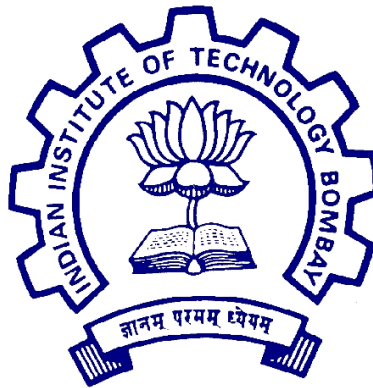


Data Analytics - Detection of difficulty areas in videos based on student viewing activity

Summer Internship 2015

Submitted in fulfilment of internship Project



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

By
Arinjoy Basak, Data Analytics Group

Project Mentor and Guide: Sukla Nag
Principal investigator: Dr. D. B. Phatak

Last Updated: July 2, 2015

Acknowledgement

I would like to take this opportunity to thank all of the people without whose assistance and influence, this project would not have been completed.

I would like to thank Mrs. Sukla Nag for giving me an opportunity to work on this project, and for all her guidance, assistance and help in making this project possible. She has been a constant source of support and expert guidance through the many situations that have occurred during this project.

Contents

1	Detection of difficulty areas in videos based on student activity	1
1.1	Introduction and outline of the chapter	1
1.2	Discussing the problem statement	1
1.3	Basic outline of the idea	2
1.4	The Project Flow	3
1.4.1	Source of data for the model	3
1.4.2	Processing the data	4
1.4.2.1	Extracting and Cleaning the required data	4
1.4.2.2	Video length extraction using Google YouTube API	6
1.4.2.3	Redundancy removal and Error correction	8
1.4.3	Creation and prototyping of the data module	9
1.4.4	Final Implementation and visualization	15
1.4.4.1	Extracting the data from Hive tables	15
1.4.4.2	Filtering of events per user and per video	16
1.4.4.3	Processing and generation of results for final summary table	17
1.5	Future Work	19
1.6	Technologies Used	19

Chapter 1

Detection of difficulty areas in videos based on student activity

1.1 Introduction and outline of the chapter

This part of the project report talks about the project work done regarding the development of data models for the detection of difficulty regions in videos based on the students behaviour, recorded through the log events in IITBombayX. The chapter is outlined as follows. First, we introduce the problem statement that we wish to address through our work, and provide a basic outline of our idea to deal with the problem. Then, we talk about the steps that we have followed in order to create and formulate the data model. Finally, we describe how we had designed the data model and the functionality both at the processing and visualization level for a final implementation of the analytics module in the scenario of Big Data. After the final implementation and testing, the module was integrated with the OPENedX Insight analytics system for IITBombayX, and made available for use by the course instructors participating in the Blended MOOCs model. The entire work of this part was completed in 3 weeks of the internship.

1.2 Discussing the problem statement

IITBombayX provides a number of Massively Open Online Courses (MOOCs) to be accessed by students from students all over India, irrespective of any barriers whatsoever. For the students participating in MOOCs, from the point of view of the course instructor, the primary mode imparting the lessons to the students, and the basic mode of communicating the knowledge to the students is through the lecture videos that the instructor publishes as part of the course. The videos are distributed among the various weeks and lessons in the courses, with each lesson having possibly multiple videos. Ideally these videos are watched by the students in succession, as the course progresses, and the student's learning is successively evaluated through the tests and questions he has to complete online. Occasionally, a student may diverge from the normal train of events and watch the videos at his own pace, and out of order, but that is not our primary concern here.

Even though the student watches the videos online as part of the course, the student may (and they often do) face problems and difficulties in understanding certain concepts, much like in a classroom environment where the teachers deliver the lectures. In contrast to the classroom environment, here, the student can go back to a part of the lecture video over and over again, and watch and note the concerned part repeatedly until his or her doubt in the particular topic is cleared. While this is quite common, we have to note the causes for such occurrences. Among myriad other reasons why this could happen, we believe that the reason why students would have to repeatedly watch a certain section of the video is because:

- The course videos are difficult for the students to understand.
- The course videos do not provide sufficient clarity on the subject that it deals with.

Now, student behaviour such as this (namely, repeatedly watching certain sections of the videos) is an information of interest to the course instructors managing and designing a particular course, as

they would always be willing to improve upon the course and help the students in their learning process. If this data were made available to the instructor in an understandable interface, then the instructor would be able to tell whether the participants in his MOOC are indeed finding the videos in the course useful for their purpose, or are simply spending their time perplexed over certain regions of the videos and are thus not being able to utilize the learning advantages of the MOOC system - missing the forest for the trees, as it were, and ending up learning little, none, or worse - going back with mistaken understandings regarding the subject of the course. Thus, we understand how important it is in order to address this issue - and we intend to solve this through data analytics tools and data modelling.

1.3 Basic outline of the idea

The solution that we came up with was based on the fact that the OPENedX InSight platform, on which IITBombayX is based, records all the student activities on the platform as the logs. In other words, everything that the students do on the platform while accessing and interacting with the course content - including video actions, are recorded by OPENedX, and made available to all those who need to use it. We can also identify not only the users responsible for an activity, but also the particular course module (problem, question, page or video) that the user interacts with. In this way, we can also track the behaviour and activities of a student with respect to a particular video in a course, and the various video events such as pausing a video, playing it, seeking to a certain part in it, and so on. These events can then be analysed to determine meaning and inferences regarding the behaviour of a student on a particular video.

Primarily, we could focus on only the events such as pausing a video, playing a video, or seeking back and forth on a certain region of the video, and collect the locations of the pauses and seeks in the video to determine how much of the video has been watched by the user, and how have the different parts of the video been watched by the user. Once these events are collected, and the data processed, appropriate visualizations - such as graphs, charts, or some other means - could be developed on this data, or tools and alerting systems developed in order to tell the instructors of the course about the different regions in the video where the students have indeed faced difficulties. An alerting system would require the system itself to be able to compare the data of different videos within the same course, undertaken by the same instructor, to be compared to each other and then determine from among them which are the regions of the videos where the students are facing difficulty in understanding.

Once such regions of the videos have been detected and the alerts issued or visualizations provided - the instructor, knowing the subject matter that corresponds to the regions of difficulty, would then take appropriate measures on his side to make the course contents more lucid for the students participating in his course. Solutions to this problem by the instructor could range from the following:

- Adding more explanatory material, such as text materials, links, PDFs, and so on - or similar materials,
- Release of an additional video explaining the matter in a more lucid fashion for the students; covering materials which would enhance the understanding of particular subject matter by the students; or even the re-release of the same video, with edits and additional content for explaining the subject relating to regions of difficulty to the students,
- More quizzes and practice exercises, so that the students may be able to learn better by progressing through the matter using practical examples and hands-on experiences of the problems, answering questions, and going through corrections based on evaluations of their performances,

And so on and so forth, as the instructor may see fit for the situation.

Our project mainly aims to solve the problem mentioned in the previous section through the development of a data analytics module as a part of the IITBombayX InSight data analytics system. The broad functioning of the system would be as follows. The system would periodically go through the log data generated for the platform for every single video, assess the behavioural trends and activities of the students, and process it to determine the difficult areas in the corresponding videos on the basis of different parameters such as the amount of views for the different regions of the videos, time spent, etc. Once the processing of the data is completed, we would have the analysed summary data which

could then be used for drawing inferences, learning, or visualizations for the analytics dashboards. One approach is to directly visualize the data on the InSight dashboard interface provided to the instructor for a course using the summary data for a video directly. Alternatively, we could determine the regions of difficulty in the videos and compare these performances over all of the videos, and determine a trend of watching the videos; based on which, it would be able to tell, or classify whether a video has been deemed by the students as too difficult or just perfect for the students. In the current project however, we have focused chiefly on the process of generating the summary data, starting from the log data, and visualizing it at the end on the OPENedX InSight dashboard, allowing the instructor to make intuitive decisions based on the graphs plotted based on the data showing the metrics developed on the video events.

1.4 The Project Flow

In this section of the report, the details of the work done to achieve the final application are described. Starting with the raw data in the form of the log files, a considerable amount of effort has gone into several steps involved that transform the raw, uncleaned and unintelligible data into cleaned, well formatted data, to processed summary data and finally, its visualization on the dashboard. The steps can be clearly outlined as follows:

- **Extracting the data:** This step involves the extraction of the data from the huge amounts of the log data that is generated by the IITBombayX platform every day, periodically, followed by converting it into a format and saving it on a store which can be regularly and efficiently accessed for processing by the further stages of the applications.
- **Processing the data:** This step involves the accessing of the required data from the saved stores in an efficient manner and properly process it and clean it prior to feeding it to the data model. It will be explained later in detail, but it so happens that even after retrieving the data from the stores, it is required to do certain processing on the data before it can be given to the data model for generating the relevant feature data from the events.
- **Creation and prototyping of the data model:** Following the extraction and processing of the data, a data model is used for extracting the feature data that we would be using for our summaries. In this stage, a working prototype of the data model was developed to work on a sample of real-world data, so that it could be developed, revised and corrected as fast as possible, and then taken to the final implementation. The prototype development stage involved the use of MySQL and Python scripts and libraries. the extraction of the required data was done from MySQL tables using carefully crafted SQL queries. The data cleaning and processing code, and the code to implement the prototype of the data model to generate summary data, was done using a Python program.
- **Final implementation of the data model:** The working prototype of the model was then scaled up and coded for frameworks to work in a Big Data environment, where the data model and the processing tasks would be working on log data generated in a scale of potentially tens of gigabytes per day. This stage involved the final implementation of the data model and its deployment in the IITBombayX InSights dashboard, including appropriate visualizations to aid the monitoring and decision making process of the course instructors. The Big Data technologies used to achieve the final implementation include the Hadoop Distributed FileSystem, Hive data warehouses (which use Hadoop for storing the databases and tables created and accessed in Hive), and SparkSQL to provide a fast and very efficient method process the massive amounts of data using a MapReduce workflow and parallelized functions. A visualization for the final summary data, which determines the total number of accesses and total time spent in the different regions of the video by the people who have seen the video, is made available on the front end — which use Django framework, and R visualization scripts using GoogleVis libraries.

1.4.1 Source of data for the model

To begin with, the prototyping of the data model or even the creation of the basic application required for us to have some real data to work on, giving us the details of the student's activities on the IITBombayX platform. Now, the OPENedX platform generates the log data regarding each and every student activity, and records it in log files, which are stored on the servers. Typically, these log files span over several

gigabytes in size, and therefore, any processing on these would require application of Big Data tools. The individual logs themselves are recorded as JSON strings. A JSON string consists of several key-value pairs, and a single key itself may lead to a nested structure having more key-value pairs. A sample of the log event looks as shown 1.1.:

```
{
  "username": "arinjoy15",
  "host": "10.105.25.74",
  "event_source": "server",
  "event_type": "/dashboard",
  "context": {
    "user_id": 11,
    "org_id": "",
    "course_id": "",
    "path": "/dashboard"
  },
  "time": "2015-06-12T10:20:08.703318+00:00",
  "ip": "10.105.1.7",
  "event": "{ \"POST\": {}, \"GET\": {} }",
  "agent": "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0",
  "page": null
}
```

Figure 1.1: A sample log event generated .

Now, the accessing the logs themselves in such a manner is a difficult and tedious task. The structures of the JSON strings is not the same for all events, and while all events have certain common keys, they also have certain key-value pairs that are descriptive of the particular events themselves. For example, a log event describing a video event would have keys for the users, time of event, event-source, and so on - so would an event that corresponds to a student-dashboard access. However, the video event would also contain details of the video event, the video id, the current location of play or pause, and the current video speed; which would not be there in a dashboard access event. Thus, we explain the difficulty of accessing these events in their raw form.

To solve this problem, the log events were flattened out, to create tuples having fields that would have data according to the type of the event in question. So, fields that do not relate to the data for a particular event would be left NULL, while the other fields related to the same would be having valid data. These flattened out logs were then put in the MySQL table corresponding to the User logs in the Data Analytics database, so that they could be accessed and queried as required through simple MySQL queries. This data served as the source data for my task. There were certain errors and discrepancies in the database tables created from the logs, which are discussed in the subsection on 'Extracting and Cleaning' the data.

1.4.2 Processing the data

In order for us to determine the model on which we were to base our system, we drew our data from the IITBombayX log files, which were cleaned and made appropriately accessible through MySQL queries. Now prior to actual processing of the log data for the videos and with respect to the user activities, the data needs to under go further extraction, cleaning and pre-processing - as the log data also contained several redundancies and errors. All of these tasks were done at this stage of the project. This step also required us to understand the nature of the logs at a fundamental level, since in the following stages, we would have to design our data models to work on the processed and filtered events. A description of the activities undertaken in this stage are in the following subsections.

1.4.2.1 Extracting and Cleaning the required data

A first step involved exploration of the log events and the associated attributes of the logs, to identify the features that we would require to identify each video element for each course, each video event, and to extract the data. These are the features that we would later use for the creation and framing of our data model. At this stage, SQL queries were used to filter and select the relevant data for each video and each user watching it. The tables that we used in this stage for our queries included the following:

- **UserSessionOldLog** : The MySQL table which contained the events from the user logs put into a single table, in a flattened format. The table had a total of 58 columns, and contained nearly 4 million entries. However, upon exploration of the values of the attribute using specialized queries, the attributes we found to be of interest to us were as follows:
 - **userName** : The name of the user who is responsible for the event in the user log,

- moduleSysName : The system id of the module (in this case, usually the video) which is being used or accessed by the user concerned.
 - courseName : The name of the course in which the module is located. This allows us to detect the videos corresponding to a particular course while creating summary data for a video.
 - currVideoTime : The current location of the video when the log event was recorded.
 - oldVideoTime : The old location of the video when the log was recorded (required for certain video events such as seeking),
 - createDateTime : The system (real-world) time when the log event was created by the system - allowing us to arrange the events in a chronological fashion to determine the temporal sequences and semantics of the events generated.
- CourseVideos : This is the MySQL table which contained all the details of all the videos in the different courses that were available on the IITBombayX platform courses. The table itself contained 14 attributes, and contained 1628 entries. Upon exploration of this MySQL table too, we found that there were only certain attributes of interest to us, that were as follows:
 - videoSysId : Similar to UserSessionOldLog.moduleSysId, this is also a system id which is used to determine the individual video modules that are begin referenced in each event or even each activity.
 - videoUTubeId : This field is a string field containing the YouTube id of the corresponding lecture video whose videoSysId is provided; the videos are actually published on Youtube, and a link to these videos is provided in the corresponding units and sections in the actual course.
 - videolength (added) : This is an additional field which was computed and added to the fields later by work done as part of this project. This field would contain the floating-point values indicating the length of the YouTube videos (i.e., the lecture videos) obtained from the YouTube attributes of the videos. The process of extracting this data is elaborated in a later section entitled 'Video length extraction using Google YouTube API'.

One point that needs to be noted here is that while the code had been written to flatten out and clean the log JSON strings, it was not complete in extracting all the log information correctly from the logs. A number of erroneous situations were encountered while going through the tables UserSessionOldLog and CourseVideos. For example:

- There was one CourseVideos entry (id='2017625218ba4309b4cd42309f5d82e2') which did not have an YouTube id, chapter title and which meant that no information about the video could be extracted from the YouTube API even if we wanted to.
- In certain cases, the events such as save_user_state were supposed to record values in the fields for oldVideoTime and currVideoTime, but instead, they showed up to be NULL. This may have happened in the log files due to a network error, due to which erroneous data was recorded in the files, or due other factors beyond our control at this point.
- For all the save_user_state events, despite clearly being video events associated with a definitely identifiable video, the attribute moduleSysId did not contain the videoSysId of the corresponding video in CourseVideos. It was NULL for such cases, making it impossible to identify which video the event was associated with. This was a problem at the time of working on the project, but which was subsequently fixed in the course of the project.
- The sequence of events that was observed on practice simulations of the platform was not necessarily observed in the log data collected for the events. In many cases, the events were recorded out of order — and hence required inference rules to associate them correctly to avoid errors. Again, this could be due to network errors, or some erroneous situations or faults that cannot be controlled or monitored by us at this point.

All of these errors, nevertheless, were dealt with using appropriate generalization cases and writing the codes and conditions for them accordingly. These are discussed later in these section, under 'Redundancy removal and Error correction'.

Now, in order to start working on the prototype, we needed to find the most efficient and complete queries that would extract all the necessary data that we would require for our data model. As far as we were concerned, we simply needed to select for a video, and for a user who had watched the video, to get all the events of that user accessing that video. This could then be processed upon by our programs.

To start with and develop our data model and our queries, we first focused our search on videos belonging only to the CS101.1x course. This course alone had 155 videos, out of the nearly 1600 videos of all the courses that are currently provided on the platform. In order to further restrict our search for designing the model, we selected the video with the most number of views, based on the log data, and then selected the user who has accessed that video the most number of times. The end result of this process was a sufficient amount of log data for a single user viewing a single video of a particular course, which would provide enough insight into the patterns of general behaviour of a student or learner.

Some of the MySQL queries that we actually used for our data model prototype are as follows:

- Fetching top 50 most watched videos in CS101.1x

```
select t.*, CV.videoUTubeId, CV.videolength
from (
select moduleSysName, count(eventName) eventCount
from UserSessionOldLog
where courseName='CS101.1x'
and moduleSysName is not NULL
and eventType='video'
group by moduleSysName order by eventCount desc limit 50
) t,
(
select * from CourseVideos where courseName='CS101.1x'
) CV
where t.moduleSysName = CV.videoSysName
order by eventCount desc;
```

- Highest watched video : videoSysId '67a8559582864d6a8148e2ef5c997e8f'; So, number of viewers in descending order of activity were found as follows:

```
select userName, count(eventName) watched
from UserSessionOldLog
where courseName='CS101.1x' and
moduleSysName='67a8559582864d6a8148e2ef5c997e8f'
group by userName order by watched desc limit 10;
\end{tiny}
```

- Finally, for a given user, we found out the distinct events of interest giving us the behaviour of the user in the following manner (say, for the same video as before, but userName='ricky'):

```
select distinct eventType, eventName, moduleSysName, eventSource,
oldVideoTime, currVideoTime, createDateTime
from UserSessionOldLog
where userName='ricky' and (eventType in ('video')
and moduleSysName='67a8559582864d6a8148e2ef5c997e8f'
or eventType in ('video', 'navigation') and
eventName in ('pageclose', 'saveuserstate')) order by createDateTime;
```

1.4.2.2 Video length extraction using Google YouTube API

In order to determine a way to measure the relative amount of time spent by the student on a particular video (such as the fraction of the video playing time), we needed to have an idea of how long the video actually is. A student may spend the same amount of time on two different videos, but the different

lengths of the videos in relation to the time spent on the videos would be a clear indication of actually how much attention and dedicated effort the student is providing to the video in the course. A higher percentage of time would mean a more engrossed attention, while a lower percentage would probably mean a casual learner.

However, this data regarding the duration of each of the YouTube videos was not available to us in the first place, as the table CourseVideos did not contain any field to determine the length of the videos. So, an additional work was done to extract the actual duration of the videos on the all courses using the YouTube id's available for the videos in the CourseVideos table (i.e., the videoUTubeId attribute). The lengths of the videos extracted from the JSON style strings returned by the Google APIs were processed into float values, and the result stored in a new column in the tables, called 'videolength', through a Python script.

It should be noted here that the Google API requires an authorisation key to be supplied by the developer or the program in order to access the API. For the script used, I have hard-coded a developer key I obtained from the service, and it can be used effectively, subject to certain daily quota restrictions (50,000,000 calls per day, 3000 calls/second/user).

The Google YouTube API basically sends GET requests online to the API together with the Youtube video id and the fields of the 'video' element that it has to return. For our case, we required only the contentDetails attribute of the video element, in which, the value corresponding to the key 'duration' gave us the duration of the video in a ISO 8601 format (PT59M59S). This was processed and converted to a float value, which was then stored in the MySQL table through the script using a MySQL connector for Python.

The following is the code for the same:

```
#Proxy settings - change accordingly
import os
os.environ['http_proxy']='http://arinjoy15:arinjoy15*@proxy.cse.iitb.ac.in:80'
os.environ['https_proxy']='http://arinjoy15:arinjoy15*@proxy.cse.iitb.ac.in:80'

from apiclient.discovery import build
from apiclient.errors import HttpError
import mysql.connector

#creating connection and obtaining cursor
cnx = mysql.connector.connect(user='root', password='',
                             host='127.0.0.1', database='IITBxDataAnalytics')
cursor = cnx.cursor()

#accessing list of video id's from the table.
query = ("SELECT videoUTubeId FROM CourseVideos WHERE videoUTubeId IS NOT NULL")

cursor.execute(query);
videoslist=dict()

for video in cursor:
    videoslist.setdefault(str(video[0]),0.0)

#creating Youtube API service through the key
api_key = INSERT_OWN_DEVELOPER'S_KEY_HERE
youtube = build("youtube", "v3", developerKey=api_key)

#processing for each video
for video in videoslist.keys():
    timestring = str(youtube.videos().list(id=video, part="contentDetails,snippet")
                     .execute().get("items",[])[0]["contentDetails"]["duration"]).split("PT")[1]
```

```

timestring = timestring.replace("H",":").replace("M",":").replace("S",":")
timestring = timestring.split(":")

second=0
minute=0

if not timestring[1]=="":
    second = int(timestring[1])
    minute = int(timestring[0])
    videoslist[video] = minute*60.0 + second

#updating the table columns
for videoName in videoslist.keys():
    query = ( "UPDATE CourseVideos SET videolength=%s WHERE videoUTubeId=%s" )#to change to include the
    data = ( str(videoslist[videoName]), str(videoName) )
    cursor.execute(query, data)

cnx.commit()
#closing the cursor
cursor.close()
cnx.close()

```

It should be noted at this stage that in the table for the course videos, there were certain errors in the details of the videos recorded. In particular, the video with the id '2017625218ba4309b4cd42309f5d82e2' had neither youtube id nor a title name. Hence this video had no duration calculated from it, as it was evicted from the result set by the IS NOT NULL condition in the WHERE clause.

1.4.2.3 Redundancy removal and Error correction

In this stage of processing the data prior to applying to the data model, we worked on removing the errors, redundancies and reordering of events in the logs. This was necessary because based on the log data that we received, even if they were flattened out, the simple SQL queries would not be sufficient enough to get the precisely clean and concise data that we would require for the data model. There were many instances of the same events that were recorded multiple times at the same instant, several instances of the same event recorded over consecutive times, erroneous occurrences of events recorded at the wrong times, and several other errors. So, we approached this problem as follows: From the list of events, keep only those events that corresponded well with the corresponding video watched by the user, and eliminate the rest of the events as they were redundant or wrong, or just useless in our current context. Doing the operations specified in the following lines not only removed the wrong and unnecessary events, but also increased processing time in view of the fact that the number of distinct events that needed to be processed were reduced by a large factor.

Firstly, in order to remove multiple occurrences of the same data (for example, log entries having the same eventType, eventName, video module, and positions of timing, and even the time of recording the video), we introduced the DISTINCT clause into the MySQL queries to remove such occurrences. We found upon comparing that the use of the DISTINCT clause alone reduced the number of necessary and important events by a very large number. Also, the query ascertained that the events returned were arranged in the chronologically increasing order of time, so that the events could be processed in the order in which they actually occurred for the particular video and the user, allowing correct processing for our data model program.

However, this as far as the error correction and redundancy reduction by queries goes. Once these data were returned as lists of events by the queries run on the mysql connector functions (cursor.execute()), the rest of the error detection and correction were done as part of the Python programs written for the data model, although, the data model and the error removal scripts are separated from each other. The program was written in order to eliminate such rows, instead of using a join operation, because the join operation would take a longer amount of time (given the large number of events as entries in the table, requiring a self-join, therefore), and furthermore, it could be afforded, as the data was only required for the video once in a specified period (generally, a day), and so this computation could be done without

much hassle.

Following the entries returned by the MySQL queries, we proceeded to investigate into the results that were returned by the queries. It was found that despite the fact that the events returned by the query were all distinct with respect to the attributes being selected by the SELECT operator, there were several occurrences where the same event was being recorded multiple times, but over different times (over successive values of the createDateTime attribute, separated by say, 1 second in real world time). In such cases, we would consider only one occurrence of such an event, by comparing these events pairwise in sequential order. So, if for two successive events, all the attributes fetched had the same value, except for the createDateTime attribute, we kept only one of them in the new list which is being prepared and the repetitive instances were ignored.

```
if not (( activity1.eventType == activity2.eventType ) and
        ( activity1.eventName == activity2.eventName ) and
        ( activity1.oldVideoTime == activity2.oldVideoTime )
        and ( activity1.currVideoTime == activity2.currVideoTime ))
{
    Then keep the video in the final list of events.
}
else Discard
```

Now, our simple query also returned events such as page_close and save_user_state for events which did not correspond to the current video under consideration. However, such specific requirements for returning the rows/entries could not be specified in the queries, and we thought it better to process it through the Python program itself. So, we came up with the following programmatic solution: Keep ignoring events until the first event for the video under consideration is detected, and then apply the necessary filters to include events for the videos (as mentioned before), until the page_close event for the video occurs, which is the last event for a video under consideration. Once a pageclose event occurs, include the most relevant save_user_state event for that page_close event, and wait till the next relevant event for the video under question. Also, the data model in our design required that the page_close event be followed by the most relevant save_user_state event (in case multiple save_user_state events were recorded for the video in the list of events), because ideally, each page_close event would be followed by a save_user_state event. The logic for doing so is discussed in the next section where we talk about the general flow of events while a video is being played. Thus, all the events for a video being played are covered.

Furthermore, if there were such events for the current video where the video positions given were wrong (such as, being out of the duration of the videos), then these events would be evicted from the list immediately. There were also certain blatantly wrong instances of user log events, where certain necessary and characteristic fields were left NULL completely. Such cases of events too were evicted from the lists immediately.

Following this stage, we were able to get a cleaned, concise view of the activity of users for that video. This usually contained a much more manageable number of rows than the original data, which could be efficiently processed via a program as well as examined manually. We proceeded to design an algorithm to implement the criteria to extract this data from the log files, examined in order of the createDateTime field in the tuples. We aimed to cover all the tuples in the result in a single traversal, and calculate the data for the different features for the video.

1.4.3 Creation and prototyping of the data module

In this part of the activities, we analysed the various video events and determined a model to extract the features we will use to determine the time spent by the students on the videos they have watched, and accordingly use it for our summary and visualizations. We had decided on two features that we would map in order to find out the viewing activity of the students with respect to a particular video, which are as follows:

1. Count of the number of times the different regions of the videos were accessed by the different students.

2. Total duration spent on a particular region of the video by the student.

The hypotheses that formed the basis for our selecting these two features is as follows:

The more number of times a particular region is visited by students, and the longer time students spend there, the more difficult it is for them.

This hypotheses is justified because if the students repeatedly access a particular region of the video, and go through to actually watching it, then there would be an increase in not only the time spent by the user in that region, but also the number of accesses to the particular video section. In the worst case - the section might be the entire video. This would happen if the student was indeed stuck/perplexed by a particular topic discussed in the video.

Now, in order to find out the given metrics for a particular region of the video, it is essential to divide the video into several parts in the first place. For this, we had considered the concept of a timeFrame to divide the different regions of the video. A timeFrame is essentially a region of the video, x seconds wide, which is accessed by a user whenever he watches through the part of the video being spanned by that timeFrame, or completes more than half of that time frame during one play-pause session. So basically, we considered each video to be divided into a series of timeFrames of a specific width, and computed our metrics for assessing the difficulty regions by counting the number of accesses to the different timeFrames, and the total times spent by a user (or multiple users) in a particular timeFrame region. For the purpose of determining the timeFrames too, we would definitely need the video durations for each video, and this was also another reason why we proceeded to retrieve the video lengths from the YouTube data regarding the videos obtained using their YouTube id's.

In order to give an outline of analysing the log events, we must first discuss how the video events are actually appearing in the log files generated by IITBombayX. In order to truly understand this entire process, simulations and experiments were done by hand using multiple machines and real time supervision to observe the sequence and the types of log events that were being recorded in response to each event in the IITBombayX courseware pages. A sample video was opened in a particular window, and it was played, closed, stopped, rewinded, fast-forwarded, closed abruptly and then reopened - and we even went as far as to remove the internet connections and make and break the connections to see how the events were being generated. The log files were also analysed by hand, as were the results of running specific queries on the UserSessionOldLog table. Based on these, we were able to understand about the basic and ideal anatomy of a session of a video playing, right from the start when the video is loaded, upto the point when the user either closes the page or goes to another video.

All the video events are signified by the eventType attribute value 'video'. For each of the video events, the set of events can be as follows:

- load_video : Which indicates the loading of a video on a particular page, or a particular lesson.
- play_video : Which indicates the action of a user having clicked on the video on the webpage to start playing it.
- pause_video : Which indicates the action of a user having clicked on the play/pause button to pause the video in the middle of playing it, without stopping it or ending the video completely.
- seek_video : Which indicates the action of the user in using the mouse to drag the scroll button of the video/ the video pointer forwards, backwards, or moving it back and forth throughout the video to start playing the video at a different point in the video.
- stop_video : Which indicates the fact event of the video playing all the way till the end point, and then stopping, such that playing it again would mean that the video would start from the beginning.
- save_user_state : The event which records the last point in the video till which the video was watched by the user. This event is fired in a lot of different cases, which will be discussed later.

Other than these events, we also considered an additional event called 'page_close', which comes under the eventType 'navigation'. This event is responsible for actually determining whether the user has

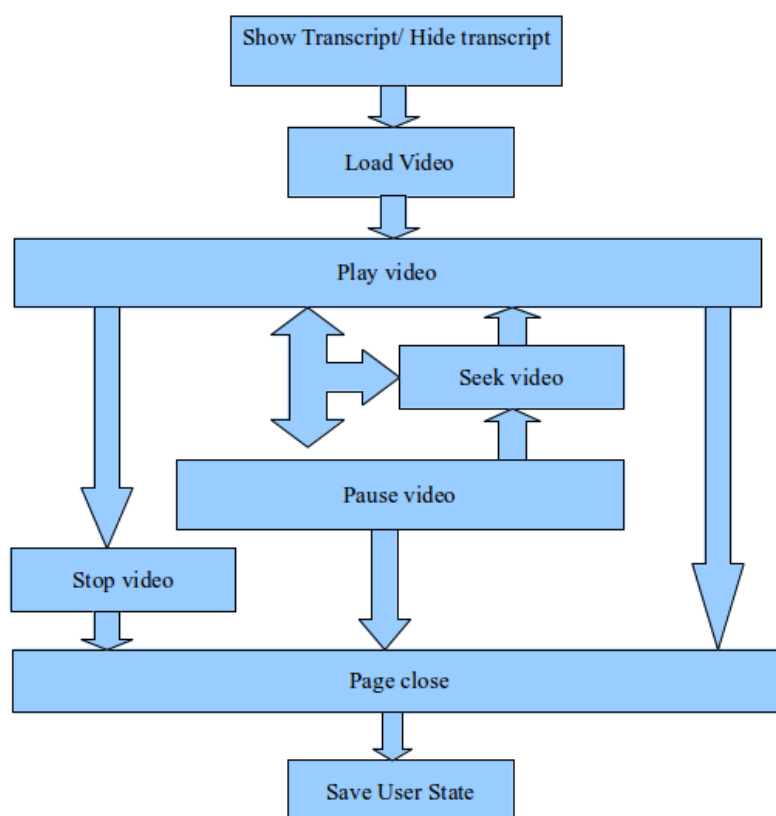


Figure 1.2: A simplified and ideal sequence of video watching.

is still on the page containing the video, or has actually switched to some other video and has left the earlier one. This event needed to be recorded, as it indicates the point of end of a user's activities for the time being, until the next time the video is loaded by the user for watching, and accordingly note the time watched by the user upto the closing of the page.

Let us now talk about the basic anatomy of a video playing event. The following diagram depicts a simple and very simplified flow of events for a single session spent by a user watching a particular lecture video 1.2.

As shown, the life of a video session begins when a user (generally a student) clicks on a link to go to a week or a lesson and lands up on a page which contains a video. As soon as the page is loaded, the video is loaded on to the browser side, and the several 'video' type events are triggered, such as: show_transcript, translation_transcript, hide_transcript, and of course, load_video. Usually, this event is followed by a save_user_state event, which records the last point of the video which was watched by this user. In case this is the first time this video is being watched, the currentVideoPosition attribute it would be set to 0.0; otherwise, it would correspond to the last position of the video upto which the user had watched and has been recorded by the last save_user_state event for the video. Once the video has been loaded, in the ideal case, the user would go on to play the video, and go on to execute the sequence of normal video processes as shown in the diagram. He may play, and then pause; or he may play, seek, and continue playing; or pause, seek, and then start playing, and so on — there are lots of such permutations of these events.

It should be noted at this stage that each pause video event would be followed by a save_user_state event, which records the last position upto which the video has been watched by the user, at the time of the pause. Also, when a seek_video event occurs, ideally, there is a save_user_state event which is fired which saves the last point upto which the video was watched before it was the seek operation took the video scroll to a different point in the video for the user. Following this, the video may continue playing or remain paused, as it were before scrolling backwards or forwards. Finally, when the user has

finished watching the video, the `stop_video` event is triggered, which records the fact that the video has ended. Following this, the user may start watching the video from the beginning again, or seek to a particular point in the video and start the same activities as before; or as in the normal, ideal case — go to another video's page, or a different lesson or week on the courseware. This would trigger a `page_close` event, which would be recorded by the event logs to give the page which has been closed by the user. Additionally, this would also trigger the event `'save_user_state'` for the video in concern, to record the last location in the video which was playing, or which had been paused by the user/student. Thus, the simple anatomy of the video session is discussed.

However, as we discovered upon examining the logs for individual users arranged in chronological order by the MySQL queries used on the `UserSessionOldLogs` table — the sequence of events of the users were not as simple as they appeared to be. While the basic outline of the events stays the same, we have to account additionally for the fact that the users may skip to different parts at different times, may open new videos in new tabs on the browser — or even certain errors and discrepancies in the events that were quite obviously, due to factors beyond the control of someone who is only looking at the logs generated. The errors included mistaken orderings of events, or events which were recorded at far different positions in the logs than where they were supposed to be. Most of these events would be due to the disturbances in the networks, so that successively generated events reached the record logs in an order different from the order generated. It could also be due to the scheduling behaviour of the tasks and the processes by the servers, which would again result in a similar situation. An example of this could be as follows: a `save_user_state` event, instead of appearing after the `page_close` event, appears before it in the log tables. Among other explanations for this discrepancy, one of the basic reasons why this could occur is because of the network errors. Thus, as part of the data model, we also had to develop logic to deal with such sequences of events appropriately, so that we could correctly extract the desired feature data we are desperately trying to get.

Basically, our plan has been to observe these video events and identify the sequence of the events for a particular student to determine whether the student has been watching the video or not - and if so, then how long he has been watching, which parts he has been watching, and whether he has skipped to certain parts in the video, or even repeated watching a certain part of the lecture video. In other words, the video events were closely analysed to get an overall view of the student's activities and interactions with the video in a single session, as discussed previously. For this we needed to track when the video was played and paused, and when the page was being loaded or the page was being closed by the user.

Now, we proceeded to design an algorithm to extract this data from the log files, examined in order of the `createDateTime` field in the tuples. We aimed to cover all the tuples in the result in a single traversal, and calculate the data for the different features for the video. In order to capture the different regions in the video that were visited by a user (and subsequently, all the users), we divided the entire duration of the video into a number of `timeFrames`, each of which was of width 4 seconds. So, the time frames were like: 0-4secs, 4-8secs, etc. The exact width of the `timeFrames` is still a matter of optimisation, and after running several examples and simulations, we can decide on the correct width of the `timeFrames` which would be most apt to capture the student activities. Each of these `timeFrames` were like a bucket, which were incremented each time they were traversed during a session of the video being watched by the user. The algorithm sequentially went through all the events, one by one, and maintained a track of the state of the video being watched internally through separate flag variables. Variables were also used to keep track of the range of the video that was watched by the user in a part of the video session, so that we could calculate exactly what part of the video was watched by the user until a change occurs in the behaviour of the user. A change in the behaviour of the user is explained thus: if a user, for example, plays a video, then pauses it, or seeks to a particular region of the video, or even closes the page — then we would account it as a change in the behaviour of the user. It does not mean that the pattern of behaviour of the user changes, — it simply means that the user does something else other than simply watching the video, and this leads to a certain disruption in the normal viewing activity. Hence, going in an incremental or rather, a step-by-step fashion, we will need to keep track of all the time in the video that the user has watched upto the time such a disruption occurs. Each time such a disruption occurs, we may or may not change the internal state of the video (as maintained by our algorithm), and we increment the bucket for all the `timeFrames` in the video that have been covered upto the point of traversal. The following is a list of the steps and the disruptions that we considered while designing the algorithm, and how we dealt with the disruption to appropriately extract the time period in the video

that has been watched by the user.

- From the time a video is loaded (namely, encountering the 'load_video' event, we consider the video to be in the paused state. We simply initialise our internal pointers to the time ranges, and wait for the video to start playing.
- As soon as the video starts playing, we note the starting point of the video playing event and keep on watching the successive times (or timeFrames) in the video the user watches upto. This continues as long as the video is playing, and is indicated internally by setting flags to indicate so. This is because the IITBombayX system, as we said earlier, pulses the events such as play and pause several times second, and over times times too. this worked to our advantage, as it allowed us to explicitly track all incrementally the point upto which the user had watched the video at that point.
- However, if it so happens that the same video is being watched simultaneously by the same user in separate tabs, then it would be necessary to capture all such events, and the same time Frames that are being covered by the user (because the effort counts!). The order wouldn't matter, as long as all the timeFrames are being effectively covered by our algorithm. So, as a special-case-handling mechanism, in case we noticed a different play_video time for the event, we considered a new video watching session was in order — so, we counted the time upto that point, and restarted a new video watching session from that point.
- In the most general case, the interval between a play and a pause event was considered as the time spent by a student. This is explained intuitively.
- Similar to the case as discussed with the play_video events, we would also need to track the pause_video events for videos playing in other tabs as well. In the general case, the student would quite obviously close the video he is watching in one tab before switching to the other one. However, he might even watch them simultaneously, for the purpose of comparing, and hence we would need to consider this effort being put in by the user. So, in this case, we wait and note for other different pause_video events that may have occurred, and accordingly, account for the time watched by the students.
- The time spent upto a seek was considered the time spent by a student, and the final point of seek marking a region that is probably being visited a second time (although, it may be a case of seek forward – where a student skips past a part of the lecture to go to a different part of the video, because he/she already has the knowledge being imparted in that part). In any case, we will consider the video, if 'playing', to go into a pause state immediately, until the next play_video event for the video. The time spent by the user upto the start of the seek_video event (in which case, the start point of seek would be given by the oldVideotime attribute o the log entry) is then recorded as the region of the video watched by the students. if the video is already paused (due to being paused prior to seeking by the user, or simply because a seek event has already happened for the video by the user), then we only account for the change in the starting time (or timeFrame) of the video — indicating the point from which the user would start playing the video, if there is a new play_video event.
- In case of a stop_video event, we simply account for the times in the video that have been watched by the user upto the end point of the video, and consider the video closed. of course, the save_user_state events would account for the current time upto which the upto which the video has been watched, thus keeping an additional track or check on the video activity. In the beginning of the playing of the video, this would indicate the point where the video was stopped earlier.
- Finally, once all is said and done, we come to the situation where we the video and page(or tab) is closed for good. This may occur once the video has stopped playing, or is paused, or even while the video is playing. In all of these cases, the closing of the page of the video is recorded by the IITBombayX platform as being a page_close event, as mentioned before, and is immediately (usually) followed by a save_user_state event. As we have mentioned earlier in the section on error detection, we also had cases beyond our control where the events appeared out of the logical order of occurrences — in this case, save_user_state appearing before the page_close event. So, in the processing stage prior to the data model, we had reordered the save_user_state event that was most relevant related to the page_closed event following right after the page_close event.

- In case the page of the video is closed and the video is paused, we do not have anything to worry about. We have already updated the regions of the video that the student has watched up to the `pause_video` event. However, if the video is still playing when the page is closed, then we would do better by counting the time in the video upto the point where the page was closed. This can be indicated by the fields in the `save_user_state` event associated with that `page_close` event. Of course, the internal flags would be set to indicate that indeed the page has been closed, and that the video is currently paused
- As a final correctional measure, we included the situation where the `page_close` event has not been recorded for the last video session of the user in the logs. However, we still need to include the time watched by the user — so, we consider the time watched by the user upto the last point in the video.

From the `timeFrameBuckets` list, we can get the count for the accesses to different regions of the video. The process is thus continued for all the users for a particular video, and ultimately, for all the videos in a particular course, as is given by the logs provided. Thus, the output is produced by our program, which are basically sets of tuples for every video watched in the logs, every user who has watched it, and the `timeFrames` in the videos. At the prototype stage, the feature data generated after the data-model stage would be tuples or entries in a MySQL table, which would then be used for inference generation, or visualizations (in words, out final steps of work). The tuples would have the following characteristics:

- Primary key : \langle an incrementing unique numeric id \rangle , `videoSysId`, `userName`, `timeFrameId`
- Non-key attributes : `frameAccessCount` and `frameDuration`

Once the table is populated with this data, as we said, we can generate our summaries from this table, to be used by visualizations and for learning methods, reports, and so on. Some of the possible summary information we can extract from this data could be as follows:

- Total time spent by a user on a particular video : We could execute an aggregate query on the final summary data to extract the total amount of time that a user has spent on the total video, by summing over all the `timeFrames` and all corresponding `frameDuration` entries for that particular video in that course.
- Total time spent by all users in a particular `timeFrame` : An aggregate sum function could be executed on a particular `timeFrameId`, individually for all the users who have watched that particular video `timeFrame`. Typically, this would be a SUM function executed on the `frameAccessDuration` attribute for the selected `timeFrameId` for all the users for a video, grouping the entries by the `videoName` and `userName` attributes in the summary tables.
- Total number of accesses to a particular `timeFrame` : An aggregate sum function could be executed on a particular `timeFrameId`, individually for all the users who have watched that particular video `timeFrame`. Typically, this would be a SUM function executed on the `frameAccessCount` attribute for the selected `timeFrameId` for all the users for a video, grouping the entries by the `videoName` and `userName` attributes in the summary tables.

As stated earlier, in order to develop the prototype of the data model and test run the algorithm, for correcting and perfecting it, we needed to run the program on some sample test, but very real data. Thus, we ran our program on the videos of the CS101.1x course. We first tested this on the top 10 most watched videos in the CS101.1x course, and on the top 50 viewers for the corresponding videos. The queries used to find the list of the videos and the users are similar to the ones we have used before for extracting the data for a single video and for a single user. The process was executed repeatedly for the different videos and the users to extract the final feature data for the videos.

The following graph 1.3 was plotted on the basis of the data obtained by running the module on a single video, and its top 50 users. The video in consideration had the `videoSysName` = `'67a8559582864d6a8148e2ef5c997e8f'`, or otherwise known as the video on Elementary Graphics, in the CS101.1x course.

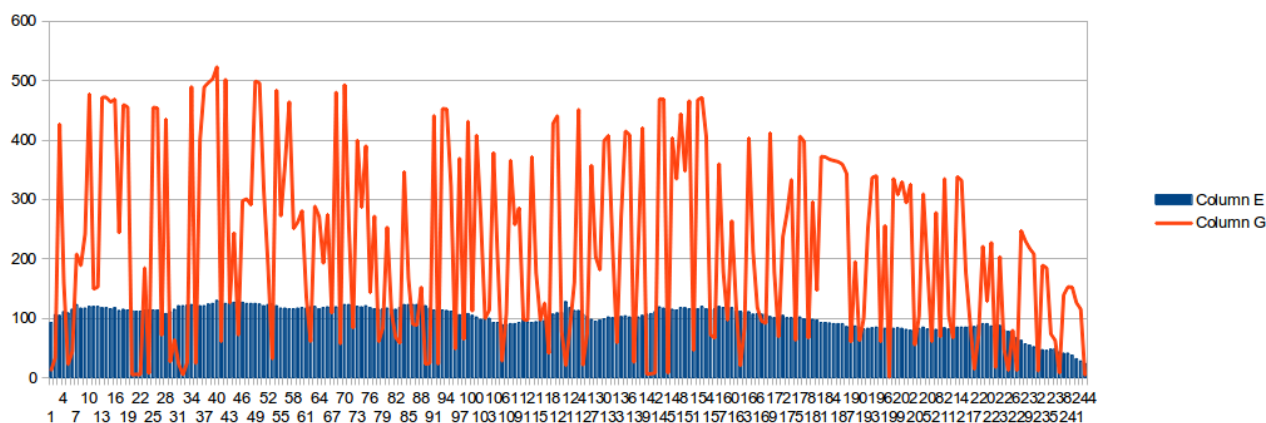


Figure 1.3: An example graph. Legend: X-axis - Time frame number. (dimensionless) Red curve - The total time spent by all the users in a particular timeFrame. (seconds) Blue Columns - The number of accesses to those timeFrames. (dimensionless) .

1.4.4 Final Implementation and visualization

Once the data model was finalized and found to be working properly at a prototype level, we took it the project to the final step — implement the data model in Spark to draw the data from populated Hive tables containing the cleaned Users' log data, put the processed data back into the Hive tables, and finally process the result data to create summary data that would be populated into the MySQL Analytics database tables, which would be used for visualization on the IITBombayX platform dashboard for the instructors.

The final data module would be working as a live application in a Big Data environment, as has been said before — since it will deal with tens of gigabytes of log data generated by the IITBombayX platform. To appropriately handle this, we would require a tool that can efficiently work with such vast amounts of data, but do it efficiently nevertheless. Spark does that for us, using the concept of RDDs, and allows us to run queries similar to SQL on the RDDs it creates from the data in the memory. To work with such a setup, an installation and proper configuration of Hadoop (for distributed filesystems), Hive (for creating data warehouses), and Spark (for running SparkSQL queries and executing operations on the data). Once this is done, the workflow of the operations, and in a more broader sense, the algorithm for the Spark programs were decided upon.

1.4.4.1 Extracting the data from Hive tables

The data that we originally worked upon (the UserSessionOldLog table) was a MySQL table. However, working in a Big Data environment requires the maintenance and access of a fast data warehouse, such as Hive. In order to transfer the data from the MySQL database to the Hive tables for the same, we used the Sqoop utility to transfer the data. The command for the Sqoop utility can be written as follows:

```
sqoop import --connect jdbc:mysql://localhost:3306/IITBxDataAnalytics
--username root --password '' --table UserSessionOldLog -m 1
--hive-import --hive-table default.userlog
--warehouse-dir /user/hive/warehouse
```

The command can be explained as follows. We specify the connector and the database to connect to using the `--connect` option, and together with it, the username and password to access the data base, and the table to be imported. Then, we specify the name of the destination table in the Hive database and where the warehouse for the hive tables is located on the Hadoop HDFS (this can be verified by going to the Hadoop administration page for the namenode, and then browsing the filesystem. The process was completed successfully for us, when we also transferred the CourseVideos table and a sample of the UserSessionOldLog table to the Hive databases.

While it is known that Spark runs very fast, it is required that the data must be in the main memory in order for the processing to be very efficient. So, this required us to specify a single query to draw the user data from the Hive tables in a single go, and perform further processing on it 'in-memory', using the concept of RDDs where useful, and then collecting the data from the RDDs when they needed to be used directly. So, work was done to optimize and perfect the right queries that would fetch the necessary event data in a single go from the databases. The query which finally worked was as follows:

```
SELECT DISTINCT orgname, coursename, username, modulesysname,
               eventtype, eventname, oldvideotime, currvideotime, createdatetime
FROM userlogsmall
WHERE eventtype='video' OR (eventtype='navigation' AND eventname='page_close')
ORDER BY orgname, coursename, username, createdatetime
```

This query can be explained thus. We divided all the events based on the organisation which conducted the course, the name of the course, the user who was involved in the event, the module being used or involved, and the eventType and eventName for the corresponding event, together with the oldVideoTime and currVideoTime if the events are of the type 'video', as is required by us. The events are then ordered by the organisation, coursename, the username, and then the time of the event — so that while processing the events, we can not only determine the temporal sequence of the events, but they will be processed depending upon the user who was involved in the event, so that all the behaviour of the user can be taken into account while processing his behaviour during the watching of a single video.

To begin with, in order to test the final implementation of the data model, we selected a subset of the original records to contain only the records of the top 30 most active users on the platform, using the following MySQL/Hive query (the records were first isolated into a separate table 'userlogsmall' in MySQL, and then exported to Hive via Sqoop):

```
create table userlogsmall (
  select USOL.* from UserSessionOldLog USOL,
  (select userName, count(eventType) cet
   from UserSessionOldLog
   where userName is not NULL and userName not in ('')
   and courseName='CS101.1x' group by userName
   order by cet desc limit 30) t1
  where USOL.userName = t1.userName and courseName='CS101.1x')
```

1.4.4.2 Filtering of events per user and per video

Following this step, the events for each user and each video were separated out from the event logs in the order in which they have been extracted. The idea here was that instead of extracting the events per video and then per user, we extract them per user, and then per video because that captures any changes of patterns in the user's behaviour in watching the videos. The result of this process was a single key for each orgname, coursename, videoname and username (for every user watching a video in a course), and its corresponding value being a list of events for that user and that video in that course. A number of conditions were considered while computing these lists by sequentially going through the sequence of events extracted. They are outlined as follows:

- For each log event of the type 'video', we are sure of the video id and the user who has watched it. So, it is immediately appended to the list of events for the corresponding key (i.e., (orgname, coursename, videoname, username)).
- If a log event is the very first log event in the sequence of events for a particular user (which is also the case of the first log event in the result set of the queries executed before), then we check if the event is a not a save_user.state or a page_close event. If so, then it is a video event with a definite username and videoname, we add it to the list of events for that key constructed. Otherwise, we ignore it, as we have no use for an event taken out of its context. For example, a page_close event maybe the first event in the result set for a particular user, but it may be the page_close event for a module other than a video — and this is the case, as has been observed.

- If, however, we encounter a `save_user_state` event or a `page_close` event for a user we are currently working with, we keep track of the last (user,video) key we were working with, and add the event to the list corresponding to that key, because it is most likely related to that (user,video) combination.

The above operation was carried out all in Python code written in the Spark module. This is because, performing join operations, given the volume of records we are dealing with, will result in a huge amount of memory usage and processor time, which can easily be tackled if we instead go for the sequential verification strategy selected over here. The events are verified, and associated with the corresponding video and the user watching the video.

It should be noted at this stage that in the table for the course videos, there were certain errors in the details of the videos recorded. In particular, the video with the id '2017625218ba4309b4cd42309f5d82e2' had neither youtube id nor a title name. Hence this video had no duration calculated from it, and the events corresponding to this video could not be considered. In order to deal with this, we first allowed our filtering algorithm detect and accumulate the events for the video and for all the users that have watched the video. Then, the events for the corresponding keys having that video name/id were eliminated from the final list of key-value pairs, the value here being the list of events for the user and the video.

1.4.4.3 Processing and generation of results for final summary table

Now that the events have been filtered out and associated with the corresponding videos and the users, it is time to process the events and generate the statistics for the timeFrames that have been visited and accessed by the users while watching that video. At this stage and in the summary stage, the concept of MapReduce workflow was used to speed up the process of analysing the data generation and processing tasks. The concept of MapReduce revolves around the fact that the data can be converted into a set of key-value pairs by passing it to a mapper function, and these key value pairs are then 'merged' together using a reducer function to create the final result. It is a workflow methodology which basically uses parallelism, and given the scale of data we are working with, it worked to a great advantage for us.

Prior to this, the RDD (Resilient Distributed Database) was created for each of the (key,value) pairs, where the key was the a combination of the orgName, courseName, videoSysId and the userName, and the corresponding value was the list of events for that key. The key value pairs would look something like this:

```
(
  (u'IITBombayX', u'CS101.1x', u'fa1f6040f46a43298cc25fc33db89a83', u'Shrikrishna'),
  [
    (u'video', u'load_video', 0.0, 0.0, u'2015-02-17 08:22:03.0'),
    (u'video', u'play_video', 0.0, 0.0, u'2015-02-17 08:29:32.0'),
    (u'video', u'pause_video', 0.0, 11.525, u'2015-02-17 08:29:32.0'),
    ..... and so on....
  ]
)
```

The RDD parallelized this combination to create the key,value pairs, which were then sent to a mapper function called `processing()`, which essentially performs the analysis required on the key-value pairs, as described in the section on the development of the data model prototype. The single function contains all the code needed to generate the details for the timeFrame access for a single video and a single user who has watched it. The code for these parts are given as follows:

```
eventMap=sc.parallelize(eventDict.items()).map(lambda p: (p[0], p[1]))
videoDetails = eventMap.map(processing)
```

While the input consisted of key value pairs where the value was a list of events, the output produced key-value pairs having the same keys, but having the timeFrameBuckets list as the value. To recall, the timeFrameBuckets list is a list of bi-tuples (x,y) , where x denotes the number of accesses to a particular timeFrame, and y denotes the time spent by the user in that timeFrame of the video. After processing, the data would look something like this:

```
(
    (u'IITBombayX', u'CS101.1x',
     u'fa1f6040f46a43298cc25fc33db89a83', u'Shrikrishna'), [[4, 16.0],
                                                             [4, 15.05],
                                                             [2, 8.0],
                                                             .....and so on ....]
)
```

This step thus produces the necessary data of the analysis, which can be used various kinds of purposes later on, such as visualizations, summary metrics, and learning activities. Also, this data could be stored as itself, in the form of a tuple in the Hive tables, so that each time a new visualization needs to be created, the processed (but not summary) data can be accessed from the Hive tables.

In case of final implementation of the data model and processing module, the processed data generated would be stored in the corresponding Hive table as tuples having the following fields:

- Primary key : \langle an incrementing unique numeric id \rangle , orgName, courseName, videoSysId, userName, timeFrameId
- Non-key attributes : frameAccessCount and frameDuration

One of the possible visualizations that can be generated from this data is the total number of times accessed and the total time spent by users in a particular timeFrame of a particular video. What this would give is an overall analysis of the number of views and the time of the views on the video in question. An instructor would be able to use this analytical data to determine what are the regions of the videos that are being watched by users (who have watched this video), most frequently. Accordingly, these could be the regions of investigation for the instructors, as these are the regions of difficulty for the students as stated in our hypotheses.

The summary data is generated as follows. First, the results of the previous analysis stage would be converted into an RDD through the parallelise function, and then converted into appropriate key-value pairs — only this time, the key would not contain the user id. Instead, it would contain the videoFrame number, because we are required to do a sum over all of the users for a single videoFrame in a single video. This is done by converting the key and list combination as given before into a list of key-value pairs, and then reducing all of the lists into a single list using a reduce() job, which basically concatenates the lists together.

Following this, the result of reduce() was again parallelized to get an RDD, on which, we ran a final map-reduce job on the keys as stated before, but this time, the reducer function we used added the tuples together to sum up the total number of accesses and the total time spent in a timeFrame. The results of this map-reduce were then written to the MySQL summary table video_difficulty_analytics using Python code.

The workflow stated can be summed up in the two code lines as follows, which illustrates the power of Spark and of map-reduce:

```
videoUserTimeFrameAggregate
    = sc.parallelize(
        videoDetails.map(
            lambda x: [ (x[0], i, x[1][i]) for i in range(0,len(x[1])) ]
        ).reduce(lambda a,b: a+b)
    )

#x[0]--> key as used in processing(), i--> timeFrame id, x[1]-->timeFrameBuckets

result = videoUserTimeFrameAggregate
    .map( lambda x: ( (x[0][0], x[0][1], x[0][2], x[1] ), x[2] ) )
    .reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1] ) )

#x[0][...]: orgName, courseName, videoSysId,
#x[1]--> timeFrameId, x[2]-->tuple of count and timeSpent.
```

The summary data generated for creating the visualizations would be stored in the corresponding MySQL table (video_difficulty_details) as tuples having the following fields:

- Primary key : id (an incrementing unique numeric id), orgName, courseName, videoSysId, videoFrame
- Non-key attributes : count and timeSpent

The summary data was then drawn from these two tables using an R data visualization script using the googleVis library to plot a graph on the analytics dashboard. The graph had two y-axes: one for the timeSpent on the video timeFrames (indicated on the X-axis), and one for the 'count' field, giving the number of accesses to the timeFrame in the video by all the users.

1.5 Future Work

Although we have so far been able to visualize the feature data such as time spent in different regions of the video, and the number of times people have visited the different regions of the video. However, the project has greater potential than just this. Once we have the data to work with, we determine the patterns in the overall viewing activity of not only a single video, but also all the videos in a particular course undertaken by an instructor. One possible approach could be to first determine clusters of high video activity (namely, large number of views and number of accesses) across the various timeFrames of the videos. Once these clusters have been determined, one could possibly create a type of scheme for determining the score of such a cluster. The reason for this being that we will have clusters in all the videos, depicting the regions of difficulty in these videos. So, we need to be able to compare these clusters with each other, in order to determine which clusters indicate a higher level of difficulty than the others. The clustered regions for different videos in a course could be compared with each other in order to find the regions of difficulty, and report them to the instructor. The instructor, as said in the beginning, would receive notifications for the same, and take appropriate actions to alleviate the problem.

An even further improvement to this work could be the automation of the recommendation and alert process. In such a system, the system itself would recognize the regions of difficulty in the videos using the method described above, and automatically recommend materials and course videos to the students, thus automating the educational platform to make it even more personalized for the people accessing the course, requiring only the instructor to manage the course, monitor student activity, and address any queries the students may have regarding the lectures.

1.6 Technologies Used

A variety of technologies have been used by us during the development of this data model and final application module. In particular, we used the following during the mentioned stages of the development process:

1. Prototype stage :

- MySQL : The database management system used for storing the UserSessionOldLog and CourseVideos table, and in the final implementation, for storing the summary data extracted from the video events of the users.
- Python : The language base used for programming in this project, and for prototyping and testing the data model.

2. Final implementation stage :

- Hive : The data warehouse infrastructure utilising Hadoop as well as allowing access to files as RDDs for executing queries.
- SparkSQL : The fast data processing engine, using in-memory primitives, and faster than Hadoop.