## Contents

1 Introduction ............................................. 3

2 Requirements ........................................... 3
   2.1 Non-Functional Requirements ...................... 4
   2.2 Functional Requirements ........................... 4

3 System Architecture .................................... 5

4 Technical Design ....................................... 7

5 Decisions ................................................. 9
   5.1 Language and Libraries ............................. 9
   5.2 Software ........................................ 10
   5.3 Driver ........................................ 10

6 Results .................................................. 10
   6.1 Unimplemented Features ......................... 10
   6.2 Performance Testing .............................. 11
   6.3 Summary of Testing ............................... 11
   6.4 Future Work/Extensions ........................... 12
   6.5 Lessons Learned ................................... 12

7 Appendix ............................................... 14
1 Introduction

Comcast Corporation is a global telecommunications conglomerate and one of the largest and most prominent broadcasting and cable television companies in the world. Comcast is also the largest provider of cable internet access in the United States with over 25 million high-speed internet customers as of 2017.

In their online streaming solution, Xfinity On Demand, consumers can watch movies and TV shows in video players. Each time these players are accessed on an individual device, a new “session” is started. Throughout the session, data is sent back to Comcast regarding the health and lifespan of the player. This includes bit rate changes, frame rate changes, error messages and failures, stream consistency and duration, startup latencies, and geographical location. These AnalyticsEvents are analyzed to continually eliminate potential errors, create statistics, and forward billing information on to other carriers and companies.

At any given time, Comcast is receiving AnalyticsEvents from tens of thousands of devices across the US. Clearly, this is an immense amount of data which needs to be ordered in some way if it is to be useful. The most relevant way to group the data is by the session from which it was created. This process would effectively give Comcast the ability to “replay” a user session.

The task assigned to the CSM field session team was to develop an application which takes in large amounts of AnalyticsEvents and outputs grouped sessions or PlaybackSessionEvents. Ideally, Comcast would use this application to more effectively group and analyze sessions in real-time; however, the amount of data received per day is greater than what can currently be processed. In order to prevent data loss, AnalyticsEvents are stored in Amazon S3 buckets, which allows data to be processed any time after it has been received. So, our application utilizes this “at rest” data as input and implement a solution to more efficiently “sessionize” player data.

2 Requirements

The definition of done for the project is to produce an Apache Spark application written in Scala that reads AnalyticsEvents from an Apache Parquet file, groups the AnalyticsEvents by the session from which they were created, transforms the grouped data into PlaybackSessionEvents, and writes the PlaybackSessionEvents to another Parquet file. The non-functional requirements listed below show the different software tools needed to fulfill the technical design. The functional requirements break down the definition of done into specific tasks necessary for completion of the application.
2.1 Non-Functional Requirements

- Apache Spark application
  - Unified analytics engine for large-scale data processing
- Written in Scala
  - Combined functional and object-oriented language
- Utilizes Apache Parquet for data storage
  - Column-oriented; optimized for compression and scanning efficiency
- Developed using the IntelliJ IDE
- Github for version control and stakeholder accessibility
- Use Trello for backlog and project tracking

2.2 Functional Requirements

- Read in and process large Parquet files containing multiple hours worth of player events, which are called AnalyticsEvents.
- Group AnalyticsEvents by session information. Sessions are identifiable by the session field in an AnalyticsEvent. The session field is a struct that contains two values: the pluginSessionId (PSI) and the playbackId (PBI). A unique combination of these two subfields indicates a single session.
- Gather all other necessary information from the grouped AnalyticsEvents to fully populate the PlaybackSessionEvent schema. This includes:
  - header - contains timestamp and universally unique ID (UUID) of the event
  - partner and partnerId
  - startTime
  - eventType
  - completionStatus
  - sessionDuration
  - device
  - customerAccount
  - application
• Create a map of all the AnalyticsEvents of a single session.
  – Key-value-pairs are the event UUID and the event itself
• Order all necessary information such that it fits the PlaybackSessionEvent schema
• Write all produced PlaybackSessionEvents to an output Parquet file
• Runtime of application must be, at a maximum, half the amount of data processed
  – i.e. two hours worth of AnalyticsEvents must be processed in, at most, an hour
• Test cases to validate output data-frame of PlaybackSessionEvents, specifically the map of all events from a single session

3 System Architecture

The Figures below illustrate the architecture and system design of the sessionization application. Figure 1 represents the flow of data, beginning with the creation of AnalyticsEvents and ending with storage of PlaybackSessionEvents.
The IP Video Data Producers represent consumers accessing an Xfinity On Demand video player on their phone, laptop, or other portable device. As consumers watch videos, the player continuously creates AnalyticsEvents and sends them back to Comcast. Comcast then stores the events in Amazon S3 buckets. Several hours worth of this data was retrieved at the beginning of the project and stored in Parquet files to be used as input for the sessionization application. After processing, the newly created PlaybackSessionEvents are again stored in Parquet files to be used by Comcast.

Figure 2 represents the architecture of Apache Spark, as it is the main driver for our application. Because we are working with big data, a single machine cannot provide the necessary processing power to make the application efficient. To account for this, Spark transfers partitioned data to multiple machines or servers, called worker nodes, and executes the same task in parallel. The worker nodes are effectively sent individual tasks by the cluster manager, and, once the task is completed, send them back to the manager. Thus, the cluster manager acts as the interface between the driver and the worker nodes.

![Spark Data Distribution](image.png)

In the context of our application, Spark is told to partition AnalyticsEvents based on their session. This means that each worker node is processing and subsequently returning data from a single session, allowing for an approximately even distribution of work between executors. With roughly the same amount of data being processed on each node, and each node producing a completed output, the application achieves the most efficient runtime possible.
4 Technical Design

The most technically interesting aspect of our application is the function that groups AnalyticsEvents by session and gathers all the necessary information to populate PlaybackSessionEvents. The full schemas for both events are shown in Figures 3 and 4.

Figure 3: AnalyticsEvent Full Schema

Figure 4: PlaybackSessionEvent Full Schema
Much of the information contained in the AnalyticsEvent schema is also present in the PlaybackSessionEvent schema; however, there are several key fields that need a more in-depth explanation. The most important field is session. As previously stated, this is the field on which AnalyticsEvents are grouped. A unique combination of both PSI and PBI is what indicates an individual session, so both are taken into account in the grouping function.

The second most important field is event. As seen in the schemas, this field contains all the information regarding the health of the player. Though the event field has many subfields, each AnalyticsEvent only has one or a few of these subfields populated based on what kind of event it is. If the AnalyticsEvent is a heartbeat, it will only have the heartbeat subfield populated; if it is a frame rate change, it will only have the frameRateChanged subfield populated. When grouped, all the event information from every AnalyticsEvents in that session is collected in a map, and this is what populates the events field in the PlaybackSessionEvent.

Another significant field is the header. In an AnalyticsEvent, this struct contains the timestamp of the event (when the event was created), the UUID of the event, and a couple other less important subfields. Whenever an AnalyticsEvent with a new combination of PSI/PBI is received, this indicates that a new session has been started. So, the timestamp and the rest of the header information from the very first AnalyticsEvent is used as the header for the corresponding PlaybackSessionEvent.

Similarly, the startTime and sessionDuration fields in the PlaybackSessionEvent are populated using the timestamp information from the AnalyticsEvent headers. startTime is the timestamp taken from the very first event, and sessionDuration is calculated by subtracting the timestamp of the first event from the timestamp of the last event.

Additionally, as deemed necessary by Comcast for our application, the eventType, completionStatus, and customerAccount fields are created and, for the most part, set as constant values for all PlaybackSessionEvents.

Aside from those listed above, all other fields are copied directly from the first AnalyticsEvent to its PlaybackSessionEvent. The aggregation of all fields in the grouping function can be seen in Figure 5 below.
5 Decisions

The following sections describe the main technical design decisions of the application with a rationale for each. In general, the team was focused on finding the best way to implement functional code while matching Comcast’s provided event schemas.

5.1 Language and Libraries

To reiterate, the primary language used was Scala, which is a combined functional and object-oriented programming language. Scala was implemented in combination with Apache Spark to create a productive and robust system to analyze and process big data. Spark’s ability to partition data, along with its support libraries for SQL and data-frames, allows data to be processed on multiple executor nodes across multiple servers. This functionality is crucial for the large data files we used, as a single system cannot handle the vast amount of data. As aforementioned, each executor processes AnalyticsEvents from a single session. This distributes the work as evenly as possible and minimizes the overall runtime. Additionally, Comcast’s real-time data processing program is written in Scala and Spark, so our sessionization project followed suit so it can be integrable.
The storage format for the application data is Apache Parquet, which is a column-oriented database management system. This format was chosen for its excellence in compression and scanning efficiency. Additionally, Spark has reliable support for both reading from and writing to Parquet files.

5.2 Software

The Comcast team guiding this project uses IntelliJ as their primary IDE for Scala development, so we decided to use it instead of Eclipse. Though we were all already familiar with Eclipse, we concluded it would be better to learn IntelliJ as we would likely use it in the future, and it allowed us to standardize our development environment with Comcast’s.

A private GitHub repository was used for version control. This allowed for seamless code sharing amongst the team and instant delivery to Comcast.

5.3 Driver

As previously stated, Spark’s ability to process data on multiple nodes is paramount if the application is to be efficient and viable for Comcast. In accordance with this end, the code could not invoke functions that collect data onto the local machine. As seen (or perhaps not seen) in Figure 5 in Technical Design, the code is very limited in the built-in functionality it uses.

6 Results

6.1 Unimplemented Features

Our team did not leave any required features unimplemented, and we fulfilled Comcast’s definition of done. Though all required features were implemented, there are a couple aspects that do not work exactly as intended.

First, our team struggled to accurately generate the map of event UUIDs to events in the PlaybackSessionEvent schema. Instead of a single map of key-value-pairs, our code generates an array of maps with a single key-value-pair each. Additionally, one of the subfields in the customerAccount field in the PlaybackSessionEvent schema needed to be a randomly generated string, different for each session. While our code can generate a random string and populate the subfield for every session, it is not unique for every session. Both of these situations have been communicated with and acknowledged by Comcast.
6.2 Performance Testing

Below is a list of the passing test cases implemented as part of the application to ensure that the data manipulation is accurate.

- Parquet File Test
  - This test reads in a Parquet file and writes to a new one. This verifies that our code for reading and writing data-frames to Parquet files works properly.

- Sessions Created Equals Sessions Expected
  - The purpose of this test is to verify that the number of PlaybackSessionEvents in the output data-frame matches the number of unique PSI/PBI combinations from the input data-frame.

- Events Per Map
  - This test ensures that every AnalyticsEvent in the input data-frame is uniquely mapped to a PlaybackSessionEvent and that no events are lost.

- Processing Factor
  - This test verifies that the data for each session is being processed in a time less than half the duration of the session.

6.3 Summary of Testing

Figure 6 below displays the performance results of processing runtime. The application appears to exhibit an approximately logarithmic complexity; however, after 100,000 events, the run time appears to increase at a linear rate. Though it is difficult to determine the exact complexity of the application without knowing the complexity of specific Spark functions, it is estimated, from Figure 6, to be between $O(\log n)$ and $O(n)$. These tests were performed on single machines, so performance would improve if the application was ran on a dedicated cluster of machines. A table with specific runtimes can be found in Appendix A.
6.4 Future Work/Extensions

One idea for extending the application would be to create an HTML diagnostics page. This would be a document generated per some interval of time during processing, containing a high-level overview of the data for a human to review. This would include information such as the frequency of events, average stream duration, average frame rate, frequency of stream failures, the overall number of events processed, and average number of events per session.

Due to our limited access to Comcast’s codebase and Comcast’s direction throughout this project, our final application is more functional rather than object-oriented. In context, this means that instead of reading in the input data-frame, creating objects for each AnalyticsEvent, and then creating PlaybackSessionEvent objects, we simply transform the AnalyticsEvents into PlaybackSessionEvents. A future extension, if we were given access to certain class definitions, would be a more object-oriented approach, allowing for more direct manipulation of data but possibly a longer runtime.

6.5 Lessons Learned

Scala is similar to Java in that it can be object-oriented, but it is also unique as it is functional language. The object-oriented nature made learning the basic syntax
fairly easy, but implementation of Spark functionality was significantly more
difficult, taking a lengthy amount of time to Figure out. This will likely be the case
in the future when developing with new frameworks, and we will need to allocate a
good chunk of time to account for learning curves.

Spark is an extremely powerful tool for processing big data, and since it can be
written in multiple languages, it is quite versatile. We wrote our application in
Scala to be compatible with Comcast’s systems, but writing it in Python, a
language with which we have prior experience, may have allowed for faster
development and further functionality. On the other hand, having prior experience
with SQL made working with data-frames and Spark’s version of SQL easier, as we
could visualize and more fully understand how we were manipulating data. So,
efficient development is best achieved by finding a balance between learning new
functionality and drawing from prior experience.
7 Appendix

<table>
<thead>
<tr>
<th>Number of Events</th>
<th>100</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>50000</th>
<th>100000</th>
<th>250000</th>
<th>480320</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial 1 Time (s)</td>
<td>34.685</td>
<td>74.476</td>
<td>107.424</td>
<td>140.214</td>
<td>253.478</td>
<td>394.239</td>
<td>419.053</td>
<td>547.01</td>
</tr>
<tr>
<td>Trial 2 Time (s)</td>
<td>31.646</td>
<td>74.976</td>
<td>110.34</td>
<td>131.566</td>
<td>284.241</td>
<td>402.787</td>
<td>457.564</td>
<td>665.682</td>
</tr>
<tr>
<td>Trial 3 Time (s)</td>
<td>34.178</td>
<td>75.195</td>
<td>108.11</td>
<td>151.537</td>
<td>350.206</td>
<td>410.623</td>
<td>411.361</td>
<td>476.096</td>
</tr>
<tr>
<td>Trial 4 Time (s)</td>
<td>36.979</td>
<td>84.617</td>
<td>108.536</td>
<td>166.994</td>
<td>278.276</td>
<td>400.677</td>
<td>567.917</td>
<td>479.226</td>
</tr>
<tr>
<td>Average (s)</td>
<td>34.372</td>
<td>77.3160</td>
<td>108.6025</td>
<td>147.5778</td>
<td>291.5503</td>
<td>402.0815</td>
<td>464.1238</td>
<td>542.0035</td>
</tr>
<tr>
<td>Average/Event (s)</td>
<td>0.3437</td>
<td>0.0773</td>
<td>0.0217</td>
<td>0.0148</td>
<td>0.0058</td>
<td>0.0040</td>
<td>0.0019</td>
<td>0.0011</td>
</tr>
</tbody>
</table>

Figure A: Process Performance Data