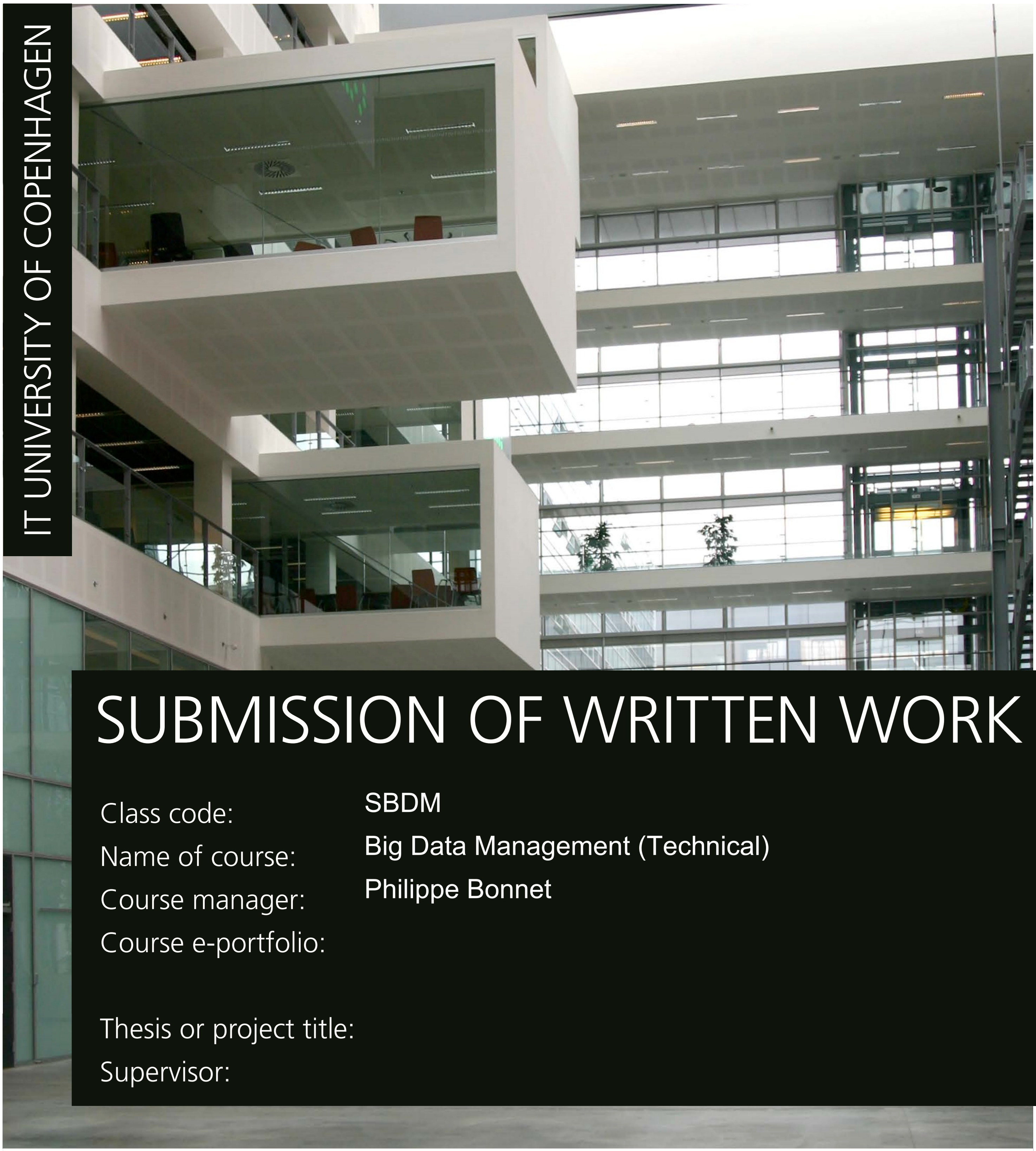


SUBMISSION OF WRITTEN WORK

Class code:	SBDM
Name of course:	Big Data Management (Technical)
Course manager:	Philippe Bonnet
Course e-portfolio:	
Thesis or project title:	
Supervisor:	

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Anders Wind Steffensen	10/02-1993	awis@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Big Data Exam - Autumn 2016

Anders Wind Steffensen

December 13th 2016

Contents

1 Question 1

1.1 A)

"Consider Apache Flink: `https://flink.apache.org`*. You should characterize this system, describe how it can be used in the context of the Lambda architecture and compare it with systems you have used during your projects."*

Apache Flink[Flink] is a streaming dataflow engine. It works in a distributed setting and makes analysis of data in motion, and data at rest easier. It incorporates multiple other systems, for machine learning, graph-analysis, and more. To further characterize Flink I will use the characterization model presented in the course.

Datamodel: Flink works on event-based streams of data. The specific format of the events are Java and Scala embedded objects. These streams can either be infinite such as a sensor which continuously sends data, or finite such as a file. Kapfka which is a stream gathering and storing framework based on HDFS, is used by Flink to get its stream of events[4]. If events come out of order in real-time, Flink is able to sort them based on logical time instead, which can greatly simplify for example windowed computations.

Partition Management: To be able to scale, Flink partitions the computations on multiple nodes, which can be placed on the same server or distributed on multiple machine on a network. This approach is different from working with data in rest, where the partitioning is based on the data itself. Flink works with streams and as such, partitions the computations instead and the data/events flow between those computation nodes. Flink automatically tries to optimize the placement of the operations nodes, such that the overhead of sending the events through the network is minimized[2].

Failure handling: Flink supports replaying of a stream to be able to recover from failures, that is if a failure occurs the stream is replayed from the last checkpoint.

The checkpoint mechanism is different from what most other big data systems use as a fall-back mechanism. The state of the nodes is periodically persisted on HDFS or in memory, such that in case of a failure replaying from that checkpoint is possible. The algorithm behind the checkpoint barriers is based on the snapshot algorithm by Chandy and Lamport[3], such that the checkpoint is serially consistent across the distributed nodes, and it is not necessary to duplicate information. Once all data has flown through the barrier/checkpoint the computation is done and the computation between checkpoints either succeeds or fails atomically as a whole. The flow of the events are never held back or stopped by the checkpoint mechanism and as such, most nodes will always be in use and bottlenecks on some nodes should not hinder other nodes to d their calculations. The flow of the events happen in a directed acyclic graph structure which also means that the consistency is strong since any parallel process will always see the data in the same order[3].

The checkpoint technique also separates the responsibility of calculation and failure handling, since changing the frequency of checkpoints does not alter the results of the stream.

Batch and Stream Processing: Flink provides two APIs, one for batch analysis and one for stream analysis. Since Flink only works on streams of data, batch processing has just been implemented as *finite* stream processing. This makes Flink a framework which is based on the concept of queries at rest, instead of data at rest. An advandtagde with this approach is also that the processing code can in a lot of cases be reused for both streaming and batch views. The two API's can be used from Java or Scala, and they provide an interface much like the Java 8's Stream library[1], where it is easy to do SQL-like commands, such as `where`, `groupby`, `sum` and so on. Furthermore Flink supports streaming windows over both time or counts which allows for more sophisticated analysis. As explained in the partitioning section, the different operations are distributed over a network of nodes, which each have a specific responsibility in the computation of the view.

Throughput: Flink prides itself with being low latency by having a powerful API to do stream processing, and by offloading some of the batch processing to the stream processing. What the streaming analysis does for low latency the ability to send information quickly forth to batch analysis, as well as the ability to scale horizontally allows for high throughput. Of course, any system which builds on top of layers of abstraction will have some latency which is not present when working on the bare metal.

On data-artisan.com[2] a graph of the throughput of different big data system is made. On figure 1 the graph can be seen. The strengths of Flink become very apparent, and as we can see in this case the throughput is many times higher than for example storm, which is also known for its high throughput. Flink even has a lower latency, than any of the other measured frameworks. This example is on a computation which doe snot require any reshuffling of data, and on another example the website shows a graph where the latency is noticeably higher than storm for example. Therefore the performance greatly varies based on the wanted computation.

[1] `https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`
[2] `http://data-artisans.com/wp-content/uploads/2015/08/grep_throughput.png`

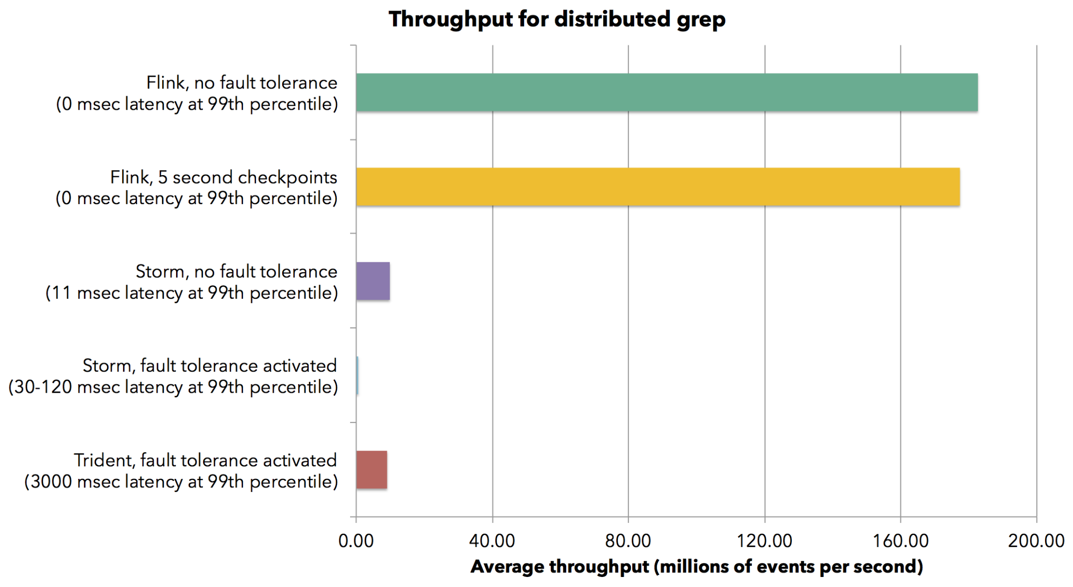


Figure 1: A graph showing the performance of different big data processing frameworks.

Interestingly enough Flink compares very well to the Lambda architecture. The Lambda architecture as introduced in the course, is separated into the following parts; the data sources, the master dataset, the batch layer, the serving layer, the speed layer and the queries. At the batch layer, batch views are computed, and similarly at the speed layer streaming analysis is done, and therefore it becomes quite clear that Flink naturally fits the lambda architecture by being able to be the main framework for each of these layers.

One point where the two are less alike are that Flink only works in terms of queries at rest, even in the batch layer, whereas the batch layer is in the lambda architecture described as data in rest with the processing *moving* over the data.

For project 2 and 3, we could have made both batch jobs and streaming jobs, but decided to only develop batch jobs. Most of our batch jobs are made out of logic which could be expressed as `-where`, `-join` and `-groupby` statements which are available with Flink. Therefore it would be possible to have written the batch views with the Flink APIs, and have gained higher parallelism, and furthermore had the ability to with ease introduce a speed layer which could do similar calculations, but allow for lower latency from data to queries.

1.2 B)

> *"You are asked to store a master data set of 80 GB given to you as an XML file. Why is the XML data format problematic when working with Map-Reduce? Would a format transformation from XML to JSON be helpful? Would a transformation from XML to CSV be helpful? How would you store this master data set? Explain your answers"*

The XML format is a tree structure format, and might not be easily be partitioned into smaller parts, which can be distributed among the data nodes of the storage system that Map-Reduce works on (HDFS). Furthermore XML is also a very verbose format and therefore data will take up more space which will make the transfer of data somewhat slower and require more space on the server. Even though Hadoop of course handles this quite fine, using less space is always to be preferred when the information is virtually the same.

Transforming the data to JSON would mostly help with the amount of data, since JSON is less verbose than XML. JSON also allows for tree-structure objects and therefore partitioning can still be difficult. Since JSON objects do not specify a start and an end tag it can actually be more difficult to split up than XML.

CSV on the other hand is a flat data structure and therefore is easily partitioned per line and split across multiple machines. Appending new data is also easy since each row is self-contained, it can be added to any data node, and therefore it is easy to do load balancing. Furthermore CSV has the advantage that it is not very verbose and it is easy to extend the input with more columns if needed.

Another possibility that we have used in Project 2 is to use a binary format. We used the serialization framework Avro[3] for this. By choosing to use a binary format, you can represent numbers as actual numbers and not text, therefore cutting down on the needed space. Furthermore if done properly it is still possible to make the format flat and therefore making partitioning easier. This approach of course requires more work to be done, and also enforces a scheme on the data, which makes it more difficult to extend the system later on.

For project 3 we converted the data to a CSV format and stored it using Hive since it was well integrated. Hive stores the CSV data as distributed files on HDFS but uses an abstraction layer which makes the access like a regular database access. Hive allowed us to make all the views we needed and therefore it seemed like a good choice to store the data.

[3]`https://avro.apache.org/`

It is important to notice that as soon as one transforms data, it per definition becomes derived data. Therefore, the master data set will be derived if stored in another format than it originally was, which asks the question on what to do with the primary data. I will argue that a transformation can be done from XML to CSV, which allows one to go through a similar process which converts the resulting CSV back into XML data. Even though the resulting data is not the same, it is equal to the primary data, and therefore just keeping the CSV and not the primary data is enough.

1.3 C)

> *"Describe pros and cons of using the Hadoop ecosystem, based on the lessons you learnt from project 2 and project 3."*

The Hadoop ecosystem, has changed the industry, and how it looks at data in general. By going from a restricted view, the big data movement tries to break the boundaries but it is still very much in its youth. The relational databases go back to the 1970s and have had many years to polish its rough edges and making it easily available to developers. The big data movement is still trying to do this and most frameworks in the Hadoop data system exactly tries to sell themselves on their easy of use, but in my experience most of these systems still have a high learning curve.

Setting up a server with a Hadoop ecosystem requires a lot of time. Systems like Horton tries to make this process easier by creating a single entry point for organizing and managing the large amount of the different Hadoop frameworks which are often required to have a complete big data system with on par functionality as a relational database system.

Another con that we experienced when working with the Hadoop ecosystem is that integrating different frameworks is often difficult. Since standards seldom exist, frameworks are made to integrate well with other specific frameworks, but if it is desired to integrate with another system then the developers are often left to figure how to do that themselves if it is even possible.

For small systems, or embedded systems where the amount of data is know and will never surpass the amount of memory on the system (The limit of memory on most consumer hardware is around 64GB), using local computation power will be faster since it does not require any distribution of data. A lot of the frameworks also have a lot of overhead on what they do, even though they scale better. Therefore in certain systems going with a SQLite database or a system specific Relational Database would be the best solution.

That said, the Hadoop ecosystem really shines when it comes to large amounts of data. A lot of businesses saw a huge rise in the amount of data they stored through the 2000s with the rising popularity of the internet, and now that processors were nearing their clock speed limit, being able to scale systems horizontally were very important. The Hadoop ecosystem is build around concepts of being able to abstract the distributiveness of the data away and allow developers to write code which automatically would scale to an large amount of machines in a warehouse.

Facebook has neared the limit of the maximum amount of data that the core Hadoop ecosystem could handle. They had a data warehouse with 100 petabytes of data. The current problem with Hadoop is that the machines needs to be placed in the same warehouse to share the data over the network, without spending too much time on sending the data[1].

By being able to handle these large volumes, data becomes available to analysis and examination, which is one of the most important concepts of big data.

One of the pros that we experienced was how well abstracted the interfaces to most of the frameworks was. When you use Hive for example you do not have to think about how the data is distributed because the system itself will do that for you, and therefore you minimize the chances of making mistakes. Of course having extensive knowledge of the system allows you to fine tune the system even more.

The Hadoop ecosystem also focuses a lot on failure handling. It is possible to setup HDFS such that it does not have a single point of failure. Even in the case of machine breakdowns the frameworks should be able to gracefully handle it and be able to replay, reroute, or abandon the process, or retrieve the data elsewhere. The developers are able to specify which approach should be taken as to how to restore the data or process of the crashed machine.

For a lot of developers it has also been a deciding factor that the Apache foundation requires its projects to be open-source, which encourages developers to help find bugs, and ensures that the system does not require any proprietary frameworks, operating systems, or hardware.

To conclude on this it becomes obvious that if a system is going to scale, it is a good idea to use the Hadoop ecosystem since other systems might not be able to handle the same amounts of data, but if the system is of limited scale, the overhead of using the Hadoop ecosystem is quite high.

2 Question 2

2.1 A)

"Consider the data set from project 3. How much of the work you did in project 2 to clean data could be reused to clean the data set from project 3? Explain your answer."

From a code perspective it would be difficult to reuse the source code of Project 2 to clean the data from Project 3, mostly because we used the serialization framework Avro, which then requires the data to be in a certain format, and outputs data in a specific format to project 2.

We used streaming to clean the data in Project 2 so in some sense it would be possible to reuse the streaming part, since by lazily streaming the data, it is possible to handle the 80GB of data in Project 3. Then by rewriting the logic of what to remove, label or ignore, the program would eventually complete.

Referring back to question 1C it should be noted that as soon as we clean data, we can no longer (unless the cleaning only tags, or ignores) assume that the derived data is the same as the primary data. Because of this an approach to backing up the original data or otherwise it should be made very clear that whatever analysis is made on the data will always be done from derived data.

We choose to not remove or delete any data in the cleaning process but simply ignoring it and allowing the batch computations to decide whether or not to use data. This was done to make it possible for future batch views to use the outliers and missing values for other computations. For example if the operation system was unknown we did not use that WiFi client in the view, but it might be interesting in the future to see how many WiFi clients had unknown OS types. Having information about the what data is not there can be interesting in itself and therefore we did not remove it from the master dataset.

2.2 B)

"Describe a cleaning process for the data set in project 3. Describe the design of a system that implements this cleaning process."

A cleaning process over this data could include checking for valid values, such as speed values that are negative. Then checking whether or not each entry falls into one of the two well defined categories, Vehicle-type or Person-type. Another step in the cleaning process would be to check for missing information or information which should not exist for an entity.

One of specific parts of cleaning the data of Project 3 is the fact that sometimes, when cars have been staying still for too long they are randomly teleported to other parts of the map. If some analysis would be done on location, some cleaning process needs to handle this. This could be done by checking the delta of x and y coordinates compared to the last position of the car. The new value could in case of a teleportation detection, be labelled such that the batch view could handle the entry correctly.

To create such a cleaning process, I would create a Map-Reduce program such that it can handle the large amounts of data. Then I would create a mapper for each of the different procedures, and checks, which each output to the next mapper in a pipeline fashion. By doing this it is possible for map reduce achieve higher concurrency. A reducer could then be placed at the end, aggregating all the results into two lists of entities, one for vehicles and one for people. At the end the results could be stored on HDFS, ready for batch or streaming analysis.

One could also implement this process as a Hive job, which would have the obvious advantage that Hive itself would split the process into multiple stages of mappers and reducers automatically. Though one disadvantage of this approach is that the data would first have to be imported into Hive tables and then the result would have to be put into another table, or the original data removed.

3 Question 3

3.1 A)

"Assume that the data from project 3 is not a massive data set, but a data stream. Every time step, a large collection of vehicles and persons is generated (based on the attributes contained in the ¡vehicle¿ and ¡person¿ elements of the XML file given in project 3). How would you proceed to characterize such a data stream?"

To characterize the stream of data I decide to look at three factors; structure, mean throughput and peek throughput. I assume that the stream is irregular. One could also have chosen to characterize the data of the stream, such as the range, min, max, outliers and so on of the attributes.

In project 3 the data of the stream contains semi-structured data since it is in XML format. The structure in question is as follows:

```
<Timestep>
    <vehicle, id, x, y, angle, type, speed, pos, lane, slope/>
```

```
</Timestep>
```

or

```
<Timestep>
    <person, id, x, y, angle, speed, pos, edge/>
</Timestep>
```

depending on which type the input has. In general when working with XML the structure of the stream is easily obtained since the structure of the data pr definition is part of the data itself. In some cases the structure of the XML is even specified in an explicit XML schema. Had the data been unstructured it would have been much more difficult to define, since one should try to create and fit a schema at the same time.

The mean throughput can be described as the average of the number of discrete elements over a time period (for example per second). Often when working with infinite streams we need to specify and limit the period of time to base this mean over, since calculating over all existing data can make the computation very expensive. One way of calculating it could be using a window of the stream. By examining a window in the stream one could approximate the average amount of entities pr timesteps, by dividing the number of elements with the delta time of the window. This approximation of the overall mean throughput would probably also be more describing than an overall mean, since streams are infinite and ever changing. The overall mean would not be representative for the stream currently, and it would therefore be a bad foundation to base decisions on. The current mean throughput might be a better indicator on whether to scale the system horizontally or not.

A third characterization could be the peek amount of elements in the stream. This could be approximated by comparing the current mean throughput with the last highest mean throughput. By using the mean instead of absolute values, the computation is less sensitive to very short peeks, which might or might not be representative. This value could have some kind of fallout value (for example a day or a week), such that it continues to be representative and a former peek does not continue to overshadow the current reality. The peek value can be used to help the developers decide on whether or not to scale the system, or rent extra computation in small periods of time, as it is for example seen possible with AWS[4] or Azure[5].

[4]https://aws.amazon.com/

[5]https://azure.microsoft.com/en-us/?b=16.48

3.2 B)

"Describe a meaningful view based on the data set from the Project 2 data set. How do you obtain that view? Describe the problems you faced obtaining such views in project 2 and how you fixed them."

I have chosen to showcase the second view from our Project 2 as I find that the most interesting. The entire report of Project 2 can be seen in appendix **??**. The view was described as follows:

"How can Wi-Fi data be used for tracking an individual at ITU?"

The hypothesis is that information about what access points a client has been connected to makes it possible to track a single person at ITU. Since each Wi-Fi client has an unique ID in the data set, tracking that ID around the ITU through various access points in certain rooms, one could connect this information to teaching activities. One could essentially build a schedule corresponding to a person, the holder of the unique ID, and by cross referencing the public course base, identify any student or teacher.

It is necessary to assume that every person is connected to the Wi-Fi whenever they are at ITU and even more important that they are connected to the access points in the rooms that they have courses in.

To obtain the data we created a map-reduce program. The main part of the analysis is done in a single mapper. The map method can be seen in figure 2. The mapper joins the reading with its WiFi Client if it can, then it filters the reading based on whether it measurement type is Access Point or not. Then for each specific reading, the mapper joins the reading with its Access Point. Then the output is defined as the ID of the Wifi Client as the key and a string representation of the location and time of that reading as the value. The mapper filters on null values for Access Points, and Locations.

```
1   public void map(AvroKey<Readings> key, NullWritable value, Context context
        ) throws IOException, InterruptedException {
2     Readings readings = key.datum();
3     // Where
4     if(wifiMap.containsKey(readings.getUUID().toString()))
5     {
6           // join
7       WifiClient wifiClient = wifiMap.get(readings.getUUID().toString());
8       // where
9       if(wifiClient.getTypeOfMeasure().equals(WifiClientMeasure.AccessPoint)
            )
10      {
11            // select
12        for(Reading reading : readings.getReadings())
13        {
14              // join
15          AccessPoint ap = apMap.get(reading.getValue());
16          // where
17          if(ap != null && ap.getLocationId() != null && locationMap.
                containsKey(ap.getLocationId()))
18          {
19            Date date = new Date(reading.getTimeStamp());
20            Location location = locationMap.get(ap.getLocationId());
21
22            context.write(new Text(readings.getUUID().toString()), new Text(
                  location.getRoom() + "-" + dateFormat.format(date)));
23          }
24        }
25      }
26    }
27  }
```

Figure 2: The map method for batch view 2 in project 2.

The reducer simply aggregates the rooms, times together to a list over each specific WiFi client.

A major critique point of this batch computation is that it only consists of one mapper, and therefore does not use the map-reduce framework to its full potential. It would have been a better idea to split this into multiple mappers, for example the for loop on line 12-24 could have been calculated in another mapper for that job. Similarly all filters could have been each mapper, and then chained together. This could greatly increase the concurrency and therefore the scalability of the batch job.

A snippet of the resulting view can be seen in figure 3

```
...
fd958189-5ad3-5586-a7ad-d3fe4e6f4695
 4A32-2016-10-12:11, AUD44A60-2016-10-24:08, AUD44A60-2016-10-24:09,
 AUD32-3A56-2016-10-04:09, AUD32-3A56-2016-10-04:08, 5A60-2016-10-12:07,
 4A58-2016-10-24:08, AUD44A60-2016-10-10:09, AUD32-3A56-2016-10-04:10,
 3A12-2016-10-06:11, 3A12-2016-10-06:12, 5A07-2016-10-12:11,
 AUD32-3A56-2016-10-13:09, 5A05-2016-10-31:11, 5A07-2016-10-12:10,
 3A52-2016-10-25:11, 3A52-2016-10-25:10, AUD44A60-2016-10-24:10,
 4A58-2016-10-24:09, AUD44A60-2016-10-31:10, 4A16-2016-10-24:14,
 5A07-2016-10-05:08, 4A16-2016-10-24:11, 4A16-2016-10-24:13,
 4A16-2016-10-24:12, 5A07-2016-10-12:08, 5A07-2016-10-12:07,
 AUD32-3A56-2016-10-25:10, AUD44A60-2016-10-10:10, 4A58-2016-10-24:10,
 5A07-2016-10-12:09, 4A16-2016-10-10:12, 4A16-2016-10-10:11,
 4A16-2016-10-10:14, 4A16-2016-10-10:13, 4A22-2016-10-31:13,
 4A05-2016-10-12:11, AUD32-3A56-2016-10-06:09, AUD32-3A56-2016-10-13:10,
 5A07-2016-10-05:09, AUD32-3A56-2016-10-25:09,
...
```

Figure 3: Resulting data

This data can be represented a bit better visually in a schema as seen in figure 4.

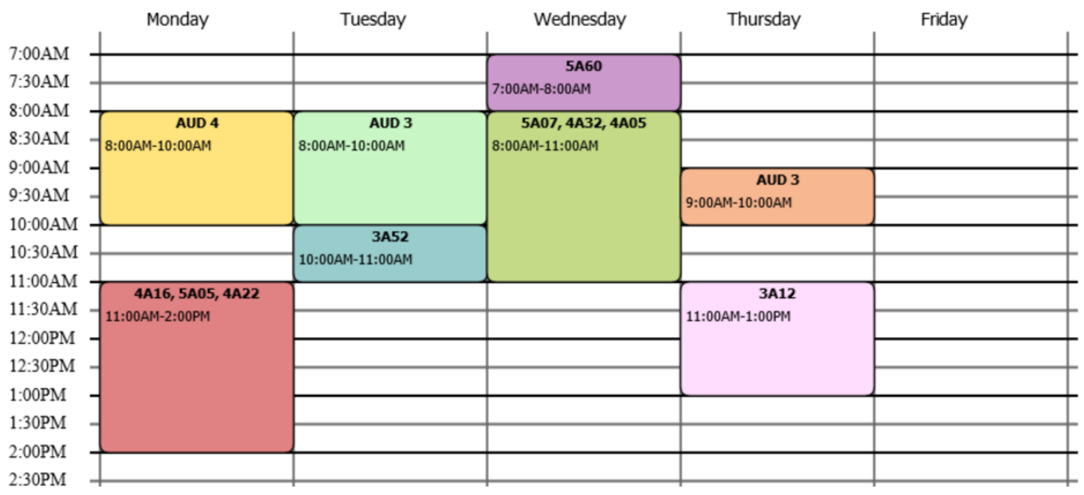
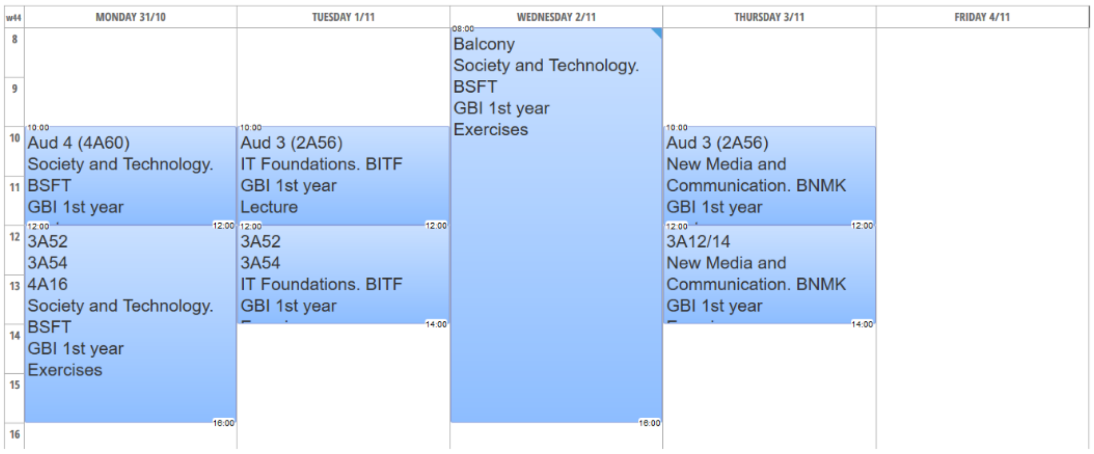


Figure 4: Resulting Schema

With these results, the schema of a WiFi Client can be correlated to the schema of a student at ITU. By matching which rooms at which times a Wifi Client has been to with a student's schema, we might be able to specify which person, or at least the programme of a student, a specific WiFi Client matches. For our result snippet shown above, the most plausible result is that the WiFi Client corresponds to a 1st year GBI student, since their schemas overlap nicely(see figure 5).



w44	MONDAY 31/10	TUESDAY 1/11	WEDNESDAY 2/11	THURSDAY 3/11	FRIDAY 4/11
8			08:00 Balcony Society and Technology. BSFT GBI 1st year Exercises 16:00		
10	10:00 Aud 4 (4A60) Society and Technology. BSFT GBI 1st year 12:00	10:00 Aud 3 (2A56) IT Foundations. BITF GBI 1st year Lecture 12:00		10:00 Aud 3 (2A56) New Media and Communication. BNMK GBI 1st year 12:00	
12	12:00 3A52 3A54 4A16 Society and Technology. BSFT GBI 1st year Exercises 16:00	12:00 3A52 3A54 IT Foundations. BITF GBI 1st year 14:00		12:00 3A12/14 New Media and Communication. BNMK GBI 1st year 14:00	

Figure 5: Timeedit Schema for a 1st year GBI student

One of the most difficult parts of creating this batch view was handling the difference between WiFi Clients and Access Points, and in particularly how readings should be mapped to these. The design we chose was to use Map-Reduce over the readings, and make the mapper read in all the meta data into memory in the beginning of the job. Then when mapping the readings the meta data is looked up in two local list, one for WiFi clients and one for Access Points. We decided on this approach since the meta data file was of limited size and did not grow as quickly as the readings data. But this approach is not the most effective and if we had for example used Hive and imported each of the entities into three seperate tables. The process of joining meta data to readings would then have happened in a concurrent fashion. This is one of the clear advantages of leaving the responsibility of distribution logic to the frameworks.

We also spend some time figuring out how to use the Avro serialization framework and how to integrate Avro with Map-Reduce. Avro uses a JSON like schema to define the binary format. Defining a schema with lists of lists was especially difficult, which was the case for the reading entity. Furthermore we needed to define two Avro schemas for the different types of data; one to the data before cleaning and one after it was cleaned and transformed.

References

[1] Facebook pushes the limits of hadoop. `http://www.infoworld.com/article/2616022/big-data/facebook-pushes-the-limits-of-hadoop.html`, author= Andrew Lampitt .

[2] Official website for apache flink. `https://flink.apache.org/`.

[3] Stephan Ewen Kostas Tzoumas and Robert Metzger. High-throughput, low-latency, and exactly-once stream processing with apache flink. `http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink/`.

[4] Neha Narkhede. Real-time stream processing: The next step for apache flink. `https://www.confluent.io/blog/real-time-stream-processing-the-next-step-for-apache-flink/`.