Instead of SDK integration, STREAMLINE project is used to build reliable Flink streams that mediate the data between Rovio and 3rd party vendors. As Rovio had not yet run the pilot at the time of the writing this document only describes the solution but does not have any measurable KPIs. The results of the pilot and potential production use will be reported in D5.6.

3.1 KPI 1: At least one new use case implemented
The main STREAMLINE KPI for Rovio was to introduce at least one new streaming use case.

3.1.1 Baseline and Target measures
Rovio did not have comparative real-time analytics platform in use prior to Apache Flink and did not have any real-time analytics use cases implemented.

3.1.2 Results
Rovio has deployed two use cases into production: custom real-time reporting dashboard and integration to 3rd party systems.

3.2 KPI 2: Business KPI's and customer profiles have a maximum latency of one hour
Purpose of this KPI is to measure the maximum latency of real-time dashboards and player profile updates.

3.2.1 Baseline and Target measures
Rovio did not have a real-time reporting and profiling system in place before STREAMLINE project and latency was not yet measured in D5.2 which focused more on the system reliability.

3.2.2 Results
Current real-time dashboard and integration to 3rd party systems has the latency of configured window size (e.g. 60 seconds). Real-time profile updates are planned for D5.6.

3.3 KPI 3: Service Uptime Percentage
Rovio Games are developed and operated globally. It is therefore required that all services including analytics provide high service level with 24/7 support.

This KPI is about measuring the service level of Apache Flink and STREAMLINE based features using service life-time uptime percentage. Service life-time uptime percentage is calculated by subtracting from 100% the percentage of minutes when system was not in state “OK” for reasons other than scheduled maintenance. We measure the uptime of all the Flink streams in production. Service life-time is calculated from the beginning of the year 2017. The measurement data is retrieved from Nagios monitoring system.
3.3.1 Baseline and Target measures

Rovio target uptime percentage is 99.97%. Last year we measured a 99.99% uptime percentage for STREAMLINE use cases.

3.3.2 Results

The data from Nagios monitoring system suggests that service-uptime was 99.3% with scheduled maintenance breaks included (Figure 3.1).

![Service 'Flink streams' On Host 'cloud'

2017-01-01 00:00:00 to 2017-11-23 11:55:57
Duration: 326d 11h 55m 57s

<table>
<thead>
<tr>
<th>Service State Breakdowns:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State</strong></td>
</tr>
<tr>
<td><strong>OK</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>WARNING</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>UNKNOWN</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>CRITICAL</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Undetermined</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>All</strong></td>
</tr>
</tbody>
</table>

When looking at the service log entries we can see that traditional maintenance breaks cause roughly 15 minutes service downtime when we restart the Flink cluster. Also, there was a larger service break in September which was actually caused by a network issue between monitoring system and Flink instances. Based on the raw data we did not reach our availability to targets. For next year we consider alternative way of measuring the service availability as having the Flink restarts and maintenance operations do not result end service down time.
3.4 KPI 4: Ease of development
This is a qualitative KPI to measure human latency in implementing use cases on STREAMLINE and Apache Flink as opposed to Apache Spark. We use the Portfolio ID Graph Analysis use case to measure this, as in that use case we first implemented the baseline with Apache Spark.

3.4.1 Baseline and Target measures
Initial implementation was done using Apache Spark GraphFrames API. Development on that API was easy, but we hit a bug and wanted to try the implementation with Apache Flink.

3.4.2 Results
The implementation of actual graph analysis was rather trivial on both frameworks. However, Apache Flink lacks behind on processing different data formats. We used ORC data format and we were unable to find good official guides how this is done. Information was scattered in different unofficial blogs and repositories, some based on older versions of APIs. This resulted that most of the time in Flink implementation was spent on integration with ORC file format which in Apache Spark is just one function call. To build an efficient hybrid streaming and batch platform it is important to provide better support for batch processing.
3.5 Solution Description

Rovio's real-time STREAMLINE use cases are implemented into the Rovio analytics pipeline, which is described in Figure 3.2. Data is collected from game clients, internal services and 3rd party systems. Internal data is collected by exposing a REST api for receiving analytics events which are then stored into Apache Kafka. Same approach is also used for some 3rd party systems where streaming is required. One of such systems is an attribution provider which aggregates user acquisition data from different networks and sends Rovio application install events with user origin information. Most of the 3rd party systems are integrated by pulling daily reports using a daily ETL process. This reporting data is not stored into Apache Kafka, but instead written directly to S3.

Data from Kafka is processed by two systems: batch processing and real-time.

In batch processing, data is periodically stored or streamed to S3 as daily partitioned data sets. These raw data sets are then processed on a daily basis with Elastic MapReduce jobs to produce daily aggregates and columnar ORC datasets. Daily aggregates are stored to an Amazon Redshift database and are then analyzed there to produce user profiles and KPI’s. User profiles and KPI’s are stored on different serving layers such as Cassandra database for real-time player segmentation, QlikSense database for Games Business Intelligence dashboards and Amazon RDS for Rovio internal Beacon dashboard. Profiles are also stored to columnar ORC data format. The columnar datasets are queried from different systems such as Presto, Amazon Athena and Spark to produce reporting dashboards or additional player insight using machine learning algorithms.

Real-time systems connect directly to Kafka and process the data in a streaming fashion. Currently all of the streaming use cases are implemented with Apache Flink. In the second year of STREAMLINE project we have two active use cases running in production: integration to 3rd party systems and real-time grafnana dashboards. These are discussed in more detail in following subsections.
All analytics jobs, including Apache Flink streams, are scheduled using Azkaban workflow manager. Analytics jobs, services and data sources are monitored with Nagios server monitoring software. Additionally, we have a higher level of monitoring and service orchestration system in Rovio. Most importantly, in the context of STREAMLINE, we can use TeamCity continuous integration system to deploy Flink streams automatically to test, staging and production environments whenever we merge pull requests in our GitHub repositories.

### 3.5.1 Building FlinkJobs projects

All Apache Flink streams and batch jobs are stored in a GitHub repository named FlinkJobs. Projects are built with maven. To build a project go into the project folder and run the command described in Listing 3.1.

```sh
$ mvn clean package
```

*Listing 3.1: Building STREAMLINE projects with maven*

This will create an uber-jar suitable for submitting to Apache Flink. Jobs are submitted either by using a command line tool or Azkaban.

### 3.5.2 Deploying FlinkJobs using command line

To start the jobs from command line run the command described in Listing 3.2.

```sh
$ ssh ubuntu@10.1.2.8
$ aws s3 cp s3://ds-analytics-emrjobs-cloud/flink/omniata-stream/scripts/launch-flink-cluster-cloud.sh ./
$ chmod u+x launch-flink-cluster-cloud.sh
$ ./launch-flink-cluster-cloud.sh
# wait/check that the job starts
$ listactive | grep omniata-abisland-stream
# Flink UI to check that records are processed
$ python flink-ui.py `listactive | grep omniata-abisland-stream | cut -d " " -f1` | grep FlinkUI
```

*Listing 3.2: Deploying FlinkJobs using command line*

The command line utility described above was used in Omniata stream only. The preferred way to launch Flink jobs is to schedule them through Azkaban.

### 3.5.3 Azkaban Workflow Manager

Processing of analytics pipeline jobs and streams is orchestrated with Azkaban scheduler (Figure 3.3). Flink Azkaban Plugins are used to schedule Flink batch jobs and start/stop Flink streams in Amazon Elastic MapReduce clusters.
3.5.3.1 Flink Batch Job Plugin

Flink batch job plugin is used to run an Apache Flink batch job on an EMR cluster. Example job configuration is described in Listing 3.3.

```plaintext

```type=flinkbatch```

```name=Profiler DNU Initialization```

```
# -c is used to specify the main class unless the jar has it defined by itself
step.1.options.-c=com.rovio.ds.DNURuleInitiate
step.1.jar=/home/hadoop/flink-jobs/profiler-ab-testing-1.0.0-SNAPSHOT.jar
step.1.args.--input_data_path=s3n://ds-analytics-raw-
$({rovio.env}/hoarder/topic=audit.supermoon/processdate*/
step.1.args.--path_s3=s3n://ds-analytics-aggregate-$({rovio.env}/profiler/ab-testing/

cluster.master.type=m1.large
cluster.core.type=m1.large
cluster.core.count=1
```

Listing 3.3: Flink batch job configuration example

The job configuration parameters are described in Table 3.1.

<table>
<thead>
<tr>
<th>Key</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
<td>Name of the EMR cluster. This value is shown for example if you list active EMR clusters with the AWS CLI</td>
</tr>
<tr>
<td>cluster.inVpc</td>
<td>false</td>
<td>Setting value to true will run the cluster in Virtual Private Cloud. This is required to access some resources such as Kafka.</td>
</tr>
</tbody>
</table>
### 3.5.4 Flink Streaming Job Plugin

Flink Streaming job type can be used to run a Flink stream in an EMR cluster. The supported properties are the same as for Flink batch job type. The only difference is that the job type parameter is “flinkstream”. Listing 3.4 describes an example configuration for a streaming job.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>step.${i}</td>
<td>Prefix for step specific properties, for example step.1.</td>
</tr>
<tr>
<td>step.${i}.options.${key}</td>
<td>Prefix: any flink options passed to flink start job command before the actual job jar.</td>
</tr>
<tr>
<td>step.${i}.jar</td>
<td>Jar file as found on the master node after being copied from S3 by a common bootstrap action.</td>
</tr>
<tr>
<td>step.${i}.args.${key}</td>
<td>(none) Prefix: any args to be passed to the flink job (main class)</td>
</tr>
<tr>
<td>step.${i}.name</td>
<td>Name of the EMR step</td>
</tr>
<tr>
<td></td>
<td>${jar} split('/').last()</td>
</tr>
<tr>
<td>aws.emr.</td>
<td>Properties than control the behaviour of EMR cluster behaviour on error. For example:</td>
</tr>
<tr>
<td></td>
<td>• aws.emr.actionOnFailure=CONTINUE</td>
</tr>
<tr>
<td></td>
<td>• aws.emr.autoTerminate=false</td>
</tr>
<tr>
<td>cluster.${group}</td>
<td>Prefix for EMR instance and types. For example:</td>
</tr>
<tr>
<td></td>
<td>• cluster.master.type=m1.medium</td>
</tr>
<tr>
<td></td>
<td>• cluster.core.type=m1.large</td>
</tr>
<tr>
<td></td>
<td>• cluster.core.count=1</td>
</tr>
<tr>
<td>flink.savepoints.enable</td>
<td>false This parameter can be used to enable Flink save points and periodic external checkpoints.</td>
</tr>
<tr>
<td>flink.fromSavepoint</td>
<td>Flow override only: to resume from a specific savepoint or checkpoint instead of determining the latest restore point automatically. To not restore state at all, set the special value false.</td>
</tr>
<tr>
<td>step.1.args.--checkpoint.interval</td>
<td>60000 Frequency of periodic checkpoints in milliseconds.</td>
</tr>
</tbody>
</table>

Table 3.1: Flink job configuration parameters
3.5.5 Automatic recovery from Savepoints and Checkpoints

Flink itself supports automatic recovery from checkpoints while the job is being re-run according to Flink’s restart-strategy. However, Flink doesn’t restore anything automatically after a job has used all of its retries, or has been shut down for any other reason.

Before launching a new cluster, our Flink Job Plugin scans the configured checkpoint and savepoint directories to find the latest savepoint or checkpoint to restore from. That path is then passed to flink run command as --fromSavepoint. This behaviour can be disabled or overridden by specifying the savepoint path manually.

To enable checkpointing we set the parameters defined in Listing 3.5 in flink-conf.yaml as part of cluster bootstrap.

```
state.checkpoints.dir: s3://bucket/flink/checkpoints/${JOB_NAME}/
state.savepoints.dir: s3://bucket/flink/savepoints/${JOB_NAME}/
```

Listing 3.5: Checkpoint configuration parameters
Unfortunately, there’s no way to specify those parameters in the Flink job itself (Java code in our case). This means that it would be problematic to determine what the latest checkpoint is for a given job, if different flink jobs would be run on the same cluster (ie. using the same flink-conf.yaml), because checkpoints from all jobs would be written in the same location.

3.5.6 Singular Integration

Singular is an attribution provider whose responsibility is to resolve the user origin of the player and optimise the user acquisition campaigns. Traditionally attribution providers are integrated to the actual end user app using an SDK, but in this case Rovio wanted to integrate using server to server architecture. Main downstream integration is done using Flink streaming. We read our event data from Kafka topics, join, filter, transform and then send the data to their event API. Attribution and reattribution data is received from singular using the postbacks that are processed by our UA service. UA events are aggregated and analysed by our pipeline to determine the user origin. High level architecture of Singular Integration is described in Figure 3.4.

On high level the Flink stream does following:

- Data is filtered based on Kafka topic and custom rules that specify whether the event should be reported to 3rd party system
- Device ID to Advertiser ID mapping is stored into Flink state because Advertiser ID is not included in all input requests, but must be included in all output requests
- Event time window with configurable size is maintained to ensure that we have the Advertiser ID data available for all the events we write out from the system
- Data is written to HTTPS endpoint. Flink’s Asynchronous I/O feature is used for External Data Access to parallelize the output.
Resulting Flink topology is described in Figure 3.5.

![Singular streaming job topology](image)

Figure 3.5: Singular streaming job topology

Job uses JSON configuration file that defines the e.g. what applications and events are streamed to the Singular endpoint. Table 3.2 describes the configuration parameters.

Note that the CounterSink is only set to a lower parallelism to force rebalance, so that we can easily see output metrics after the async call on top level instead of having to dig deeper for sub-tasks (this has nothing to do with the sinksPerKafkaSource parameter).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sinksPerKafkaSource</td>
<td>This parameter is used to control the parallelism of the sinks. For example if we have four kafka sinks and sinksPerKafkaSource is set to 2, then sink parallelism is eight.</td>
</tr>
<tr>
<td>asyncSink</td>
<td>Enable the Asynchronous I/O for output sink. This should be enabled for high latency backends.</td>
</tr>
<tr>
<td>asyncParallelism</td>
<td>This parameter defines how many asynchronous requests may be in progress at the same time.</td>
</tr>
<tr>
<td>asyncTimeoutSeconds</td>
<td>The timeout defines how long an asynchronous request may take before it is considered failed.</td>
</tr>
<tr>
<td>sleepBeforeRetryMillis</td>
<td>Delay before retrying event sending to backend after failed attempt.</td>
</tr>
<tr>
<td>kafka/bootstrap.servers</td>
<td>List of kafka bootstrap server addresses.</td>
</tr>
<tr>
<td>kafka/group.id</td>
<td>Group ID for kafka consumers.</td>
</tr>
<tr>
<td>topics</td>
<td>List of kafka topics that are read by the streaming job</td>
</tr>
<tr>
<td>filters/field</td>
<td>Event field name that defined filter is applied to</td>
</tr>
<tr>
<td>filters/include</td>
<td>List of regular expression for acceptable field values</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>apiKey</td>
<td>Singular API key</td>
</tr>
<tr>
<td>apiUrl</td>
<td>Singular API url</td>
</tr>
<tr>
<td>discardedEventsPath</td>
<td>Path where events are written that could not be sent to Singular</td>
</tr>
<tr>
<td>stateBackend</td>
<td>State backend to be used. Either MemoryStateBackend, RocksDBStateBackend or FsStateBackend</td>
</tr>
<tr>
<td>timeWindowInSeconds</td>
<td>Time window for waiting when joining the data from multiple sources. Main use case is to delay the sending of events before receiving the Advertiser ID.</td>
</tr>
<tr>
<td>maxOutOfOrderlinessInSeconds</td>
<td>Events might arrive out of order and this setting defines how much older events we process before discarding them.</td>
</tr>
</tbody>
</table>

Table 3.2: Singular job configuration parameters

The JSON configuration file is described in Listing 3.6.

```json
{
  "sinksPerKafkaSource": 1,
  "asyncSink": true,
  "asyncParallelism": 500,
  "asyncTimeoutSeconds": 60,
  "sleepBeforeRetryMillis": 5000,
  "kafka": {
    "bootstrap.servers": "kafka8v-01:9092,kafka8v-02:9092,kafka8v-03:9092",
    "group.id": "singular-stream-cloud-3"
  },
  "topics": [
    "audit.wallet",
    "collector.Seaside"
  ],
  "filters": [
    {
      "field": "m.t",
      "include": [
        "PlatformIDs",
        "App Comes Foreground",
        "AppleSearchAdsAttribution",
        "AndroidReferral"
      ]
    }
  ]
}
```
3.5.7 Configurable Streaming Aggregation Job

In Configurable Streaming Aggregation use case, we implement a generic Apache Flink job where the aggregation rules are defined in a declarative manner. The input is analytics events in JSON. Users can refer to arbitrary JSON fields in their configurations, so the aggregation jobs don't have to depend on any pre-defined schema. The declarative configuration defines what Kafka topics to read, which fields to group by, window size, and pairs of aggregate function and JSON field. Additional filters can be defined.

The job writes to InfluxDB, a time-series database. InfluxDB doesn't require creating a table schema in advance. The Flink job can create new measurement types by just sending the data in. New measurements and their fields are discovered by Grafana automatically. Real-time (near) dashboards are created using the tools offered by Grafana's web UI.

3.5.7.1 Supported aggregation features

The set of features is rather limited, if compared to the expressivity of full-blown SQL syntax. We look forward to removing our custom code in favour of Flink SQL, when GroupWindows in Stream SQL becomes available. Our aggregate job implements the following features:

- Aggregate functions: count, distinctCount (implemented with HyperLogLog), min, max, sum, avg
- Filters: equal, regex, and, or, not (specified as inverse=true on any other filter)
- Time window: value and any timeunit of java.util.concurrent.TimeUnit
- Measurement: the name of target "table" in InfluxDB (ie. measurement). Configuration must also include the aliases for InfluxDB datapoint tags (the fields to group by) and fields (aggregated values).

Multiple measurements can be produced by a single instance of the job. Measurements are defined as a list in the job configuration. Different measurements may share Kafka topics as their input. The

```json
{
  "encryptionKey": "incoming_data",
  "receipt_purchase",
  "attribution_info"
}
```

Listing 3.6: Singular job configuration example
Kafka stream is split after reading from Kafka, i.e., each topic is only read once by the Flink job even if different measurements require it.

### 3.5.7.2 Configuration example

The configuration described in Listing 3.7 produces three different measurements:

- active_session, ads_campaigns & wallet_purchase
- Each measurement produces one or more aggregated fields

```json
{
    "kafka": {
        "group.id": "aggregate-all_server_measurements_combined"
    },
    "parallelism": 8,
    "measurements": [
        {
            "name": "active_session",
            "topics": [
                "audit.session",
                "audit.identity",
                "audit.wallet"
            ],
            "tags": {
                "s.cid": "app_id",
                "s.dcid": "distribution_channel",
                "s.cver": "client_version"
            },
            "fields": [
                {
                    "function": "distinctCount",
                    "source": "s.aid1",
                    "target": "unique_users"
                },
                {
                    "function": "count",
                    "source": "*",
                    "target": "event_count"
                }
            ],
            "windowSize": {
                "value": 60,
                "unit": "seconds"
            }
        }
    ]
}
```
"name": "ads_campaigns",
"topics": [
  "audit.ads"
],
"tags": {
  "s.cid": "app_id",
  "m.campaign": "campaign",
  "m.zone": "placement",
  "m.networkName": "network",
  "t.geo": "country"
},
"fields": [
  {
    "function": "count",
    "source": "*",
    "target": "impressions"
  }
],
"windowSize": {
  "value": 60,
  "unit": "seconds"
},
"filters": [
  {
    "field": "m.t",
    "type": "equal",
    "value": "ads.impression.1"
  }
],

"name": "wallet_purchase",
"topics": [
  "audit.wallet"
],
"tags": {
  "s.cid": "app_id",
  "s.dcid": "distribution_channel"
},
"fields": [
  {

```json

```
```
Listing 3.7: Custom aggregation job example configuration

The resulting Flink Job DAG is described in Figure 3.6.
Example dashboard in Grafana from simulated test data can be seen in Figure 3.7.

Figure 3.7: Example Grafana dashboard
Example of Grafana’s tools for building dashboards are shown in Figure 3.8.

3.5.7.3 Flink/Hadoop environment

We build an uberjar of the Flink job which can be launched with Azkaban on Amazon EMR, running flink on Hadoop/YARN. We had a dependency conflict between InfluxDB client library and Hadoop that made the job fail. To fix the conflict we included a modified version of Guava in the job jar using the maven-shade-plugin. Later on, we have faced similar conflicts with some other commonly used java libraries, and fixed those as well by relocating the conflicting packages with maven-shade-plugin.

3.5.8 Offset Monitoring

We monitor job health and lag by looking at the Kafka consumer offsets. For this we use the "Kafka Offset Monitor" ([https://github.com/quantifind/KafkaOffsetMonitor](https://github.com/quantifind/KafkaOffsetMonitor)). Offset Monitor UI screenshots can be seen in Figure 3.9 and Figure 3.10.

[https://github.com/quantifind/KafkaOffsetMonitor](https://github.com/quantifind/KafkaOffsetMonitor). Offset Monitor UI screenshots can be seen in Figures 4.9 and 4.10.

[https://github.com/quantifind/KafkaOffsetMonitor](https://github.com/quantifind/KafkaOffsetMonitor). Offset Monitor UI screenshots can be seen in Figures 4.9 and 4.10.

[https://github.com/quantifind/KafkaOffsetMonitor](https://github.com/quantifind/KafkaOffsetMonitor). Offset Monitor UI screenshots can be seen in Figures 4.9 and 4.10.
Details for the consumer group aggregate-all_server_measurements_combined

Brokers

<table>
<thead>
<tr>
<th>id</th>
<th>host</th>
<th>port</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>kafka1-cloud-haaredelevents-10-1-16-14.cloud-internal.rvio.com</td>
<td>9082</td>
</tr>
<tr>
<td>163</td>
<td>kafka1-cloud-haaredelevents-10-1-16-163.cloud-internal.rvio.com</td>
<td>9082</td>
</tr>
<tr>
<td>162</td>
<td>kafka1-cloud-haaredelevents-10-1-16-162.cloud-internal.rvio.com</td>
<td>9082</td>
</tr>
</tbody>
</table>

Consumer Offsets

<table>
<thead>
<tr>
<th>Topic</th>
<th>Partition</th>
<th>Offset</th>
<th>logSize</th>
<th>Lag</th>
<th>Owner</th>
<th>Created</th>
<th>Last Seen</th>
</tr>
</thead>
<tbody>
<tr>
<td>audit.ads</td>
<td>0</td>
<td>586931822</td>
<td>586931979</td>
<td>157</td>
<td>NA</td>
<td>a few seconds ago</td>
<td>a few seconds ago</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>585788391</td>
<td>585788786</td>
<td>397</td>
<td>NA</td>
<td>a few seconds ago</td>
<td>a few seconds ago</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>585000280</td>
<td>585000833</td>
<td>553</td>
<td>NA</td>
<td>a few seconds ago</td>
<td>a few seconds ago</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>586981469</td>
<td>586994543</td>
<td>1297</td>
<td>NA</td>
<td>a few seconds ago</td>
<td>a few seconds ago</td>
</tr>
</tbody>
</table>

Figure 3.9: Kafka Offset Monitor

Figure 3.10: Monitoring lag with Kafka Offset Monitor
3.5.9 Nagios Monitoring

To keep track of long-running Flink stream jobs, we wrote a Nagios script to check the status of Flink jobs using Flink’s monitoring REST API. The script enumerates EMR clusters and selects the ones where Flink is running. Then the monitoring API is used to check job status and the dynamics of read-bytes and write-bytes metrics, and our job-specific Flink accumulators. We also check backpressure from Flink REST API.

![Service Status Details For All Hosts](image)

Figure 3.11: Nagios monitoring dashboard with “Flink Streams” monitor

Nagios monitoring is configured to send alerts to Rovio 24/7 cloud support in case of critical service issues. Example screenshot from Nagios monitoring dashboard can be seen in Figure 3.11.

3.5.10 Flink logs in Kibana

Flink UI allows browsing raw logs files, but response times become huge as the log files grow big. We use filebeat to find the YARN/Flink logs based on a file path pattern, and to stream the logs to Rovio’s deployment of ELK stack. As a result, we are able to search all of our Flink job logs in one place, Kibana, and also display dashboards for error logging trends.
The screenshot in Figure 3.12 shows a rising trend of Flink error logs when InfluxDB was down, until it was fixed.

![Flink logs in Kibana](image)

**Figure 3.12: Flink logs in Kibana**

### 3.6 STREAMLINE features

STREAMLINE features used by Rovio use cases are described in Table 3.3.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Use case</th>
<th>Relevance</th>
<th>Availability</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault tolerance: Managing stateful streaming (T1.2)</td>
<td>All</td>
<td>Critical</td>
<td>Available</td>
<td>To ensure data correctness and no loss of data, all of our use cases require fault tolerance. For example, our custom aggregation job requires all the partial aggregations to be stored in the state. Upon receiving new data, states are access to update the partial aggregate and forward to the result to our dashboards. The 3rd party integration use case uses state to store the advertiser ID’s that can then later be enriched to all the outgoing events. States are periodically checkpointed. When a failure occurs, the computation is restarted and the latest checkpointed data are load</td>
</tr>
</tbody>
</table>
The hoarder component assigns a timestamp to each of the incoming events. To ensure that the order of the incoming data is preserved, we use the assigned timestamp. T2.2 allows to extract the timestamps from the events and assign watermarks which automatically re-processes out of order events and manages the window until all the events are processed. This feature is used for example in 3rd party integration use case where a combination of state and windows is used to join data from multiple Kafka topics into enriched output event.

Table 3.3: STREAMLINE features used in Rovio use cases

<table>
<thead>
<tr>
<th>Feature</th>
<th>Planned Tests</th>
<th>Test Details</th>
<th>Target Deliverable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault tolerance: Managing stateful streaming (T1.2)</td>
<td>System performance and stability in production use</td>
<td>Data correctness and no loss of data are still important aspects of our future use cases and enhancements to existing use cases. Field trials and evaluation focus on ability to deploy new use cases with the feature and performance and stability of the deployed system.</td>
<td>D5.6</td>
</tr>
<tr>
<td>Window Operation Semantics (T2.2)</td>
<td>System performance and stability in production use</td>
<td>Event time and out of order data processing are key features in future Rovio use cases and enhancements. Field trials and evaluation focus on ability to deploy new use cases with the feature and performance and stability of the deployed system.</td>
<td>D5.6</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>A High-Level Declarative Language for Machine Learning at Rest and in Motion (T3.2)</td>
<td>System performance and stability in production use</td>
<td>In the future use cases we would like to utilise for the real-time aggregation rules. However, we might not use it for machine learning. Field trials and evaluation focus on ability to deploy new use cases with the feature and performance and stability of the deployed system.</td>
<td>D5.6</td>
</tr>
</tbody>
</table>

Table 3.4: STREAMLINE features planned to be used in future Rovio use cases
4 Internet Memory Research

Internet memory collects product information online using its crawling-scraping technology. In order to organize the collected data, it uses a categorization model to predict the category of each crawled product. This model is trained on a basis of a set of products that comes from a similar data stream.

The products for training come from a set of so-called “golden sites”. Since the list of products available online evolves, as well as the global catalog that maps these product to a large hierarchy of categories/sub-categories, golden sites have to be periodically recrawled, and the classification model has to be periodically rebuilt. This is currently done by retraining the model from scratch on the basis of the previous database plus the current increment. This task is very time consuming as during the process the previous iteration of the model is completely disregarded.

Product descriptions are extracted from web pages collected by our crawler. The extractor module relies on a combination of wrappers that cover a large range of page layouts, and heuristic that help to refine, or sometimes completely replace, the wrapper result. Typical heuristics attempt for instance to identify the price of a product based on its closeness to other essential information: product name, product image and brand. It may happen, however, that an extraction fails for a whole range of pages, due to some unexpected page layout or broken wrapper. It may also happen that the classification process itself fails in the presence of an unknown category of due to insufficient input data in the product description. To cope with such failures, whatever the reason, we store all collected pages in a large HBase cluster, and sometimes run a classification workflow from this static database in order to make the extraction/classification right.

The product classification is an essential component of our systems, because most of our services rely on the accuracy and timeliness of the product database. As Streamline project deals with simultaneous processing data in motion and data at rest, we decided to focus on the optimization of the categorization workflow. We identified three use cases related to this workflow.

Use Case 1: Product categorization - parallelization of training on streams
Currently IMR is using a complex workflow, only parts of which are successfully parallelized - namely data cleaning and pre-processing but not the training itself. From the effort in this use case we hope to achieve parallelization of training, currently implemented as a centralized python workflow.

Use Case 2: Product categorization - parallel incremental training on streams
As explained above, the classification model is periodically rebuilt from scratch to cope with changes affecting the catalog categories and their description. This induces a latency that affects the quality of the model, since this quality decreases from the moment the model is built to the moment it is replaced. This also incurs a human and system overhead to select the training set, run the training phase, check the quality of the new model and replace the old model with the new one if the previous steps are successful.

We want to exploit the ability of Streamline Flink to supply new ML methods on streaming data to overcome these limitations. Our expectations are first that the model should evolve as new “golden data” arrives in the system, thereby maintaining its quality. Second, although we anticipate that full rebuild might be necessary from time to time, we hope that this will become rare and save the efforts currently spent.

Use Case 3: Product categorization – unified streams and batch workflows
IMR currently implements the categorization workflow as a sequence of MapReduce jobs over the input split into batches. This method is applied to both incoming data freshly captured by the crawler, and to legacy pages that need to be reclassified. The Streamline Flink architecture that we are currently investigating should enable a full streaming processing, eliminating the latency between the moment a page is captured by the crawler and the moment a classified product extracted from this page is stored in the product DB. Moreover, in order to limit investments in development and maintenance, the workflow operating on streaming data and the workflow on legacy data should be as similar as possible. For brevity, we will call the model produced using current centralized algorithm as centralized model, the one produced using parallel algorithm as parallel model and the one using incremental training as incremental model.

4.1 KPI 1: Model’s prediction precision
Using a ground truth dataset, every time the model is created its precision is assessed. The goal is to maintain the same precision with Streamline Flink than the current one.

4.1.1 Current system
The model is built separately from the main data acquisition workflow, based on the Scikit-learn Python implementation of the Passive-Aggressive algorithm.

4.1.2 Baseline and Target measures
The ground truth is established using a dataset whose documents are not present in the training set. For Use Cases 1 and 2 the comparisons between the centralized, parallel and incremental models will be used to determine the possible deteriorations of precisions using various approaches. Ideally, the precision should not deteriorate as a function of the algorithm used. Given that the current precision is 83.3%, we would not expect to downgrade the precision below 80% as the price to pay for increased scalability.

4.2 KPI 2: maintenance of a high precision
The precision of the model decreases as new categories are discovered, and old ones are deprecated. The incremental update of the model should limit this deprecation effect.

4.2.1 Current system
We rebuild the model periodically, approximately every month, but so far this rebuilt is not triggered by a ceil value of the precision.

4.2.2 Baseline and Target measures
Assuming we manage to obtain a high initial precision level with the Streamline Flink implementation (at least 80%, as explained above), we aim at keeping an almost constant precision level during the first month that follows a full rebuild. This ensure that the need to rebuild a model will appear, if any, after more than one month.
4.2.3 Status in M24
We are currently implementing a precision tracker that measures the precision of the model throughout its exploitation. The results of this tracker will be reported in D5.5 and will serve as a baseline for the KPI evaluation.

4.3 KPI 3: Start-over Training Necessity
Using the precision measure comparison for the centralized (or parallel) and incremental model we detect possible deterioration in the quality of the prediction of the incremental model. If the deterioration surpasses a given threshold, the input model for incremental training will have to be replaced with fresh parallel model.
This KPI is to measure how often this costly operation of incremental model reset has to take place.

4.3.1 Current system
Currently, the model is rebuilt from a full training set once a month.

4.3.2 Baseline and Target measures
The new solution should result in a significant improvement. We adopt as a suitable perspective a rebuilt of the model every three months at most.

4.3.3 Status in M24
This KPI depends on the availability of the Passive Aggressive classifier in Streamline Flink.

4.4 KPI 4: Training phase time
Currently, the training phase is a centralized process that runs on one server due to incapability of parallelization of the algorithm. Using this KPI we would like to measure the time it takes to train the model given a certain training set against the centralized training algorithm.

4.4.1 Current system
Currently, we are using a python implementation of the categorization training algorithm. The model is rebuilt from a full training set once a month, and the model construction time is between 4 and 5 hours. The training set consists of ~14M labelled products.
This algorithm should be replaced by its equivalent implementation in Flink.

4.4.2 Baseline and Target measures
The baseline is established as a function of size of the training set and number of servers. Now the number of servers is set to 1 because we do not have the training algorithm parallelized. The parallelized construction should scale linearly with the number of nodes assigned to the system, and with respect to the training set size. Namely, taking as a baseline the current building costs given above; we expect that
1. The construction time should be of the order of $4/N$ hours, where 4 is the current building time, and $N$ the number of nodes, assuming a fixed training set of 14M products.
2. The construction time should of the order of $4*M/14$ hours, where $M$ is the size of the training set, and $M/14$ represents the ratio of this size with respect to the current settings (14 M products), assuming a single node.
4.4.3 Status in M24
This KPI depends on the availability of the Passive Aggressive classifier in Streamline Flink.

4.5 KPI 5: Document in training latency
Each time a new page is discovered by the crawler, it is first stored in a staging area (a WARC file in HDFS). It is then eventually included in a batch processed by a MapReduce job that carries out the workflow, including the classification. This incurs a significant latency that impacts negatively the freshness and accuracy of the Product DB. We want to lower this latency thanks to a full streaming approach.

4.5.1 Current system
The current system brings no guarantee on the time spent between a page discovery and the availability of the extracted product in the Product DB. This latency is estimated by our content management team to be approximately 1 day.

4.5.2 Baseline and Target measures
Our goal is to avoid any significant latency in the page processing workflow. A few minutes should at most be spent between the discovery of a page product and its publication.

4.5.3 Status in M24
This part has witnessed significant progress since we refactored the production workflow to include Flink as a streaming engine, and Kafka as a staging area. Unlike the other KPIs, this one does not require the availability of the PA algorithm, since the Python-based classifier is efficient enough to process a product in a few ms. We can therefore incorporate it in an efficient workflow based on streaming.

4.6 STREAMLINE features
The specific STREAMLINE features, used in IMR use cases, are identified in Table 4.1

<table>
<thead>
<tr>
<th>Feature</th>
<th>Use case</th>
<th>Relevance</th>
<th>Availability</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault tolerance: Managing stateful streaming (T1.2)</td>
<td>All</td>
<td>Minor</td>
<td>Available</td>
<td>This feature is not essential in our case, since it is essentially harmless to classify a document several times. Every document has to be classified at least once, but it is acceptable to classify them several times, so this feature has a minor impact, providing that documents involved in a failed window are resubmitted during the failover operation.</td>
</tr>
</tbody>
</table>
Table 4.1: STREAMLINE features used in IMR use cases

Table 4.2 list and describe the plan, associated steps, existing constraints and target deliverable to test Streamline functionalities in each case.

<table>
<thead>
<tr>
<th>Features</th>
<th>Planned tests</th>
<th>Test details/constraints</th>
<th>Target deliverable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window operation semantics (T2.2)</td>
<td>All</td>
<td>Important</td>
<td>Available</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Currently we divide the input in batches, and process one batch every day. Training is done on batches as well. The window semantics in Flink allows to reproduce, automatically, this mechanism.</td>
<td></td>
</tr>
<tr>
<td>Data Mining and Online learning from streams (T2.3)</td>
<td>All</td>
<td>Essential</td>
<td>D5.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Currently, the main bottleneck in our workflow is the learning/classification module. It is essential that we obtain a Streamline parallel implementation of this algorithm during the course of the project. We dispose of a centralized implementation that has been carefully tuned to obtain the best possible results. We also have training which have been communicated to SZTAKI. We will work with our partners to evaluate the forthcoming implementation, both in terms of performance and quality. Future evolutions mostly consist in introducing the PassiveAggressive classifier, currently used with a centralized Python implementation.</td>
<td></td>
</tr>
<tr>
<td>Data Mining and Online learning from streams</td>
<td>Evaluation of precision</td>
<td>We will run the Scikit learn and PA Streamline implementation on the same training set, and evaluate the precision obtained in both case by running the classifier on a batch of collected documents.</td>
<td>D5.6</td>
</tr>
<tr>
<td>Data Mining and Online learning from streams</td>
<td>Deterioration of precision</td>
<td>We will evaluate how the model precision decreases with time in the current centralized implementation, in order to measure the gain obtained with incremental learning</td>
<td>D5.5</td>
</tr>
</tbody>
</table>
Data Mining and Online learning from streams | Evaluation of global retraining necessity. | The incremental feature of the PA classifier should limit the necessity to rebuild the model too often. We will again compare with the centralized Scikit Learn implementation. | D5.5
---|---|---|---
Data Mining and Online learning from streams | Training phase duration | We will evaluate the linear scalability of the distributed implementation of the PA classifier | D5.6
Data Mining and Online learning from streams | Latency in classifying a document. | The latency is expected to be minimized with the STREAMLINE implementation. We will measure and report the average latency | D5.5

Table 4.2: STREAMLINE feature test plans
5 Gap Analysis

This chapter describes missing Apache Flink and STREAMLINE features and other technical issues each industrial partner encountered during their use case implementation and deployment.

ALB use cases identified the following gaps in Flink that, if fulfilled, will greatly improve both users and customers:

- HyperLogLog is an algorithm used for the count-distinct problem, approximating the number of distinct elements (cardinality), which is able to estimate cardinalities of >10^9 with a typical accuracy of 2%. This approach is currently in production at ALB and is an extremely valuable asset to be included in near future releases of Flink.

One major technical issue was also identified in ALB use case – some critical data quality issues were identified and reported to MEO for fixing (see section 2.3.1). Until those fixes are available, we will have some impacts on the implementation, which is contributing to some delays in the implementation and field tests.

Rovio use cases highlighted following deficiencies in Apache Flink and STREAMLINE features:

- Features such as count distinct not supported out of the box, but require custom unique checking using out-of-core state backend
- Documentation about log configuration is incomplete or outdated. It seems that when running jobs on a cluster, the only way to change log configuration is to modify the log4j configuration in Flink installation directory. This means it’s impossible to configure logging separately on a job level.
- SNAPSHOT versions are published for Flink java libraries, but no Flink distribution (tgz) is available. It would be easier to try newest Flink features if there was no need to build Flink from source by yourself.
- Batch APIs are not at the same level as with competition and are lacking features we would need like out-of-the-box support for ORC file format
- Working with state is complicated and difficult to debug. For example we had to build a lot of functionality to resume a job from latest savepoint, a feature that we would assume is provided by the system as default functionality.
- Debugging problems such as off-heap memory issues is complicated.
- Machine learning libraries are quite limited as compared to competition and thus Rovio has opted to use different technologies for its machine learning use cases and only use STREAMLINE and Apache Flink to e.g. preprocessing of feature data.

IMR use case identified the following expectations from the forthcoming Flink ML functionalities:

- The Web UI seems to show only the volume of data exchanged internally between operators, and not the data consumed from the source. This is a bit confusing, and leads to useless information in our case since our workflows consist only of a sequence of maps. It would be useful to report data amounts from the source and to the sinks.
• We are missing the ML algorithm that would be the main asset of using Flink, apart from the streaming mechanism.
6 Conclusion

The present document reports the second iteration of the Field Trials and Evaluation carried out in task T5.4 of STREAMLINE work plan. Final reports on Design and Implementation activities will be reported in deliverable D5.5 (M33), and Field Trials and Evaluation activities will be reported in deliverable D5.6 (M36).

During the initial phases of STREAMLINE, ALB emphasis was put on the underlying process of designing and testing an embryonic IPTV recommender system and identify/evaluate how to integrate STREAMLINE proposed technologies. From v1 to v2, main evolutions were

- Previously identified requirements were consolidated and some additional effort has used in detailing analytics and profiling requirements
- Global Architecture processing steps were updated and new processes based on Apache Flink and Apache Zeppelin were introduced
- Existing data sources were updated and new data sources were introduced; data quality was thoroughly tested
- A STREAMLINE based Recommender engine was implemented and its initial accuracy was tested

One major technical issue was identified in ALB use case – some critical data quality issues were identified and reported to MEO for fixing (see section 2.3.1). Until those fixes are available, we will have some impacts on the implementation, which is contributing to some delays in the implementation and field tests.

The adequacy of the implemented recommender engine was initially validated. It will require some evolutions which will rely on testing on new and larger datasets.

Some relevant issues were identified and will be assessed in v3, namely

- evaluate the adequacy of a research context based solution
- implement a model for live collaborative filtering recommendation and evaluation

In the second year of STREAMLINE Rovio has focused more on strengthening its internal analytics offering utilising open source tools such as Apache Flink, Spark and Presto. The main use case continues to be about player profiling and reporting, but instead of trying to build a separate streaming solution based on Apache Flink to co-exist with our standard pipeline Rovio wants to build a hybrid solution where Flink based modules are integrated into existing solutions where applicable. The strategy is to have open datasets (columnar data for batch and Kafka topics for streaming) and then use the best tool for the job. Rovio also continues to evaluate different 3rd party service providers and Apache Flink is utilised to build real-time pipelines to efficiently integrate into the systems without the need for client side SDKs.

During 2nd year Rovio has reached its high level STREAMLINE KPIs defined in the proposal by having multiple new use cases implemented and deployed with maximum latency of business KPIs under one hour. On technical side Apache Flink based services with STREAMLINE features were stable and Rovio was able to implement required features efficiently. In future development, Rovio plans to focus on 3rd party system integration and complementing our analytics pipelines with streaming use cases.

During the second year, IMR has revised the implementation of its main classification workflow to enable a full streaming approach. Data exchanges between the components (crawler and Haddop),
initially based on multiple serialization steps, has been replaced by a pipelined architecture, where Kafka handles the flow of page streams supplied by the crawler, and Flink consumes Kafka input with its streaming interface. This new architecture is currently under evaluation and should address one of our main objectives, namely the elimination of the important latency in data acquisition. Year 3 should be fully devoted to the introduction and evaluation of the PassiveAggressive classification method in Streamline Flink, and its integration in the categorization workflow.