IT UNIVERSITY OF COPENHAGEN

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(from here: 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. Flink uses Kapfka, which is a stream gathering framework based on HDFS, to get its stream of events from[3].

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. Opposed to working with data in rest, where the partitioning is based on the data, Flink works with streams and as such, partitions the computations instead and sends the events between such computation nodes. Flink therefore tries to optimize the placement of the operations, 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 fallback mechanism. The state of the nodes is periodically persistent on HDFS or in memory, such that in case of a failure replaying from that checkpoint is possible. The system is based on the snapshot algorithm by Chandy and Lamport, such that the checkpoint is consistent across distributed nodes, and it is not necessary to duplicate information. The computation between checkpoints either succeeds or fails as a whole, and is then persistet as the new checkpoint. The approach also handles the problem that is, if a computation fails, on an infinite stream, it is not feasible to recalculate the entire computation again. Once all data has flown through the barrier/checkpoint the computation is done. The flow of the events happen in a DAG like structure which also means that the consistency is strong, since it is not possible to access an event which has not been through all the operations before that access[1].

This technique also separates the responsibility of flow control and throughput control, 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 stream analysis. Since Map-Reduce streams HDFS files to do batch processes, Flink has implemented batch processing as a special case of stream-processing, greatly simplifying the process. The only difference is that while streaming data is infinite, batch data is finite. The two API's can be used from Java or Scala, and provide a Java-Streams-Like interface, where it is easy to do typical SQL commands, such as where, grouping, sum and so on. Furthermore Flink has made it possible to easily define a window of the stream to allow for more sophisticated analysis.

Flink provides Pipelining to make nodes able to concurrently work on different tasks, even on different machines.

Throughput: Flink prides itself with being low latency 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 vertically allows for high throughput.

On data-artisan.com[1] 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 faster 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.

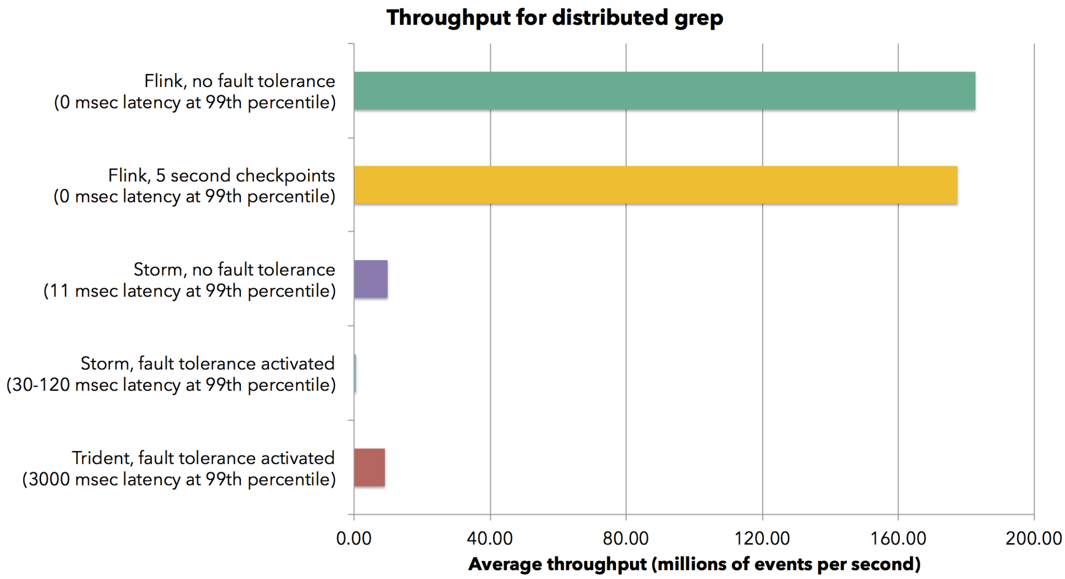

Figure 1: A graph showing the performance of different big data processing frameworks.

Interestingly enough Flink has a lot of comparisons to the Lambda architecture. The Lambda introduced in the course, is separated into the following stages, the data sources, the master dataset, the serving layer, the speed layer and the queries. At the serving 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. Further more, since Flink allows the same code for streaming analysis to batch analysis this would greatly increase the effectiveness of the developers. Flink also builds on top of HDFS and therefore automatically works with a distributed master dataset.

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,

[1] `http://data-artisans.com/wp-content/uploads/2015/08/grep_throughput.png`

and furthermore had the ability to with ease introduce a speed layer which could do similar calculations.

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

Transforming the data to JSON would mostly help with the amount of data, since JSON is less verbose than XML. JSON is still a tree-structure language and therefore partitioning would still be difficult especially because 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 flat data structure and therefore is easily partitioned per line and therefore allow map-reduce to work on multiple processes, greatly increasing performance. Furthermore CSV has the advantage that it is easy to extend the input with more columns.

Another possibility that we have used in Project 2 is to use a binary format. We used the serialization framework Avro[2] for this. By choosing to use a binary format, you can represent numbers as numbers therefore cutting down on the needed space. Furthermore if done properly it is still possible to make the format flat. 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.

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 seem 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.

In project 2 we stored the data directly on HDFS, and used Map-Reduce to make our views, and in this case using Avro serialization of the data worked quite well, since it allowed us to work with data as objects like any other Java program. Therefore it is difficult to conclude which specific approach should be taken and very much depends on the data and what the purpose of the system is.

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, which puts a question on what happens to 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 which, even though is not the same data, is equal to the primary data, and therefore just keeping the CSV and not the primary data is enough. If it is not possible to derive the original data from the master data set, then the approach to storing the primary data should be up to the goal of the data. If the goal of the system can change then it could be important to have a backup of the data, but if the goal is clear cut then it might not matter.

check om det her overhovedet er ok

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

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 structured boxed view, the big data movement tries to break these boundaries but it is still 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 it self as easy to use, but in my experience most of these systems still have a high learning curve. Furthermore 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 entrypoint for organizing and managing a large amount of the different hadoop frameworks.

Another con is that integrating different frameworks is often difficult. Since there does not exist a standard often times 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 out how and if it is possible themselves.

For small systems, or embedded systems where it is known that the amount of data will never surpass a low limit, the overhead of using big data technologies is also often not worth it. Then using relational database systems, can be enough

A lot of the frameworks have a lot of overhead on what they do, even though they scale better. If you know you will never store a lot of data, or want it to be available on the device of the user, going with a SQLlite or a system specific Relational Database would be smarter.

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 save through the 2000s and now that processors were nearing their clock speed limit, being able to to scale systems vertically 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 arbitrary amount of machines. Even in the case of machine breakdowns the frameworks of Hadoop will handle it and be able to replay, reroute, or abandon the process, and the developers are able to specify which approach should be taken as to how to restore the data of that node.

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 to use as input and outputs it in a certain way as well. One could have very generic mappers and reducers which in combination could have had the same effect and could have been put together differently to match this project.

We used streaming to clean the data in Project one so in some sense it would be possible to reuse the streaming part, since when streaming it is possible to handle the 80GB of data in Project 3, but with some other logic on what data to remove, label or ignore.

One of the kinds of cleaning that would need special development for 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.

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 positive or 0. Then checking whether or not each value has a type, such as 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. Another more difficult part of cleaning the data would be to check for the teleporting vehicles. This would require some table of information on where each car was last measured and if the distance from that point to the current point was to far, then the next entity should be labelled something saying which would make it possible to handle this in the analysis.

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 pipelining fashion. By doing this it is possible for map reduce to parallize as much of the process as possible. 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 disadvantadge 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?"*

The data of the stream contains structured data since it is in XML format. The structure 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. This information, since it is in XML, is easily obtained since the structure of the data is part of the data itself. Had the data been in JSON format it would be more difficult, albeit not impossible to find this structure, and had the data been unstructured it would have been even more difficult, since one should try to create and fit a schema at the same time.

Furthermore to describe the data stream one could examine the number of discrete elements that are present over timesteps in total or an average of that. Given these values it becomes obvious that we need big data systems to handle the stream. This could be done either with batch jobs, but it could also be done by making windowed stream analysis. By examining a window in the stream one could approximate the average amount of entities pr timesteps. This approximation of the overall mean 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 whether or not to scale the system.

Another characterization could be the peek amount of elements in the stream, which could be calculated by checking the last value and seeing if the currently calculated mean is higher than that. By using the mean instead of hard values, the computation is less sensitive to very short peeks, which might or might not be desired. This value could have some kind of fallout value (for example a day or a week), such that it continues to be representative. The peek value can be used to help the developers decide on whether to scale the system, or rent extra computation in small periods of time, as it is for example seen with AWS or Azure.

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 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 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 mapper. The map method can be seen in figure **??**. The mapper joins the reading with its WiFi Client if it can, then it filters, the reading based on whether it measures the Access Point. Then for each reading, it joins it with the reading with its Access Point and outputs the WifiClients id as the key and the location and time of that reading as the value. The mapper filters on null values for Access Points, and Locations.

```
public void map(AvroKey<Readings> key, NullWritable value, Context context) throws
     IOException, InterruptedException {
  Readings readings = key.datum();
  // Where
  if(wifiMap.containsKey(readings.getUUID().toString()))
  {
        // join
    WifiClient wifiClient = wifiMap.get(readings.getUUID().toString());
    // where
    if(wifiClient.getTypeOfMeasure().equals(WifiClientMeasure.AccessPoint))
    {
          // select
      for(Reading reading : readings.getReadings())
      {
            // join
        AccessPoint ap = apMap.get(reading.getValue());
        // where
        if(ap != null && ap.getLocationId() != null && locationMap.containsKey(ap.
            getLocationId()))
        {
          Date date = new Date(reading.getTimeStamp());
          Location location = locationMap.get(ap.getLocationId());

          context.write(new Text(readings.getUUID().toString()), new Text(location
              .getRoom() + "-" + dateFormat.format(date)));
        }
      }
    }
  }
}
```

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

It becomes obvious that having this as one mapper does use the map-reduce framework to its full potential. It would have been a better idea to split this into multiple mappers, for example in the situation where the mapper iterates over the readings, it would have been smarter to use a mapper for that job. This could greatly increase the parallelism and therefore the scalability of the batch job.

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

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

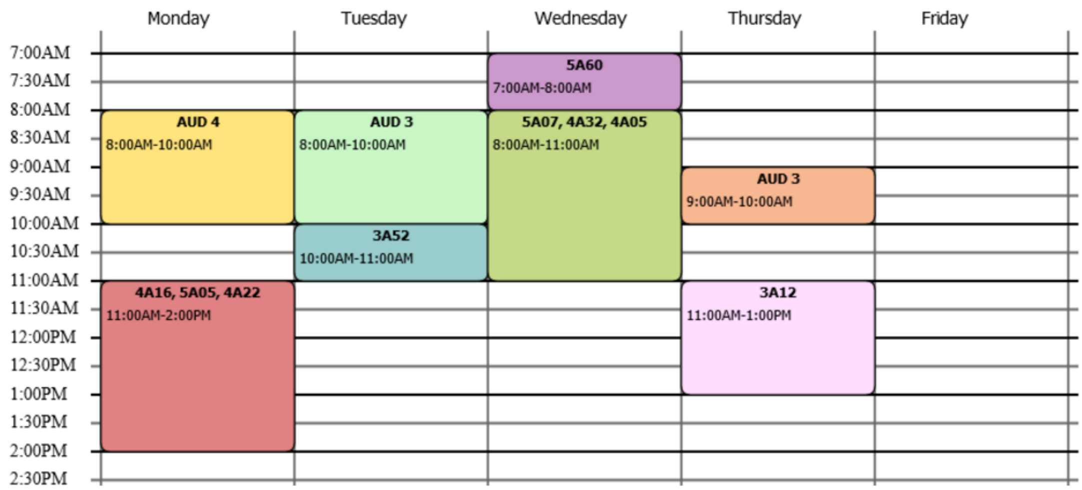

Figure 4: Resulting Schema

With these results, the schema of a Wifi Client can be correlated to the schema of a student at ITU by matching it with course information and TimeEdit. If the Wifi Client has been to the rooms that match a student's schema then we might be able to specify which person, or at least which programme, 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(see figure 5) overlap nicely.

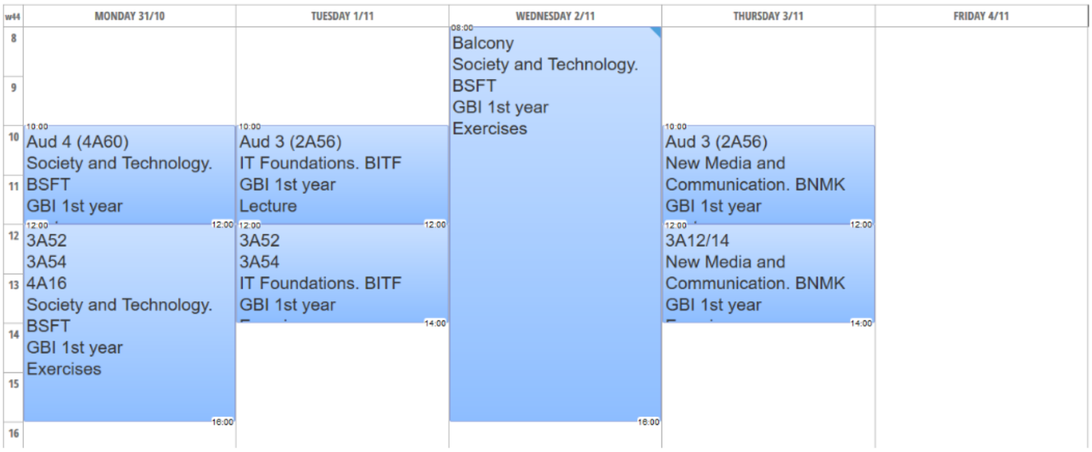

Figure 5: Timeedit Schema for a 1st year GBI student

One of the most difficult parts of creating this batch view was handling the difference between WiFi Clients and Access Points, and 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 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 fast as the readings data, of which there came one new file each day. But this approach is not the most effective and if we had for example used Hive, the process of combining meta data to readings would have happened in a parallized fashion.

We also spend some time figuring out how to use Avro serialization and how to integrate Avro with Map-Reduce. Avro uses a JSON like schema to define the binary format, the data is in. Making lists of lists was especially difficult which was the case for the readings, and furthermore we needed to define two Avro schemas for the different types of data, before and after it was cleaned and transformed into the different types.

References

[1] 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/`.

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

[3] 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/`.