SUBMISSION OF WRITTEN WORK

IT UNIVERSITY OF COPENHAGEN

Class code: *SBDM-Autumn 2016*
Name of course: *Big Data Management (Technical)*
Course manager: *Philippe Bonnet*
Project title: *Exam Assignment*

Full Name: *Irina Alina Gabriela Luca*
Birthdate: *24/05-1993*
E-mail: *irlu@itu.dk*

Table of Contents

Question 1 – Systems and Data Models (40%)

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.

A.1. Data Platform Characterization

A.1.1. Data model and programming abstractions

Apache Flink represents an open-source data platform, with programming abstractions in Java and Scala, which proposes two fundamental APIs: the DataSet API for batch processing and the DataStream API for stream processing. Additionally, Flink targets domain-specific libraries and APIs, including machine learning (FlinkML), graph processing (Gelly) or SQL-like operations (Table API & SQL). Flink's process model is emphasized by three main players: the client turns the program into a dataflow graph, which is further on passed to the JobManager that will coordinate and distribute the data-flow through one or more TaskManagers, using streams and operators. Flink provides huge flexibility in terms of access to many systems as data sources/sinks, working, just as Hadoop, with the notions of 'InputFormat'/'OutputFormat', and offering Avro support, as well. Flink manages to look like a batch processor, along with its real-time stream processing nature, but still providing low latency. Batch processing is actually considered a special case of Flink's streaming computation model[1], but with a wide streaming window. Generally, Flink executes tasks through an iterative and cyclic data flow, supporting DAG operations. Flink relies on an algorithmic technique introduced by Chandy and Lamport in 1985, so that it regularly saves streams' state snapshots to a form of durable storage (HDFS, for instance), achieving a combination of flow control, true streaming programming model, high throughput, low latency and very low overhead for its fault-tolerance mechanism.

A.1.2. Partition management

A. Sharding

Flink's data-flow is executed in a parallelized manner, such that operators are split into parallel subtasks and streams are parallelized into stream partitions. Generally, Flink performs hash-partitioning, with sorting in the reduction step. Data sets are joined with two strategies - the 'ship' and 'local' strategies. In the first step, Flink can use either the Repartition-Repartition strategy (all the data sets are integrally shuffled across the network) or the Broadcast-Forward strategy (only some data sets are sent to the parallel partitions). In the second step, each parallel instance executes a local join (Sort-Merge-Join/ Hybrid-Hash-Join strategies). Choosing between these strategies depends on how many partitions there are or how big each involved data set is.

B. Resource Management

Flink's distributed execution involves the interaction between a client, the master processes (JobManagers) and the worker processes (TaskManagers), which can effectively be managed through resource frameworks like YARN, for example.

[1] Reference [46] - DataArtisans, Apache Flink: Batch is a special case for streaming

C. Job Scheduling

Flink performs job scheduling through TaskManagers that can execute parallel tasks through pipelines. The JobManager becomes responsible for task distribution and coordination, turning the JobGraph it receives into an ExecutionGraph, associated with a job status that describes the current state of the job execution. Flink uses a master-slave architectural pattern.

D. Failure Handling & Replica Consistency

Flink's checkpoint mechanism becomes important for failure recovery in a streaming situation, since Flink effectively checkpoints the state of the utilized operators and restores it, instead of reacting upon the whole input. A batch computation failure is handled by recomputation of intermediate results, so that all the needed transformations (e.g.: reduce, reduceGroup) are reevaluated. Flink generally makes use of 'strict exactly-once processing guarantees'[2], by maintaining distributed consistent snapshots, which offers strong consistency guarantees.

A.2. Apache Flink in the context of Lambda Architecture

If I were to place Apache Flink in the context of Lambda Architecture, I would need to underline some of the limitations the Lambda Architecture imposes, in comparison with the powerful computational model proposed by Apache Flink. Back in the days, the stream processing could not yield precise results or cope with a high volume of events, but nowadays, Apache Flink provides both accurate and fast results. Therefore, there is no more need to choose between low latency and high throughput, because Flink handles both effectively, so the multilayer proposed by the Lambda Architecture can be perceived from a possibly more mature perspective. One way to cope with the high latency issue, earlier underlined by the Lambda Architecture, was to couple a stream processor with the batch architecture. Nevertheless, several other issues still remained to be addressed, such as out-of-order event handling, flexibility in changing system characteristics, components administration and configuration etc. Eventually, Apache Flink comes up with a very potential streaming architecture, which, if used along with a message queue (maybe with Kafka), for instance, would yield better workflow and performance and fix many of the above mentioned issues, which remain unaddressed in the context of the Lambda Architecture.

A.3. Apache Flink: Comparison with systems used during the projects

Both IBM Cloudant and Hadoop (used during the projects) seemed appropriate for the given requirements and comparing Apache Flink with them is not meant to be a direct parallel, because Apache Flink is conceptually different, and can be used in other contexts than the ones we have worked with so far. In the data platforms' map [3], all three systems are situated in the non-relational zone, which is a similarity. However, IBM Cloudant is placed very far away from Apache Hadoop and Apache Flink on the map, since it is a DBaaS system, which provided us with an easy-to-use API

[2] https://www.user.tu-berlin.de/asteriosk/assets/publications/flink-deb.pdf
[3] http://www.csee.umbc.edu/~kalpakis/courses/661-fa15/papers/data_platform_map_june_2015-1.pdf

and map-reduce functionality. We stored our JSON documents in the cloud, avoiding additional installation and creating the needed batch views with an easy workflow for Project 2. Project 3 also involved creating batch views, though we used Hive on top of Hadoop for that purpose. Even if Flink can not only run in standalone mode, but can just as well be combined with Hadoop, and many components from the Hadoop ecosystem (YARN or HDFS, for instance), it still primarily draws attention to its streaming capabilities, regardless its additional features or the Kappa architecture. However, Flink can directly compare to MapReduce, since it can potentially be an alternative to Hadoop's MapReduce and is actually often used in such a setup, together with Hadoop's HDFS. There can be many reasons to it, amongst which also the fact that Flink proposes a unified framework, building a data flow for batch, streaming, machine learning, SQL-like querying etc. and providing a good fault-tolerance mechanism, iterative stream processing and so on. Flink also includes an optimizer for batch computations and relies on its pipelined engine (hybrid, in fact), compared to Hadoop's MapReduce, which is based on a batch engine and therefore is slower. Compared to MapReduce, which Flink's programming model relies on, in fact, Flink introduces a lot more transformations, as well (Iterations, Join, CoGroup etc.), and a more generalized key-value pair model, also decreasing the need to materialize intermediate results, as specified before (pipelining). Since in Flink's terminology, batch is perceived as a special case of streaming and stream processors like Flink can handle at least just as big workloads as batch processors, but with even lower latency, one can assume that Flink proposes a more powerful model than Hadoop and its MapReduce functionality, thus augmenting the batch architecture with a real-time streaming one.

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.

Hadoop processes a data file in blocks and therefore, no matter the input file, its splitability becomes a key property when storing the master data set. An XML file does not provide flexibility with regards to splitability, because of its hierarchical structure (it contains a beginning tag and an ending tag, as well as complicated nested tags). Compared to XML, CSV can be split at any point of the file, because the data would still have semantic value. Considering that our 80 GB does not fit in the default size of an HDFS block, then splitability matters, so that Hadoop can distribute the data and process it in parallel. Moreover, one cannot just read the entire XML 80 GB file into memory at once, which is also the case we faced in Project 3, where the large XML file contained <timestep> tags nesting both <person> and <vehicle> tags. The solution was to implement a Java program that parsed through the XML file once and turned every person/vehicle into a CSV record, containing the corresponding timestep. From this previous experience, one might say it helps turning the large XML file into CSV files ('person.csv' and 'vehicle.csv', in our Project 3), stored in HDFS and served to Hive in the end with a command like "load data inpath 'csv/person.csv' into table person;", which are, therefore, easy to work with. Basically, there would be no need to go back the XML file, because this

kind of derivation process did not affect the actual data set in any way and thus, it can still be considered primary data.
Additionally, MapReduce's InputFormat does not provide support for XML and the parsing process might be too CPU requiring. When MapReduce runs, it firstly localizes the input file and then it processes and splits it (using RecordReader, InputFormat, InputSplit). Choosing the right approach to map-reduce the XML input depends, for instance, on whether we deal with XML files that are larger than the normal HDFS block size (128MB) or not. Since our file is extremely large, it would create multiple blocks, distributed and replicated across the cluster. However, a solution to process the 80GB of XML data and even larger datasets would be to use Mahout's XMLInputFormat, which would yield lower memory consumption and better processing speed. Mahout's parsing process is handled by identifying records by an XML start tag, respectively XML end tag. On the other hand, I believe a transformation from XML to JSON would not be helpful either. Serving JSON file as input to MapReduce can be just as problematic, because one cannot ensure that it can be partitioned properly for concurrent reads, and MapReduce's InputFormat does not provide support for JSON either. Moreover, a JSON file's hierarchy rises the issue that the same property name can be encountered in several blocks and even worse, there are not even a start and an end tag that can make segmentation somewhat easier. It would work if the input file would have a particular JSON structure (each JSON object stored on a line, for instance), but that would also pose new questions.
Nonetheless, using Mahout for this task would introduce extra complexity, and therefore, my solution would still be to use a small Java parser, like we did in Project 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, consisting of two fundamental layers such as MapReduce and HDFS, has been playing a significant role in the big data world. The considerations below will specifically rely on the experiences acquired during Project 3, since this is where we opted for a Hadoop ecosystem solution. Given a large XML file containing data about vehicles and people correlated with particular timestamps, the main task in Project 3 was to compute three batch views that provide insights regarding the dataset. Since the Hadoop ecosystem is extremely large, I will narrow down the scope by particularly identifying pros and cons for HDFS and Hive.

C.1. HDFS

After generating our CSV files ('person.csv' and 'vehicle.csv'), each occupying approximately 23 GB (see Appendix [1]), we stored them in HDFS. Conveniently, HDFS offers high bandwidth to perform mapreduce computations and represents a cheap storage option. Experiencing HDFS proved the ease to work with it. However, just as the CAP theorem has its own challenges, the Namenode in HDFS is responsible for the keeping track of file storage and distribution across the nodes and becomes a single point of failure, such that storage would have been in danger if the Namenode went down.

C.2. Hive

In order to produce the three batch views in Project 3, we decided to use Hive, which provides an SQL abstraction layer over Hadoop's MapReduce, through HiveQL/ HQL. Creating the batch views through Hive seemed very advantageous during Project 3, for multiple reasons:

- It enabled us to easily use HiveQL and apply or previous SQL background, since HQL simply compiles to MapReduce jobs;
- Loading the CSV files into Hive tables was very fast, regardless the large size of our input;
- We initially considered the possibility to integrate Hive with HBase and even if the idea did not materialize in the end, it still provides flexibility to the implementation;
- The HiveQL queries that we worked with during the whole process were easy to understand and write, needing a lot less effort than writing MapReduce programs.

However, while working on Project 3, using Hive also seemed tricky in some ways:

- Some MapReduce jobs took quite a long time to execute, depending, of course, on what was required by the query. While computing the average global speed for people took very little time, returning people's distribution in a time range on a chosen street took a lot longer;
- With some of the queries, HQL seemed to be more restrictive than SQL, especially with subqueries or some of the utilized clauses. In the query below, the idea was to show streets (edges) where average speed is lower than the global average speed, which we plugged directly from a previous query. Since we used the 'group by' clause, we were restricted to aggregate the coordinates 'x' and 'y', and we chose max() for it[4]: 'select edge, avg(speed), max(x), max(y) from person where speed < 0.9283566686669638 group by edge;' ;
- Compared to Pig, which can store intermediate data, Hive was not as easy to debug.

All in all, the Hadoop ecosystem is huge and all its players come with pros and cons, but the above mentioned ideas correlate with the experiences gained during Project 3, in particular.

Question 2 – Data cleaning (30%)

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.

Essentially, the process of data cleaning represents identifying data outliers and missing data. However, the context in which data cleaning is performed becomes important, as well. One might underline two phases where cleaning techniques could be applied: firstly, the phase of storing the master data set and secondly, the phase of creating batch views from the master data set. However, maintaining the data as primary/raw as possible, instead of deriving it, would be beneficial. As Nathan Marz mentions, 'the rawer your data, the more questions you can ask of it'[5]. While investigating the data in Project 2, we found several aspects where we slightly adjusted the data set before storing it in

[4] http://stackoverflow.com/questions/5746687/hive-expression-not-in-group-by-key

[5] 'Big Data - Principles and best practices of scalable real-time data systems', by Nathan Marz and James Warren, page 31

the IBM Cloudant system, as enumerated below, even though afterwards, defining the map/reduce functions in Cloudant implied some cleaning operations, as well:

- We evaluated data uniformity and naming conflicts. For example, we turned pairs of values such as (‘unknown’, ‘Unknown’) or (‘dB’, ‘dBm’) into one form, because the semantics would not be affected and the data set clarity would slightly increase;
- We investigated data completeness and tagged with ‘Unknown’ where the data could not be used for further analysis. We did not drop the data, because we wanted to keep it in its original form as much as possible;
- We found redundancy through duplicate timestamps, but decided to store them in the system, just as well as we did with the outliers, because they might provide valuable insights. For example, duplicates of the same timestamp in Project 2[6] might signal a defect access point;
- We evaluated gaps in the timestamps, which could represent data loss, but stored them in their primary form and revealed the gaps by computing a batch view afterwards. Nonetheless, we still used tags when we stored documents in IBM Cloudant, indicating that the respective document might be inconsistent in some way, which allowed us to later decide which kind of data to take into account for computing the batch views;
- For the devices in the given dataset, we aggregated their OS and vendor when creating a batch view to show which devices are used in ITU, considering that in our case, the modification would be useful, because it would decrease the number of unknown devices from 15000 to 7500 and would not affect further analysis either.

However, in Project 3, we used a different approach in order to clean the data. First of all, we stored our master data set as raw as possible, so that no derivation would affect the primitivity of the data. When we created the Java parser to translate the large XML file into CSV files and load them into Hive’s infrastructure, we transferred the data in the exact same form it was given, thus considering that the cleaning process would be done while computing the batch views (throughout aggregations, data transformations, filtration, statistical mechanisms etc.) and would depend on what insights we want to obtain. To be precise, I will exemplify on a specific batch view that can be built from the data set in Project 3, where we worked with data concerning the traffic in Copenhagen for a particular time range, the only subjects being vehicles and people. The general ideas from Project 2 could be reused, but the approach changed, according to the different requirements and the data set. As such, the cleaning process in Project 3 involved the following:

- We started with data analysis, so we extracted a sample from both ‘person.csv’ and ‘vehicle.csv’ and injected it on a local installation of DB Browser for SQLite. Consequently, we could acquire metadata regarding the data properties and identify data quality issues. It also helped us realize what data types were most suitable to work with when inserting the data into Hive;
- In order to create a batch view that provides an estimation about the often most crowded areas in Copenhagen, we opted for counting the distinct people on a street (edge) and since we could not identify the ‘x’, ‘y’ coordinates of a street by itself, we picked the average

[6] where we worked with ITU’s wifi data and measurement values from different access points

coordinates of all the people on a particular street. Further on, we aggregated by 'edge' ('group by') , to get the result for each unique street: 'select avg(y), avg(x), count(distinct(pid)) as e_cnt from person group by edge order by e_cnt desc;'. In this case, duplicate values of the geographical coordinates would not affect the result, for instance, because we pick the average and at the same time, we were also aiming for an estimate, which is exactly what we obtained.

All in all, the cleaning approach in Project 3 differed from the one in Project 2, since the context was different and the data transformations needed in a cleaning process also depend on the stated requirements for what insight/ results one must obtain from the data set.

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

One approach to perform cleaning on the data set from Project 3 could be to use Apache Pig, a high-level scripting language well-known for its data analysis capabilities and especially for Extract-Transform-Load (ETL) processes, data cleaning and exploration. The thought of choosing Pig for this purpose would be based on several factors, as it follows:

- Since Pig belongs to the Hadoop ecosystem, the data validation process would be performed in the same environment, so extra complexity/external systems would be avoided;
- Pig comes with User-Defined-Functions (UDF) that can be created in multiple programming languages (Java, jRuby, Python etc.) and easily integrated with Pig Latin. UDFs become useful in data cleaning because they enable passing the data to more complicated algorithms or programs that perform data transformation;
- Pig provides operators such as 'filter', 'join', 'sort' etc., which would be suitable for cleaning.

The cleaning process would involve eliminating inconsistent records from the data set. Let us consider an example on the data set from Project 3, where we establish some business rules, and based on them, we perform the data cleaning. If we want to provide runners in Copenhagen insightful info about what streets are the most suitable to run on, we would therefore try to extract the streets/edges from the given data set where the average speed exceeds the global average street. Based on this decision, we consider all the records that do not comply with the rule, dirty, and therefore we use Pig to add a filter in the data flow. In this example, the filter is not complicated, but Pig can help with much more complicated filter processing, as well. Consequently, a possible system used for data cleaning in Project 3 could involve the following phases:

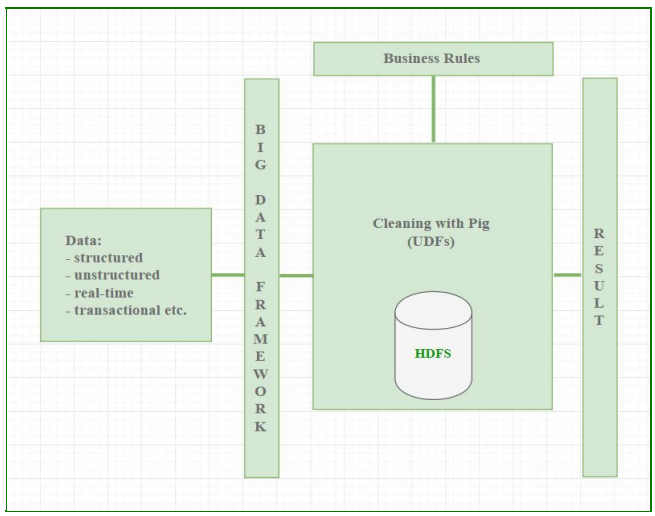

Figure 1: Data Cleaning, Pig Latin, UDFs

- Parse the given 80GB XML file and split it into 'person.csv' and 'vehicle.csv', just as before;

- Store the files from the filesystem to HDFS (e.g.: 'hdfs dfs -put /tmp/person.csv /user/irina/person.csv');
- Jump into Pig's Grunt shell and load both CSVs into PigStorage, by saving them in variables (e.g.: 'people = LOAD '/user/irina/person.csv' USING PigStorage(',') AS (pid:int, edge:chararray, x:double, y:double, timestamp:datetime [...]);');
- Create a Filter UDF, which depends on our cleaning requirements/pre-established rules;
- Pack the Filter UDF in a jar, 'register' it in Pig Latin and finally, filter the data.

This would be a very customizable manner to clean the data, since we make use of custom filters. ***Figure 1*** summarizes explicitly the above mentioned phases used for data cleaning.

Question 3 – Data exploration (30%)

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?

In order to characterize such a stream, one needs to identify:

- a schema representing the data stream;
- a description of the domain of attribute values that are present in the identified schema, or eventually, a statistical description of the values in that domain.

Our XML file structure involved timestep tags containing a collection of both vehicle and person tags. Let us consider the incoming data stream being composed of tuples/events. One could most likely identify the stream schema in several ways. One of the possibilities is to look at the incoming events as being mapped by the form '(<timestep>, <set of vehicles and people>)'. Eventually, another idea could be to try to sample a large window of the incoming stream and materialize it, but that would, again, represent only a partition of the whole stream.

However, looking backwards at how we parsed the XML once, with a Java program, and created 'person.csv' and 'vehicle.csv', I would consider the possibility where the stream schema involves events that are either mapped as '(<timestep>, <vehicle>)' or as '(<timestep>, <person>)', where a vehicle's schema is given by the attributes Vehicle(vehicleId int, x double, y double, angle double, type string, speed double, position double, lane string, slope double, timestamp double) and a person's schema is given by the attributes Person(personId int, x double, y double, angle double, speed double, position double, edge string, slope string, timestamp double). Schema definition plays a significant role in the data flow, because it provides robustness, flexibility and compatibility, making changes easier to propagate in the whole system. The above stream schema could, for instance, be applied in a Kafka-related system. Consequently, if a new Kafka topic is provided, the stream schema would define how the data flows into Hadoop and what is needed such that Hive creates an appropriate table. The more the schema develops, the more the metadata evolves, as well. The maintenance of a single schema for each Kafka topic would offer more flexibility when mapping it to Hive's infrastructure. Moreover, in this example, we could keep a global state for 'vehicle', as well as a global state for

‘person’, both states being updated with each incoming event and containing the timestep information, as well. When a computation is attempted, for instance the number of distinct vehicles with lower speed than a certain value, the result would rely on the vehicles’ global state composed until that moment. Moreover, the model could be built on a strategy where a summary of the desired computation is constantly maintained and updated with each event, instead of storing everything in active storage. Another option could also imply a sliding window that represents the most recent incoming data. By defining the stream schema as mentioned, there might also be a possibility to logically partition the original stream by item structure into two different substreams, one handling the ‘person’ event, and the other one handling the ‘vehicle’ event and their corresponding attributes.
Notably, compared to the massive data set, which can be passed through multiple times, the data stream is passed through only once and is characterized by a sequence of events. However, depending on the query one wants to run on the data set, a sampling technique using hash functions could also be appropriate. Let us consider a query which aims to study people’s behaviour or traffic dynamics in some sense, such that a stream sample could statistically be enough to answer the query for the entire stream instead.
All in all, depending on what results one aims to obtain from modelling a data stream, different techniques can serve for that matter (HyperLogLog, Bloom Filters etc.), but defining a valuable stream schema, along with suitable attributes plays an important role with regards to many aspects.

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.

The data set from Project 2 represents ITU wifi-related data, which one can derive a lot of information from, such as: devices that are used in ITU, the possibility of tracking them by ID (location-wise and time-wise), devices distribution in a particular area in ITU, booked rooms, types of OS used by the reported devices, how crowded a specific room is at a specific time etc. In order to create views in Project 2, we took advantage of the map-reduce functionality of the cloud-based database IBM Cloudant, where we created both a map and a reduce function for each and every view.
One meaningful view based on the data set from Project 2 could reveal how a specific person, tracked by ID, moves around ITU. ***Figure 2*** presents a JSON visualization of such a view, which identifies a subject by the key "034e81e193496eeacb3c5cd5[...]", possibly a hashed value of its ID, and lists rooms where the subject has been located. The rooms are also described by a numerical ID and the data is structured per days, along with the timestamp showing when the subject was located in each room. Moreover, the time when the subject came online is also provided.
In order to create this view, one needs to define a map function that only looks at the documents in the cloud that have ‘AP’ (access points) as a unit of measure. Furthermore, for each appropriate wifi-client, one needs to inspect its timeseries readings and group by what access point it is connected to at a given time. Adding the “os” and “vendor” can only help providing insight about which subject

```
{
    "key": "034e81e193496eeacb3c5cd5b27b5687603696e1dd1d2a18255bcf97403f0
    8690d5c9863fd2b81136c9856ad156ed111a000d6f758327bcd8c0d87e794e02f53",
    "value": [
        [
            {
                "extra": {
                    "date": "2016-10-13",
                    "cameOnline": "11:39:44",
                    "os": "Android",
                    "vendor": "Samsung Electro Mechanics co., LTD."
                },
                "rooms": {
                    "136": "17:14:12",
                    "186": "17:18:12"
                }
            },
            {
                "extra": {
                    "date": "2016-10-10",
                    "cameOnline": "08:35:33",
                    "os": "Android",
                    "vendor": "Samsung Electro Mechanics co., LTD."
                },
                "rooms": {
                    "25": "08:35:33",
                    "28": "14:59:43",
                    "34": "13:01:47",
                    "35": "13:15:47",
                    "58": "12:02:11",
                    "136": "14:53:36",
                    "138": "11:55:49",
                    "142": "13:17:47"
                }
            },
        ],
    ]
},
```


Figure 2: JSON sample showing tracking info about a subject (device)

type moves how (Windows/Mac/phone subjects). In fact, this could be useful for tracking purposes. For instance, more refinement could eventually give an overview regarding subjects' attendance to specific courses or at least an approximation. In order to visualize the contents of this view, we used Cloudant's built-in capabilities. That allowed us to see the whole JSON result in the browser, where we formatted it nicely with a Chrome Extension and took advantage of different parameters that we could use. For instance, the image above is a sample extracted from the entire JSON structure we obtained by accessing '/_view/get_rooms_from_client?limit=100&reduce=true&group=true' in the browser. Additionally, there were several difficulties in obtaining such views in Project 2. It was hard to understand what the different parts of the data could mean, but the more views one made, from the tiny ones, to the more complex ones, the more clear everything seemed to be. Testing whether the results actually provided what we aimed to get was sometimes difficult, especially in a system like Cloudant, where we had a window to insert our map function, respectively another one for the reduce function. The problem was coped with by testing each piece of the functions individually, or by splitting a more complex view into smaller ones and build up the result gradually. Naturally, our workflow of creating views as the one mentioned above was adjusted according to the system we used, Cloudant, which generally seemed extremely appropriate for solving the tasks in Project 2.

References

Books & Publications

[1] - Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, Kostas Tzoumas, Apache Flink™: Stream and Batch Processing in a Single Engine

[2] - Erhard Rahm, Hong Hai Do, University of Leipzig, Germany, 'Data Cleaning: Problems and Current Approaches'

[3] - Peter M. Fischer, Kyumars Sheykh Esmaili, Renée J. Miller, 'Stream Schema: Providing and Exploiting Static Metadata for Data Stream Processing'

[4] - InfoLab Standford, 'Mining Data Streams'

[5] - Shelan Perera* , Ashansa Perera* , Kamal Hakimzadeh SCS - Software and Computer Systems Department KTH - Royal Institute of Technology Stockholm, Sweden, 'Reproducible Experiments for Comparing Apache Flink and Apache Spark on Public Clouds'

[6] - Klemen Kenda, Jasna Škrbec, Maja Škrjanc Artificial Intelligence Laboratory Jožef Stefan Institute, 'Usage of Kalman Filter for Data Cleaning of Sensor Data'

Links & Articles

[7] - Alex Holmes (author of 'Hadoop in Practice'), Processing Common Serialization Formats

[8] - Data Partitioning Strategies in Parallel Database Systems

[9] - How to beat the CAP Theorem

[10] - Apache Tez

[11] - TechTarget, Apache Flink

[12] - CakeSolutions, Comparison of Apache Stream Processing Frameworks

[13] - InfoWorld, Apache Flink: New Hadoop contender squares off against Spark

[14] - Apache Flink, Jobs and Scheduling

[15] - Apache Flink, Peeking into Apache Flink's Engine Room

[16] - StackOverflow, Apache Flink - Failure Handling

[17] - DataArtisans, Stream Processing Myths Debunked

[18] - DataArtisans, Counting in streams: A hierarchy of needs

[19] - InfoQ, Stream Processing and Lambda Architecture Challenges

[20] - 451 Research, Data Platforms 2016 Map

[21] - Apache Flink, Hadoop Compatibility in Flink

[22] - DigitalOcean, Hadoop, Storm, Samza, Spark, and Flink: Big Data Frameworks Compared

[23] - Confluent, Real-time Stream Processing: The next step for Apache Flink

[24] - BigDataPage, Using MapReduce is Like Plumbing with Pre-Clogged Pipes

[25] - Metistream, Comparing Hadoop, MapReduce, Spark, Flink, and Storm

[26] - HadoopSummit, Overview of Apache Flink: The 4G of Big Data Analytics Frameworks

[27] - StackOverflow, How does Apache Flink compare to Hadoop's MapReduce

[28] - HadoopTutorials, Understanding Hadoop Ecosystem

[29] - BigDataCompanies, 5 Big Disadvantages of Hadoop for Big Data

[30] - J2EEBrain, Hadoop: Advantages & Disadvantages

[31] - HadoopDummies, 10 Reasons To Adopt Hadoop
[32] - Quora, MapReduce & XML
[33] - DataIntegration, Extract-Transform-Load
[34] - XPlenty, Hadoop ETL with Apache Pig
[35] - HortonWorks, How to process data with Apache Pig
[36] - Apache Pig, Pig - Filter UDFs
[37] - Apache Pig, Pig - Design Patterns
[38] - Big Data Processing, Pig Latin
[39] - Software Intel, ETL - Big Data with Hadoop
[40] - BigDataCompanies, 5 Big Disadvantages of Hadoop for Big Data
[41] - Confluent, Putting Apache Kafka to work
[42] - JavaCodeGeeks, Streaming Big Data: Storm, Spark and Samza
[43] - SamzaApacheOrg, Spark Streaming
[44] - Cloudant IBM
[45] - ConvergeBlog, Apache Flink: New Way to Handle Streaming Data
[46] - DataArtisans, Apache Flink: Batch is a special case for streaming
[47] - DataArtisans, High-throughput, low-latency, and exactly-once stream processing with Apache Flink™

Courses & Tutorials

[48] - Hadoop Summit, 'Efficient processing of large and complex XML documents in Hadoop'
[49] - Pluralsight Courses Platform, Ahmad Alkilani, 'Applying the Lambda Architecture with Spark, Kafka, and Cassandra'
[50] - Pluralsight Courses Platform, Thomas Henson, 'Pig Latin: Getting Started'
[51] - Pig ETL, ETL on Hadoop: Using Pig for log file Analysis on HDInsight Azure

Appendix

[1] - Project 3, CSV files after parsing the large XML file

```
[group03@H0 csv]$ ls -l vehicle.csv
-rw-r--r--. 1 group03 group03 23250401604 Nov 19 16:48 vehicle.csv
[group03@H0 csv]$ ^C
[group03@H0 csv]$ ls -l person.csv
-rw-r--r--. 1 group03 group03 23525992219 Nov 19 16:48 person.csv
[group03@H0 csv]$ hdfs dfs -ls
Found 9 items
drwxrwx--x   - group03 hdfs          0 2016-11-20 18:00 .Trash
drwxr-xr-x   - group03 hdfs          0 2016-11-01 09:24 .hiveJars
drwx------   - group03 hdfs          0 2016-11-27 21:03 .staging
drwxr-xr-x   - group03 hdfs          0 2016-11-20 10:16 csv
drwxrwx--x   - group03 hdfs          0 2016-11-20 10:10 person
-rw-r--r--   1 group03 hdfs 23525992219 2016-11-23 20:58 person.csv
drwxrwx--x   - group03 hdfs          0 2016-11-21 22:16 person_small
-rw-r--r--   2 group03 hdfs        599 2016-11-27 12:08 testperson.csv
drwxrwx--x   - group03 hdfs          0 2016-11-20 10:16 vehicle
```