Fil Rouge : Progress Report

Real time train delay prediction

February 6, 2017

Team:

Sami Bargaoui
Mohammed Benseddik
Léonard Binet
Yassine Errochdi

Tutors :

Maguelonne Chandesris, SNCF
Xavier Chapuis, SNCF
Keun-Woo Lim, Telecom ParisTech
François Roueff, Telecom ParisTech

2016 - 2017

Contents

List of Figures

Chapter 1

Introduction and context of the project

1. Introduction

"What if we could have a prediction of the arrival of trains in real time? "

It is a question which imposes itself as a real problem for the daily SNCF commuters. And it can be quite a frustrating problem. We would all like to have a certain prediction of arrival time on our Phones through an Application when there are some issues or disturbances in the traffic. The prediction may not be that precise, but it would change the way we manage our time while waiting on the train station.

Figure 1.1.: Illustration of Train delays.

SNCF (Société nationale des chemins de fer français) is actively working on this problem within the Network Department. This had lead to the project is being carried out by both the SNCF, providing open source data through their API (Open Data), and by a team of Télécom ParisTech's Big Data Masters students.

2. Objectives & Context of the project

2.1. Global objectives

- Extract and use data from SNCF Open Data Platform.
- Predict train arrival time with delays in real time.
- Challenge the existing implementation at SNCF.
- Deliver a functional application that integrates the data pipeline and prediction model.

In synthesis, the main challenge of the project is to collect the actual transit times of the SNCF trains from the data of the API, in order to produce accurate and real-time delay prediction at a given station, for a given train and a given time.

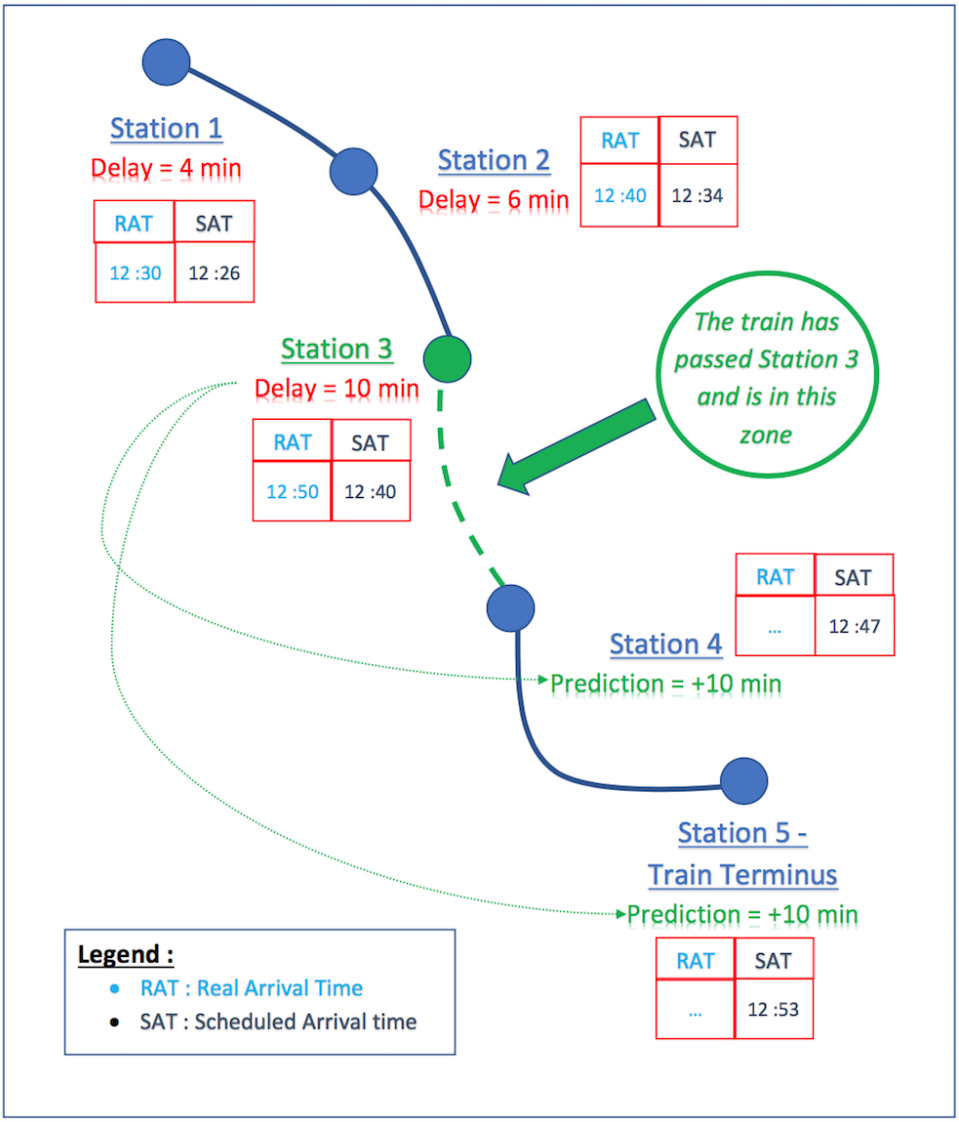


Figure 1.2.: Main Objective of the Project.

2.2. Business & Human context

The project is carried out by the SNCF network department, particularly by the R&D team.

Our company supervisors :

- *Maguelonne CHANDESRIS*
- *Xavier CHAPUIS*

School supervisors :

- *Keunwoo LIM*
- *Francois ROUEFF*

Team members :

- *Sami BARGAOUI*
- *Yassine ERROCHDI*
- *Léonard BINET*
- *Mohammed BENSEDDIK*

2.3. Data Sources

For the project's purposes, we are using the public API provided by SNCF. It is an Open Data initiative from SNCF to give, in real time, informations about the railway network including departure times of all the trains in France. We also extract planned and scheduled times from GTFS files available at SNCF's website.

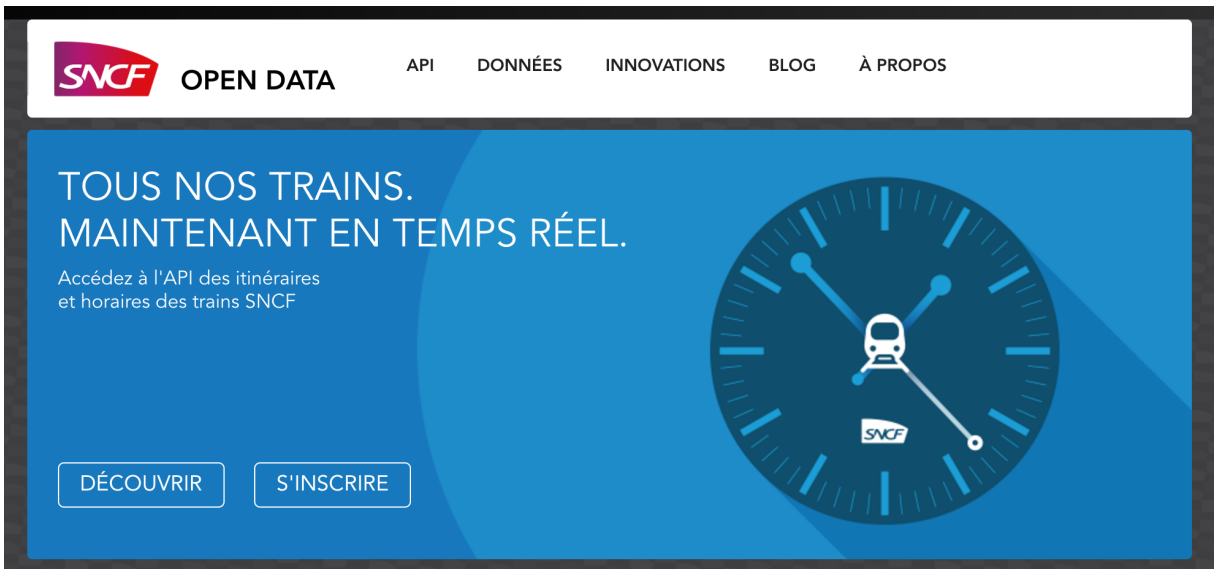


Figure 1.3.: SNCF Open Data Platform.

The real-time API contains the same data displayed at SNCF's stations on panels. The data is requested through a REST API implemented following the Open Data initiative of the SNCF. The output format is in XML for all queries.

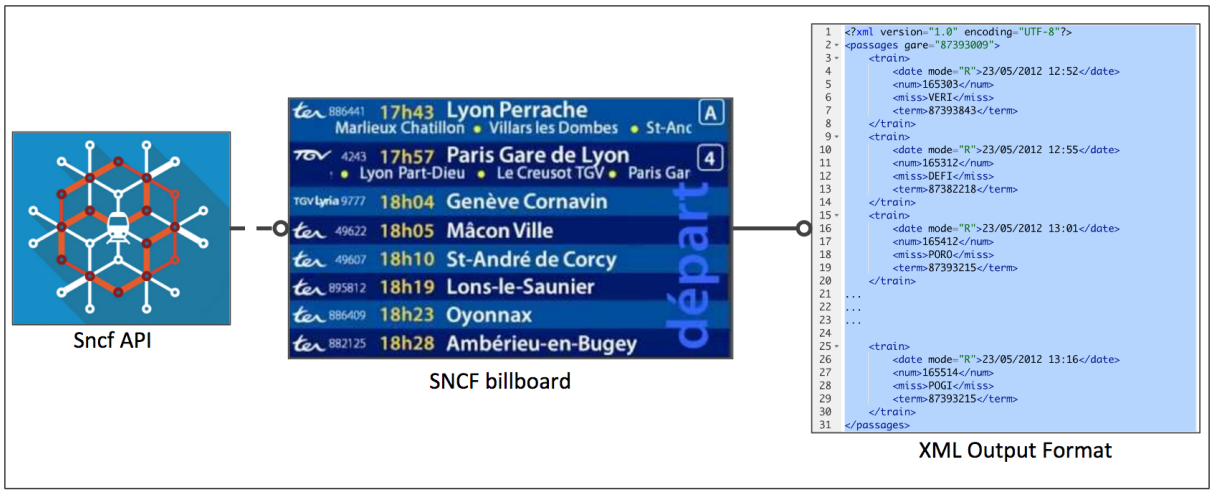


Figure 1.4.: Real Time Data From the Sncf API.

While the GTFS format files are stored within the same website in CSV files, they give the scheduled times of departure of each train, ordered by day of the week and train types.

3. Document Overview

We will present through this report our progress during this first part of the project.

We will first present a detailed state of art on similar research topics. Then we will give an overview on the work that we already have done this far. We present also our work organization, and how we have managed the teamwork during the first period.

As the work is far from complete, we also provide an exhaustive list of the tasks to be carried out for the rest of the project.

Chapter 2

State of the art

State of the art

In this section, we will detail the state of the art, in both the railway industry and in machine-learning and statistics.

1. State of the art in the logistics and transportation industry

Predicting delays in real-time is not a new topic in the logistics and transportation industry. Companies have always tried to estimate delays of trucks carrying goods, freight deliveries, airplanes, buses, subways, etc.

The focal point in this interest is that any delay estimation is a critical component in the economical fare-well of a company. It allows companies to control pro-actively the traffic status by taking actions that prevent delays and delays propagation, better management of the time schedule, transfer plans, rolling-stock and crew dispatch and circulation. It also provides passengers with accurate information whilst standing on the platform, online or in-vehicle.

Most of the companies provide estimates of delays, either in the panels at stations or through the mobile apps that they provide to their customers. Some railway companies like SNCF, Deutsche Bahn and the Central Japan Railway Company provide up-to-date arrival/departure times up to 60 minutes. If a train is delayed, the current forecast delay is displayed at the stations, in the website or through official mobile applications.

The estimate is mainly a human and operator estimate : through inside information (mainly coming for railway operators and the company's personnel) and estimates, these companies are able to provide their customers with average delays.

However, this is not detailed when looking at how do these companies provide accurate delays estimates, which models they use, which features, which metric is used to check the validity of said estimation. This is why we looked at bus traffic for example.

The London bus grid provides real-time estimates of the bus arrival and departures times.

This is mainly possible because those buses are equipped with GPS sensors. By receiving information such as longitude coordinates, latitude coordinates, average speed, with this information, it is easy to give an accurate estimate while knowing the scheduled arrival times. This is not the case for the railway transportation : companies can't pinpoint exactly the location of trains.

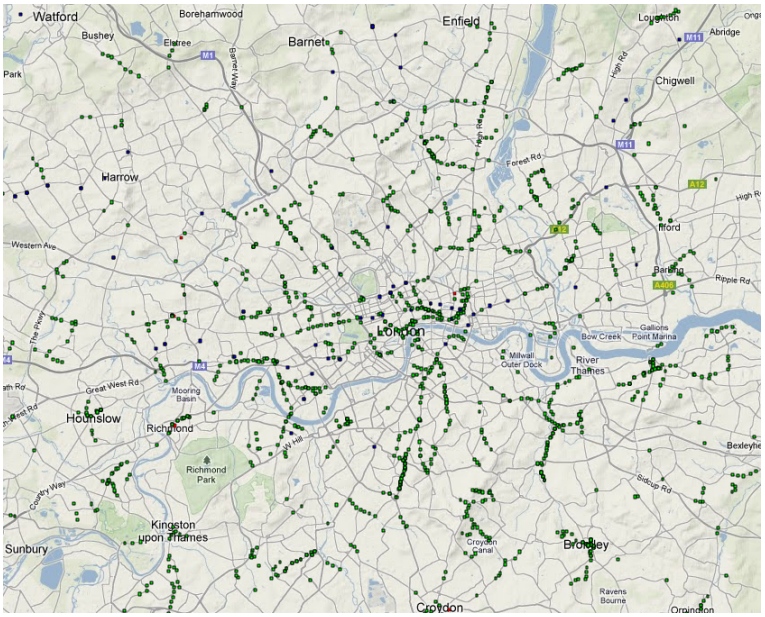


Figure 2.1.: London's bus grid in real time

2. Scientific state of the art

In scientific publications, a lot of articles and studies debated this issue and tried to provide methods to predict delays using real data about railway traffic and machine learning algorithms.

The existing approaches are either deterministic models that require frequent updates of train positions and detailed data, or stochastic models that are largely based on fixed distributions and do not exploit information available in real time.

Since we are using mainly information provided by SNCF's open data API [1], we chose to focus on a data-driven train delay prediction system using machine learning algorithms.

In this part, we will expose an example of the state of the art algorithms that are used to predict train delays.

2.1. Extreme Learning Machine

In literature, two approaches were mainly exposed : a deterministic approach based on real-time information about the exact times and traffic reports, or a stochastic approach based on probabilities using neural networks [3–5]

A train network can be modeled as a graph where each nodes are checkpoints that are connected to each other [5]. A train moving on the network is defined by an itinerary, a

series of checkpoints $\mathcal{C}$, which is a collection of station containing the origin station, a destination station and some stops in between.

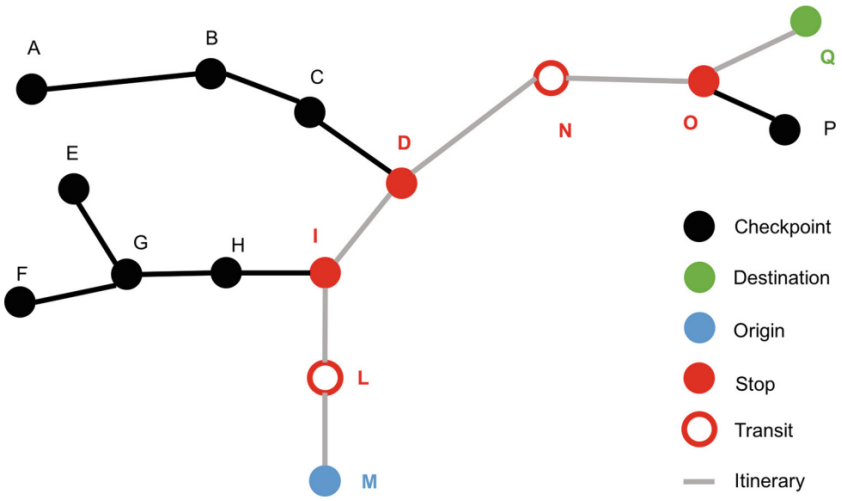


Figure 2.2.: Example of railway modeling

The differences between the scheduled departures times and actual departure times are delay departures. If we take into account the itinerary of a train, for each checkpoint $\mathcal{C}_\rangle$ where $i \in 0, 1, ..., n$, we want to be able to predict the train delays for each subsequent checkpoint C_j with $j \in i+1, ..., n_{destination}$.

In this solution, each delay is treated as a time series forecast problem, where predictive models performing regression analysis or each delay profile, for each train, and for each station or checkpoint C_i.

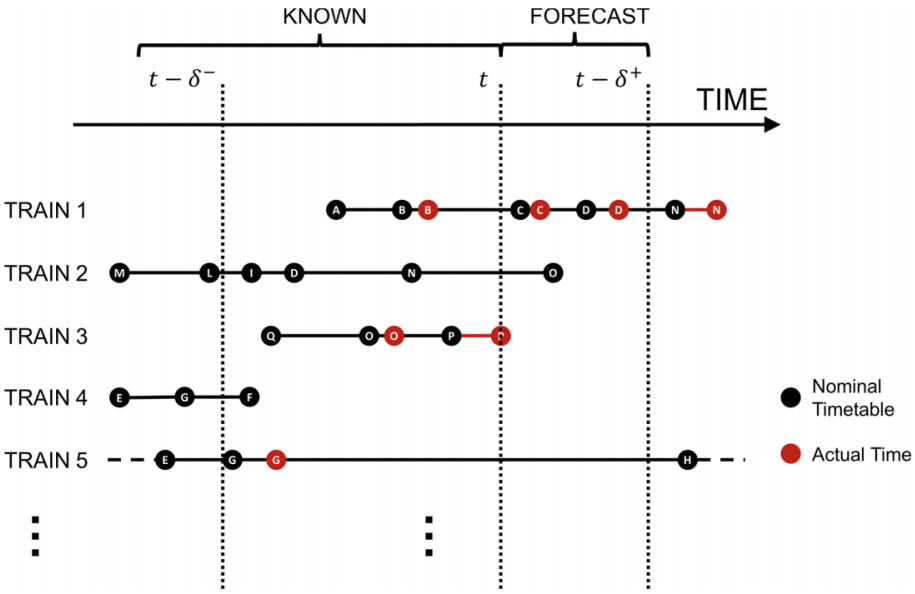


Figure 2.3.: Train delay model

The input of the algorithm would be the historical records about arrivals and departures for each train at each station, as well as extra information such as perturbation reports, disturbance in the grid, etc. If a train is a station A and going to station B passing by various stations, we need to build n delay models at station A, $n-1$ at the next station, and so on.

One of the many algorithms in literature that deals with this issue is Extreme Learning Machines (ELM). It is a learning paradigm for multi-class classification and regression problems [1]. One of the many advantages is that the training speed is extremely fast, and the hidden layer parameters do not require adaptation to the data. Compared to support vector machines where a also solve a convex optimization problem, the cost in computation is not as high and running times are faster than the usual slow times for large datasets.

ELM [2] is mostly a single hidden layer feedforward neural network. Weights of the hidden layers are randomly assigned while weight of the output layer are found via Regularized Least Squares (RLS). A vector of weighted links, $w \in \mathcal{R}^h$, connects the hidden neurons to the output neuron without any bias

$$f(x) = \sum_{i=1}^{h} w_i \phi(W_i, o + \sum_{j=1}^{d} W_i, j x_j)$$

where ϕ is a nonlinear activation function of the hidden neurons and W, the weight vector of the hidden units, is set randomly. Therefore, the weight of the output layer w is fould by solving the following RLS problem

$$w^\star = \underset{w}{\operatorname{argmax}} ||Aw - y||^2 + \lambda ||w||^2$$

where A is a concatenation of the random projection of vectors $x_{i \in 1,\ldots,n}$ based on the

hidden neuron of the ELM :

$$A = [\Phi(x_1), ..., \Phi(x_n)]^T$$

Algorithm 1. SGD for ELM.

Input: $\mathscr{D}_n$, λ, τ, n_{iter}
Output: $\boldsymbol{w}$
Read $\mathscr{D}_n$;
Compute A ;
$\boldsymbol{w} = \mathbf{0}$;
for $t \leftarrow 1$ **to** n_{iter} **do**
  $\boldsymbol{w} = \boldsymbol{w} - \frac{\tau}{\sqrt{t}} \frac{\partial}{\partial \boldsymbol{w}} \left[\|A\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda \|\boldsymbol{w}\|^2 \right]$;
return $(\boldsymbol{w}, b)$;

Figure 2.4.: ELM with Stochastic Gradient Descent

Chapter 3

Completed tasks: data extraction and organization pipeline

1. Overview

This part of the project was data-engineering oriented. The source code is available here.

1.1. Goal

The goal of this first part is to build a robust and reliable manner to extract and transform data, in order to get clean data to analyze.

- at what time did the trains departed from stations
- at what time were the trains scheduled to leave those stations
- subsequently, what were their delays

This is the first part, which will provide usable data to analyze for our analysis and predictions.

It should compute real departure times from api data and serve it schematically it this kind of format:

Jour	Train_id	Station_id	Retard
20170101	TR1	ST1	60
20170101	TR1	ST2	120
20170101	TR1	ST3	120
20170101	TR5	ST1	0
20170101	TR5	ST2	0
20170101	TR5	ST3	0

Figure 3.1.: Result table

Ideally data should be available in realtime. At least we should have access to it the day after.

1.2. Steps overview

1. Set up a scalable distributed database
2. Automate extraction and saving of data from API into a database
3. Find out how to get trains real departure dates in stations out of this data
4. Automate download and saving of schedules from SNCF website in a easily queryable manner
5. Automate schedule and real departures dates matching to compute past delays

1.3. Transversal requirements

It was necessary to implement best practices to make the process reliable:

- **Keep track** of activity to analyze how things go. Used **logging** library
- **Deploy consistently**, 'devops' way: learn to deploy in a flexible and consistent way: use of a **git** repository, use of python **fabric** automation tool to deploy through ssh on AWS servers.
- **Schedule tasks**: linux **crontab** tool to schedule different tasks on different time scales (every 5 minutes, every hour, every day, every week depending on tasks)
- **Parallelized and asynchronous programming** depending on interfaces and request apis: **asyncio** asynchronous single-threaded framework, **motor** library for Mongo asynchronous requests on a single thread. **Multi-threading** for Postgres requests (no easy to implement asynchronous framework on a single thread for postgres).
- **Automated testing:** given the multitude of tasks and modules, it was necessary to create automated testing functions and processes. Even though it makes it a bit longer to code, it saves a lot of time while debugging, and make it more comfortable to deploy without worrying about bugs. We used python builtin **unittest** framework.

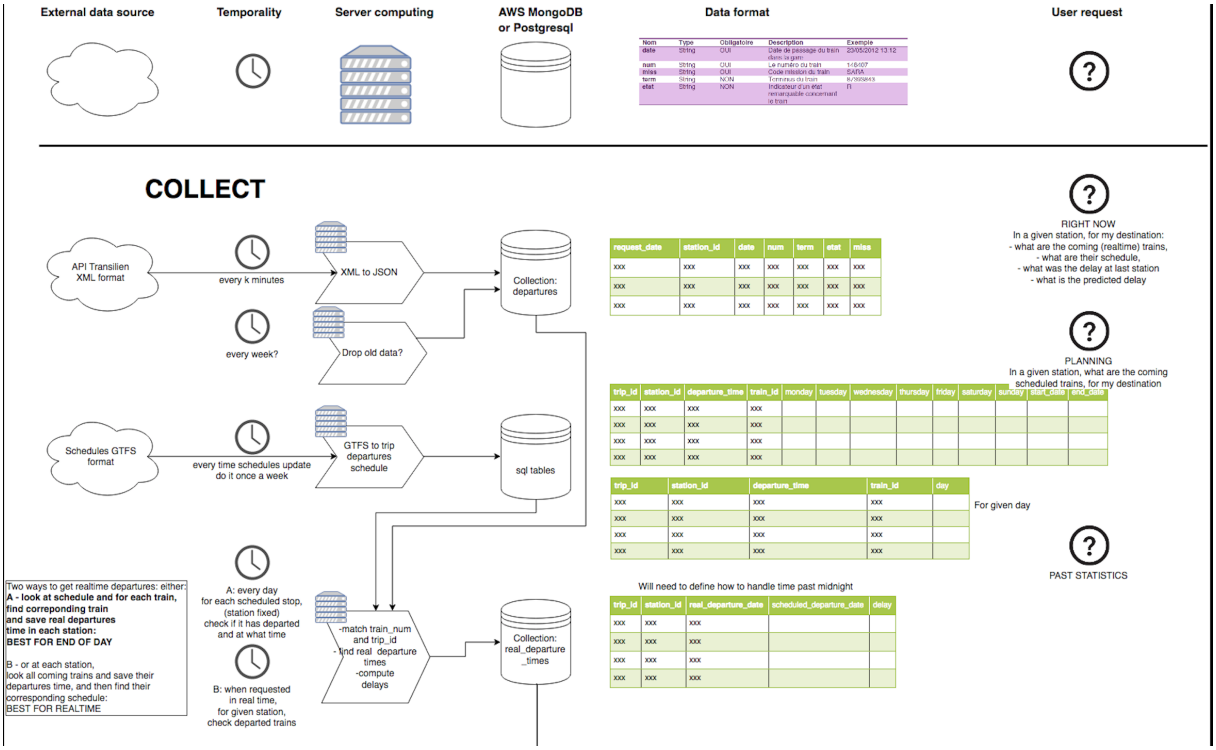


Figure 3.2.: Data pipeline overview

2. Step 1: set up a scalable distributed database

We choose MongoDB for its schema less data-structure. We knew we might change quit often the data-structure.

It is set up on a AWS ec2 instance.

We might change this choice, for it to be more easily managable. Maybe on DynamoDB which provides better performance and more scalability.

We feared it might be a bit expensive.

3. Step 2: extract and save data from API into MongoDB

We got API access keys on 20/12/2016.

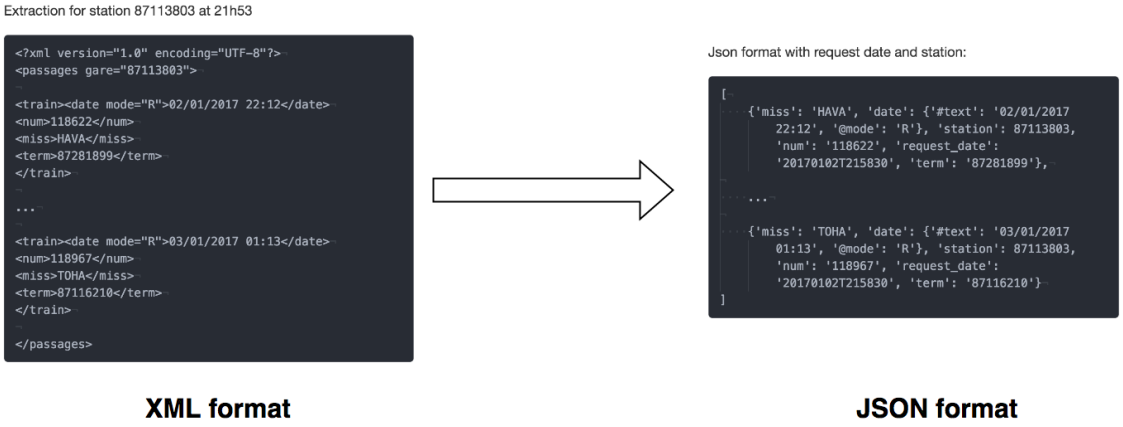
How fast the process could go was a key challenge, it will be discussed in next topic.

3.1. Requirements

- be fast enough to compute 300 api requests per minute
- do not overpass 300 api requests per minute so that we won't be kicked out
- do not multiprocess since it has sometimes some unpredictable crashes on EC2

- keep track of data accuracy (request time and how far is request time from predicted departure time)
- structure data in a clear JSON format containing this information:
  - information about request date
  - all identification attributes of train
  - api predicted arrival time of trains
  - station requested on api giving this information on station boards

Example of items transformation



3.2. Encountered problems

- request time, on AWS EC2 instance server located in Ireland was one hour to early: timestamp was not the same than expected. So we had to implement timezone aware dates. Time zone is now uniform: we only use Paris timezone dates.
- single threaded queries to launch queries in a single batch were quite of a challenge

3.3. Results

This part process 300 hundreds queries to the API at a single time with one single thread for more stability, in about 2 seconds. It takes about 1 more second to parse and format, and 5 seconds to store the about 5500 items in Mongo index-less 'departures' collection in the format described below. So less than 10 sec for an operation that should not overpass 1 minute.

Example of item of 'departures' collection

```
In [14]: dep.find_one()
Out[14]:
{'_id': ObjectId('588f18fe10975a317040a62e'),
 'data_freshness': 298,
 'date': '30/01/2017 14:35',
 'expected_passage_day': '20170130',
```

```
'expected_passage_time': '14:35:00',
'miss': 'TAVA',
'request_day': '20170130',
'request_time': '14:30:02',
'station': '87113803',
'term': '87116210',
'train_num': '118201'}
```

4. Step 3: find out how to get real departure dates out of this information

This part was the most challenging one, and even if it is working properly, we have not found a solution for some edge cases occurring when train departures scheduled before midnight are delayed after midnight.

4.1. Problem overview

All requests from the previous step correspond to what is displayed on stations boards to inform passengers of predicted departures dates of the next 20 trains about to stop in the given station. Querying the API every 5 minutes, we thus have many duplicates.

For example, let's consider a given train, say "train_num":"STOC04", on a given day, "request_day":"20170129", for a given station: "station":"87001479".

What we basically do is checking the board every five minutes. Exactly like if we were in the given station, waiting for instance for a train predicted to arrive in an hour. We would check (and store) the information every five minutes: board would say:

- at 12:00: train should arrive at 13:02
- at 12:05: train should arrive at 13:02
- etc, etc
- at 12:55: train should arrive at 13:03
- at 13:00: train should arrive at 13:04
- – no more information (train passed the station) –

Example: in 'departures' collection: duplicates for a given station, on a given day, for a given train number: 24 duplicates

```
In [20]: duplicates = deps.find({"request_day":"20170129","train_num":"
    STOC04","station":"87001479"})

In [21]: duplicates.count()
Out[21]: 24
```

But in the end, what we want to store in our database is only at what time the train departed from the station. In the previous example we assume that the train left at 13:04

since it is the last prediction before we are sure the train passed the station (because 5 minutes later the train is no more displayed on the board).

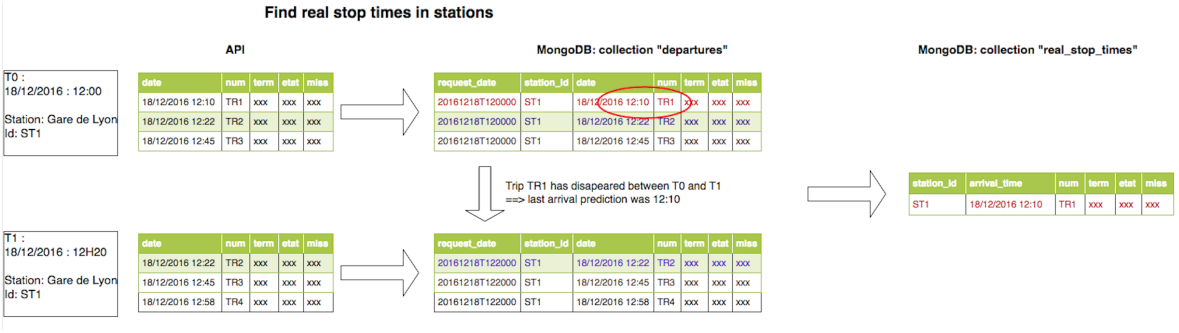


Figure 3.3.: Process explanation to find 'real' departure time

4.2. Potential technical solutions

Make scheduled aggregations: (for instance every hour)
We could have tried to query Mongo with aggregation queries, querying the last added item for a given station, a given **Pros** - require good logging, and no errors to get good reliability: managable - able to do it later

Cons - aggregates are very CPU intensive for MongoDB on millions' items collections: might require new collections every hour? or so?

IMPLEMENTED SOLUTION: Upsert all incomming data in collection with compound index.

Each time a departure object is added to the departures collection it is added in another table real_arrival_times, with a unique compound index on following fields: 'train_num', 'expected_departure_time', and 'station'. Because we should finally get only one train number, for a given station, on a given day (train numbers are unique on a given day).

Moreover it makes sense to 'override' previous stored information and keep only the last one (the freshest one) we got from the API.

So instead of storing items only in 'departures' collection, we also set another collection with unique compound index (and also others indexes for coming queries performance).

Commands to create indexes on 'real_departures' collection

```
real_departures.create_index( [("expected_passage_day", pymongo.ASCENDING),
    ("station", pymongo.ASCENDING), ("train_num",pymongo.ASCENDING)],
   unique=True)

real_departures.create_index("train_num")
```

```
real_departures.create_index("station_id")
real_departures.create_index("scheduled_departure_day")
```

Pros - easy to understand process - quit reliable

Cons - quite intensive: at each element: save it in one collection, update other collection (find+update query => might be long) - if the index is (day, train num, station): how do you do after midnight: you might erase next day data, or be erased later on: this has been mostly taken in charge by setting a unique compound index on (train_num, expected_passage_day, station), and transforming passage date to previous day when hour is between 0 and 3 am, and saving hour +24.

4.3. Make it fast: asyncio, aiottp, motor

The main difficulty was to be able to do it quickly enough. All included to be very fast, to request api, save data, request previous data from Mongo.

Multiple approaches were available, the most scalable and robust one was to make asynchronous requests: - finally: asyncio programming, with special libraries made for asynchronous tasks: aiottp, motor (for pymongo)

What was finally rejected: - multiprocessing / multithreading - split stations to request and set different parallel tasks/workers - python gevent library

4.4. After midnight problem

'Custom day' to take as index to avoid errors
An other challenge was to avoid 'overwriting' wrongly trains from previous days.

Actually, train_ids are unique per day, but are unique based on their 'first station departure day'.

So if a train whose number is 'numA' begining its service on monday is scheduled to leave station 'stationZ' at 00:15 on tuesday, there might be hypothetically a train with same train number 'numA' beginning its service later on tuesday, passing by station 'stationZ' maybe on 23:45. Because train_ids are unique per **first station departure day**.

The best solution is to have the same philosophy as GTFS schedules format, saying that at train whose schedule started on Monday 2017/01/30, will have regular times until next day, and on next day, Tuesday, if there are scheduled stop for the same service that go above midnight, will be saved as monday after 24h: for instance: 2017/01/30 25:23:00 instead of 2017/01/31 01:23:00.

Intractable problems
How to handle delayed trains scheduled on 23:59 and arriving at 00:01?
Maybe for trains arriving between midnight and 1 AM we should directly try to find their trip_id and check if they are from the day before or from the next day, checking the most consistent one (check against scheduled time if both alternatives are possible).
Might cause some performance issues.

4.5. Result

The main bottleneck is the real_departure_time saving step which is supposed to be computed ideally as fast as possible, but will be difficult to compute more than every 4 minutes without selecting bigger Mongo servers, or migrating to another database.

Log sample for the extraction and saving process (more detailed in appendix)

```
2017-01-30 15:55:01,927-- __main__ -- INFO -- Beginning single cycle
   extraction
2017-01-30 15:55:01,934-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Extraction of 300 stations
2017-01-30 15:55:02,171-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Parsing
2017-01-30 15:55:05,292-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Saving of 253 chunks of json data (total of 5549 items) in Mongo
   departures collection
2017-01-30 15:55:09,834-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Upsert of 5549 items of json data in Mongo real_departures_2
   collection
2017-01-30 15:56:31,646-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Time spent: 89 seconds
2017-01-30 15:56:31,647-- api_transilien_manager.mod_01_extract_api --
   WARNING -- Chunk time took more than one minute: 89 seconds
2017-01-30 15:56:31,647-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Extraction of 207 stations
...
...
```

What is described here is that given our API request limit per minute (300/min), we can not query all station at the same time. So the process is:

- divide the stations to query in chunks of maximum 300 items
- for each chunk:
  - query API for chunk stations
  - transform XML into JSON and add fields: station, data_freshness, expected time, expected day
  - save them in 'departures' collection (collection with duplicates), this part is really quick with asynchronous requests
  - saving them then in the 'real_departures' collection (collection which has a unique compound index), this part is much longer since the database has to first find if an element is already present on the given index, and upsert the element.
- if the process took less than 1 minute, wait till end of minute, otherway the transilien API might block us.
- continue until all chunks have been computed

Given the 550 stations requested, this process takes about 3 minutes to finish. But one has to be aware that if one process takes too much time, it won't stop another task to

arrive 5 minutes later, and it might overload the database server (currently a free AWS EC2 nano instance, we consider migrating on Dynamo or paying for a bigger server). So we kept a security distance of 2 minutes.

5. Step 4: extract and save schedules data from sncf website into Postgres

The main parts were: - to understand the GTFS data structure - to find a practical way of storing it in a way it could be fast and simple to query later

5.1. Understand the GTFS data structure

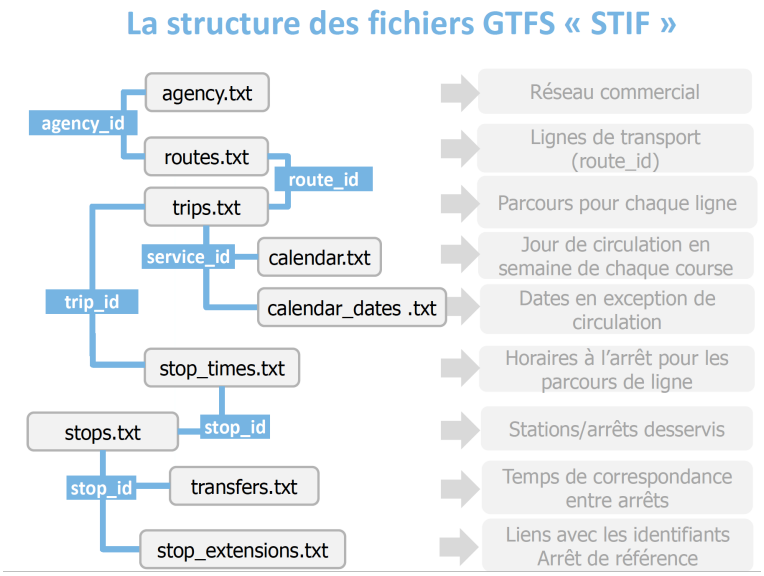


Figure 3.4.: GTFS file structure

5.2. How to store it in a useful way

Goals

- **Find trip_id:** allow to find a trip_id given a train_num and a given day
- **Find scheduled:** allow to find scheduled_departure_time given a trip_id and a station.

Requirements

Needs to be:

- quick to request item per item (OK)
- ideally could get multiple schedules within a single request (NOT YET)

5.3. Solutions

Multiple solutions have been tried:

- save scheduled in modified csv files, import it with numpy/pandas: not adapted to single requests (needs each time, even for a row, to load a multiple MB file)
- save each day, all scheduled trips in a mongo collection: works, but not useful
- **IMPLEMENTED SOLUTION** save useful data in Postgres tables

  Solution for now: 2 relatively 'flat' sql tables:

  - trips_ext (for extended): to match train id and trip id for given date
  - stop_times_ext : to find scheduled stop times for a given trip_id, and a given station.

5.4. Result

A scheduled task successfully automatically update every week these files, following the following steps:

- download a csv files containing urls at following URL: https://ressources.data.sncf.com/explore/dataset/sncf-transilien-gtfs/download/?format=csv&timezone=Europe/Berlin&use_labels_for_header=true (content example in annexes)
- extract url from this text file and download zip files located at given urls: usually two folders: 'gtfs-lines-last' and 'gtfs-routes-last'
- extract zip files headers and unzip files in folders with correct names (headers names)
- load data in pandas dataframes, apply some transformations and merging
- proceed to insert in Postgres database, with two tables:
  - table 'trips_ext': containing trips information with calendar information which allows to know on which days trips are available: this table will help to find trip_id from train_num and day.
  - table 'stop_times_ext': containing information about at what time a given trip_id should leave a given station: this table will help to find scheduled_departure_time from trip_id and station

Currently, the process deletes former tables at each iteration. We could improve it by setting a primary key to avoid

6. Step 5: Match real departure times, and schedule, compute delay (and choose when to do it)

Do not always work for now. This operation is done once a day for now.

6.1. Matching trip_id and train_num

For most trains: train num is composed of 7 digits.

It should be precised, that a trip id or train num is not enough to know exactly which trip we are talking about. We have to specify trip id or train num, WITH the day (a same trip id / train num, corresponds to several trains railing on different days).

Train num to trip id: With these 7 digits, and the scheduled day of service, we can find the trip id.

Trip id to train num: Easy: extract 7 digits.

6.2. Deciding when to compute it

7. Conclusion: overall results for the extraction process, and remaining challenges

7.1. Result: pipeline statistics summary

- There are 507 stations. Not all provide information. At most 387 of them do at peak times.
- Each call to the api to request coming trains for a given station returns at most 30 items.
- We get between 7500 and 8000 items every 5 minutes, at peak times (about 20/stations) when computing request to all stations. It takes about 150 seconds to process, 90 of which is because of the upsert operations in 'real_departures' collection.
- Computing requests every 5 minutes, we save about 650 000 items a day in the 'departures' collection (reminder: this collection contains all data provided by api).
- Computing requests every 5 minutes, we save about 35 705 items a day in the 'real_departures' collection (reminder: this collection contains real departures times, so just one per train per station in a given day).
- So 'departures' collection contains around 20 duplicates of each item.

7.2. Result: deployment strategy

Deployment strategy is fully automatized with 'fabric' framework files.
Within a single command, one can deploy a task manager on a fresh ubuntu server.
Requirements are the following:

- having set up a EC2 instance on AWS based on Ubuntu 16 image
- having set up ssh rsa keys
- save secret passwords and credentials in a json file named 'secret.json' at the root directory of the repository.

7.3. Challenge: How to handle train stops over 2 days: after midnight

This part is very tricky. Both date management and delay computing after midnight were a complicated problem. The current solution seems to be the most robust one but still has drawbacks:

- if a train departure at a station is scheduled at 23:59, and that the train is 2 minutes late expected to arrive at 00:01, how to know that it was scheduled for the previous day?
- if a train scheduled for 23:59, is saved multiple times as the previous day index, and then is late by 5 minutes, it will be saved twice.

7.4. Challenge: measure data accuracy

We have to choose how often we query the API. We should then analyze statistics to know how much a more frequent extraction impacts on our data accuracy.

We implemented a data_freshness field which computes the number of seconds separating the extraction time from the expected departure time.

7.5. Challenge: matching trip_ids and train_nums

There are still some difficulties, among which:

- some train_num are not digits, (different hierarchy)
- we sometime find multiple trip_ids for a given train number, and a given date

+

7.6. Challenge: consistent data-state for real time predictions

How to handle realtime? When should all operation be performed? Batch every hour? Or on realtime? When can we consider data as accurate or reliable? What are the requirements for each step.

Data accuracy
We have to check if 5 minutes interval is not to much.

Data: updatable or final state
Data brought to 'departures' collection is never modified. All elements are kept. Data is thus always in its 'final' state.

Data inserted in 'real_departures' can be updated. It is not necessarily in its 'final' state. We will have to build rules to know when items in this collection are considered as "final".

7.7. Challenge: robust database

MongoDB is known to have some major drawbacks and to be not very efficient, when asked to scale up. It might cause some problems when we will use Spark on it.

We consider migrating to AWS DynamoDB.

Chapter 4

Project Management

1. Introduction

In this section, we will first describe the context in which we initiated this project. Then we will detail the management mode we adopted, the milestones identified, the planning and the organization of the team. We will end with a part dealing with the next steps to be realized as well as the problems encountered.

2. Project analysis

2.1. Milestone identification

Iterative development splits large problems into smaller chunks but without an incremental approach it requires a perfectly shaped idea and solution upfront as well as dead accurate estimation to build the right thing according to plan.
So, we took the choice to realize several deliverables with customer validation at each iteration.

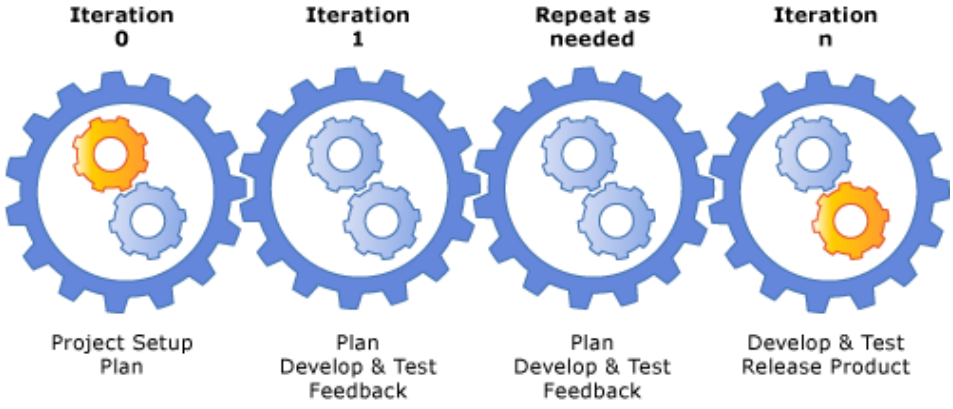


Figure 4.1.: Project Iterations

1. iteration: Data retrieval.
2. iteration: Prepossessing data to retrieve the actual hours of arrival and the scheduled times.
3. iteration: Predict delays with a simulated translation of the schedules.
4. iteration: Proposing a prediction using a more sophisticated model
5. iteration: Set up an API for retrieving delays.

3. Implementation of the project

3.1. Human management

Manage Project Team is the process of tracking team member performance, providing feedback, resolving issues, and managing team changes to optimize project performance.

The human resource management plan provides guidance on how project human resources should be defined, staffed, managed, controlled, and eventually released. It includes, but is not limited to:

- Roles and responsibilities.
- Project organization.
- Staffing management plan.

Above the team members, their skills and roles in the project:

- Leonard Binet: from a business school background, he is a technology enthusiast, he has put in place the first layers that make up the architecture of the project.
- Mohammed Benseddik : Momo a jack of all, he has contributed to the establishment of the MongoDB database, it also works on various technical issues..
- Sami Bargaoui : Sami is the mathematician of the team, former quantitative analyst his skills will be used in the processing part and setting up predictive models..
- Yassine Errochdi : Former consultant and project manager, he deals with aspects related to project management, he also intervenes on technical parts.

The competence of the team members are pretty balanced, and do not hesitate to communicate to identify problems and provide solutions.

3.2. Project time management

Below, the different important project dates:

- 08/12/2016: Project kick off.
- 03/01/2017: Meeting Sncf.
- 04/01/2017: Progress update.
- 17/01/2017: Meeting Sncf.
- 06/02/2017: Presentation.

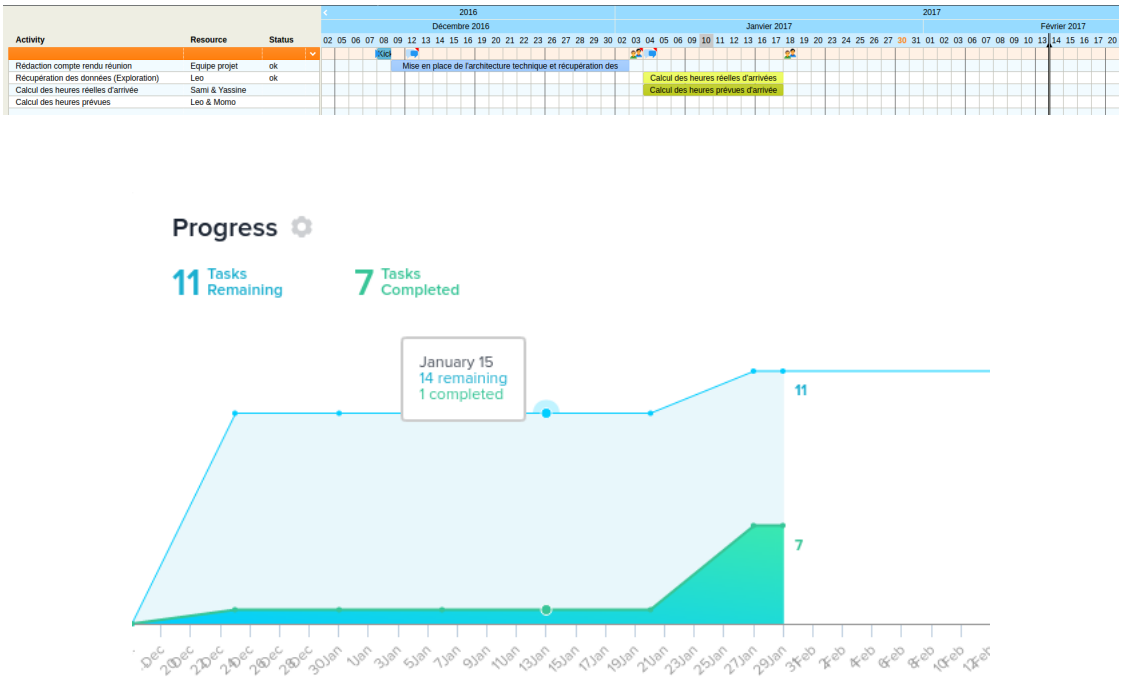


Figure 4.2.: Planning and Progress

3.3. Tools

Access to planning

In order to plan the tasks we use Asana calendar:

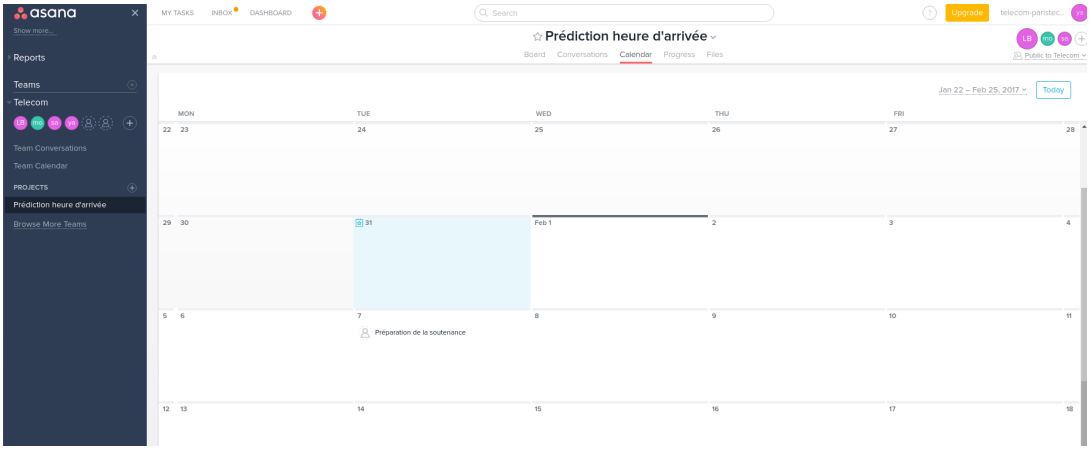


Figure 4.3.: Asana planning

Access to documentation

To store our project documents (technical documentation, meetings reports etc ...) we use Asana Files:

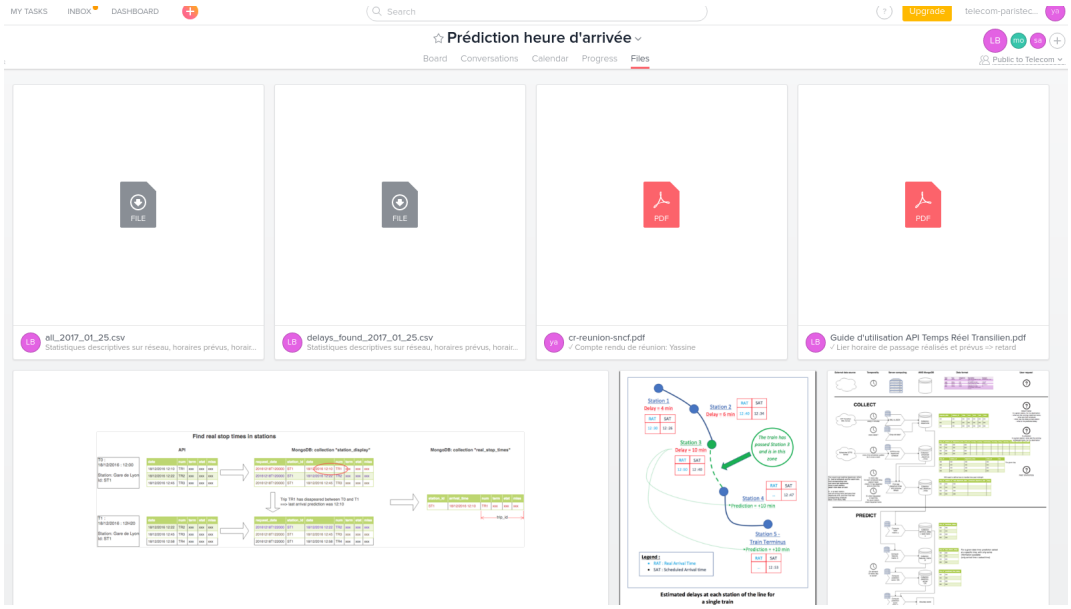


Figure 4.4.: Asana document management

Communication

In order to communicate among the members of the team we use:

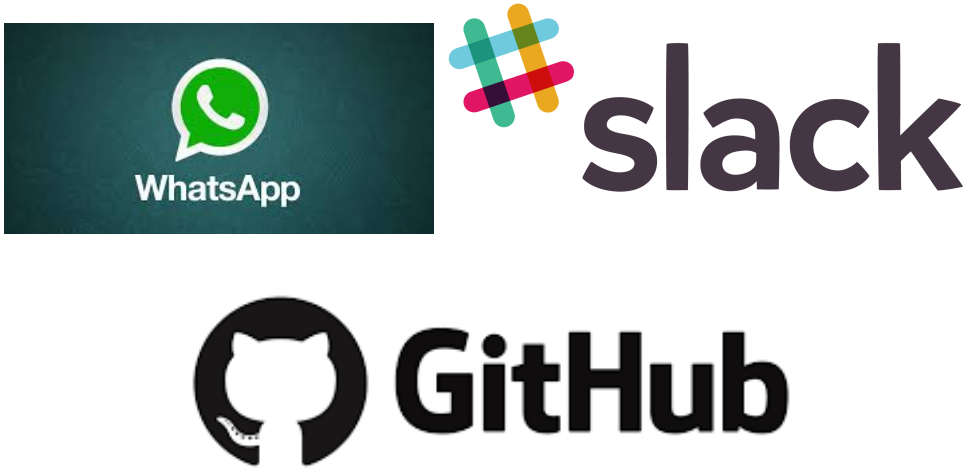


Figure 4.5.: Communication tools

3.4. Project monitoring

Encountered problems

Below is a list of the difficulties encountered:

- Communication problems.
- Unclear roles.
- Lack of involvement of project members (lack of team dynamics).

Proposed solutions

Below is the list of actions to implement:

- Appointment of a project manager (Errochdi Yassine), who will coordinate and monitor the various tasks.
- Increase the frequency of team meetings so that all team members have the same vision of project progress.
- Thoroughly delineate tasks.
- Communicate even in stalemate, to bring them as soon as possible solutions.

Chapter 5

Next Steps

1. Improvement of our System

1.1. Improve trip_id & train_num matching

All steps leading to obtaining 'delays' in 'real_departures' collection work correctly except the step matching api train_nums with GTFS trip_ids.

We can probably find ways to improve the current ratio (60%). Current process works well for 7-digits train numbers, but it doesn't work for 'letters train identification numbers'.

1.2. Migration from MongoDB to DynamoDB

The use of MongoDB as the first solution in order to have a functional and fast result was very satisfactory. MongoDB was very suitable because of its flexibility (schema-less data-structure).

However, scaling MongoDB when we will have to operate complex queries on our data with Spark, for real-time prediction will cause bottlenecks. MongoDB databases are told to be quite difficult to maintain and scale.

Moreover, if we want to speed up our data-extraction ratio, we will have to make it more responsive.

We have therefore decided to migrate to a DynamoDB database hosted on AWS. The Dynamo databases have mostly the same functionalities and query syntax.

The main benefit will be that we won't have to worry about infrastructure or how to scale. This will be especially important when we will try to implement real-time prediction algorithms.

2. Implementation of an API

2.1. Set up an API server

Since our code is written in python, we consider setting up an API with the django rest framework: http://www.django-rest-framework.org/.

2.2. Set up a reliable manner to implement algorithm on data contained in DynamoDB

We can use Spark on DynamoDB. One challenge will be to be able to configure it, and making it work on a production environment.

2.3. Implementation of the time translation algorithm

The implementation of a translation algorithm will be the first step that we must integrate into our ecosystem. This will allow us to have, of course, a basic and simplistic solution, but that will show the power of our API. The predictions will obviously be biased through this solution, but this will allow us at least to have a demo of first version.

2.4. Implementation of more complex algorithms

After having tested some machine learning algorithms, we will try to implement them in a production environment.

3. Reliability of data

3.1. Study the credibility of raw data extracted from the SNCF API

This part constitutes a real challenge within the framework of our project. The problem lies in the "credibility" of the data that is extracted from the SNCF API. Indeed, we do not have access to the actual transit times of the SNCF trains, but only to the display hours of each train. This increases the margin of error of our application, and forces us to implement a real time building system from the API data.

The system we tested gives us realistic results. We have not, however, trespassed if these results are in line with reality. We also note that we have some aberrations in our data after the extraction and processing of train times.

This part is certainly very challenging, and will ask us a real work of reflection and criticism on the data of the SNCF.

4. Exploratory Data Analysis

4.1. Collaborative list of tracks to explore

We will set up a sharing space between the members of the team to ask questions about the data to be studied. We have enormous possibilities since the data of the SNCF are extremely rich and the tracks to be explored very numerous. The aim is to have a part "Data Science" with several tracks to explore, studies to start on the various trains of the SNCF. The system of collaboration between members on this part aims to have a general work and a shared curiosity about the work on the data, even if some members of the team will be more occupied by the other parts of the project.

4.2. Set up synthetic visualizations

The different studies that will be done on the data will be mandatory to synthesize through the different visualizations. The aim is to have also speakable deliverables at the end of this part, and to show in an elegant way the different discoveries and the Benchmarks that will have led.

5. Prediction

5.1. Study of possible Machine Learning algorithms

We must, of course, implement a system for predicting SNCF train times. We have made a state of the art on the different methods used in this case, through Machine Learning algorithms. We will try in this part to experiment these algorithms and to have a solid system of prediction, which will replace the translation algorithm over the long term.

5.2. Setting up a Scoring System

An implemented Machine Learning algorithm can only be credible through a validation or scoring system. We did not have a real system of the SNCF, that's why we have to implement our own scoring function to evaluate our model.An implemented Machine Learning algorithm can only be credible through a validation or scoring system. We did not have a real system of the SNCF, that's why we have to implement our own scoring function to evaluate our model.

6. Summary

We have so much to do within the next weeks. The project is only in its infancy, and we have only installed a first infrastructure that will allow us to really attack the real work of the project. All of the above-mentioned tasks are parallelizable, and we will manage our team to optimize our efforts on the subject.

The diagram below summarizes all the tasks to be done.

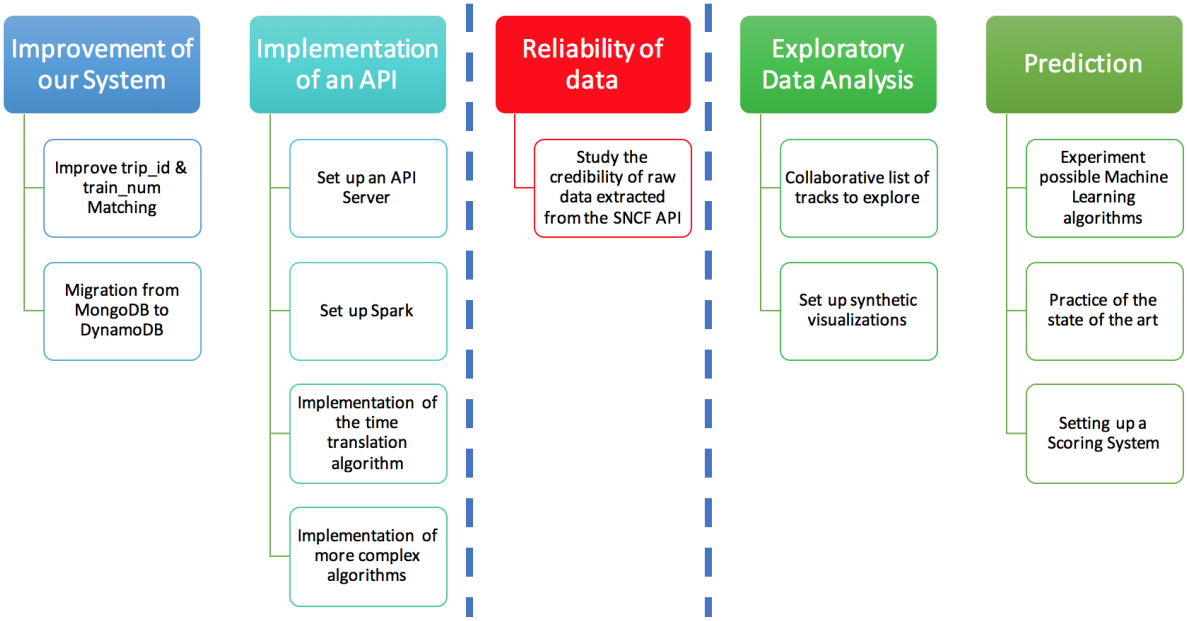


Figure 5.1.: Task List

Appendix A

Data samples through pipeline steps

1. API XML format

```
<?xml version="1.0" encoding="UTF-8"?>
<passages gare="87393009">
<train><date mode="R">23/05/2012 12:52</date>
<num>165303</num>
<miss>VERI</miss>
<term>87393843</term>
</train>
<train><date mode="R">23/05/2012 12:55</date>
<num>165312</num>
<miss>DEFI</miss>
<term>87382218</term>
</train>
<train><date mode="R">23/05/2012 13:01</date>
<num>165412</num>
<miss>PORO</miss>
<term>87393215</term>
</train>
...
...
...
<train><date mode="R">23/05/2012 13:16</date>
<num>165514</num>
<miss>POGI</miss>
<term>87393215</term>
</train>
</passages>
```

2. 'Departures' collection item after extraction and initial saving in Mongo:

```
In [14]: dep.find_one()
```

```
Out[14]:
{'_id': ObjectId('588f18fe10975a317040a62e'),
 'data_freshness': 298,
 'date': '30/01/2017 14:35',
 'expected_passage_day': '20170130',
 'expected_passage_time': '14:35:00',
 'miss': 'TAVA',
 'request_day': '20170130',
 'request_time': '14:30:02',
 'station': '87113803',
 'term': '87116210',
 'train_num': '118201'}
```

3. 'Departures' collection: without index, duplicates example for a given station, on a given day, for a given train number: 24 duplicates

```
In [19]: deps.find({"request_day":"20170129","num":"STOC04","station
    ":"87001479"})
Out[19]: <pymongo.cursor.Cursor at 0x107477550>

In [20]: duplicates = deps.find({"request_day":"20170129","num":"STOC04","
    station":"87001479"})

In [21]: duplicates.count()
Out[21]: 24

In [22]: list(duplicates)
Out[22]:
[{'_id': ObjectId('588d227284174d48cb1881a1'),
  'date': '29/01/2017 04:50',
  'miss': 'STOC',
  'num': 'STOC04',
  'request_day': '20170129',
  'request_time': '00:00:02',
  'station': '87001479',
  'term': '87758722'},
 {'_id': ObjectId('588d25f784174d49086e4867'),
  'date': '29/01/2017 04:50',
  'miss': 'STOC',
  'num': 'STOC04',
  'request_day': '20170129',
  'request_time': '00:15:02',
  'station': '87001479',
  'term': '87758722'},
 .....
 .....
 .....
 .....
 {'_id': ObjectId('588e6cea84174d7a4339a6fb'),
  'date': '30/01/2017 04:50',
  'miss': 'STOC',
```

```
 'num': 'STOC04',
 'request_day': '20170129',
 'request_time': '23:30:01',
 'station': '87001479',
 'term': '87758722'},
{'_id': ObjectId('588e706e84174d7a825cf8ba'),
 'date': '30/01/2017 04:50',
 'miss': 'STOC',
 'num': 'STOC04',
 'request_day': '20170129',
 'request_time': '23:45:02',
 'station': '87001479',
 'term': '87758722'}]
```

4. 'Real_departures' collection: example of item after being updated with scheduled time, trip_id and delay in the 'real_departures' collection (with unique compound index)

```
In [11]: col.find_one({"data_freshness":{"$gte":0}, "delay":{"$gte":0}})
Out[11]:
{'_id': ObjectId('588f2d4f10975a317041cb91'),
 'data_freshness': 204,
 'date': '30/01/2017 16:45',
 'delay': 0,
 'etat': None,
 'expected_passage_day': '20170130',
 'expected_passage_time': '16:45:00',
 'miss': 'GATA',
 'request_day': '20170130',
 'request_time': '16:41:36',
 'scheduled_departure_time': '16:45:00',
 'station': '87111278',
 'term': '87276089',
 'train_num': '145036',
 'trip_id': 'DUASN145036F01003-1_401413'}
```

5. Example items extracted from 'real_departures', for day 30/01/2017

```
In [42]: day = col.find({"expected_passage_day":"20170130"})
In [43]: day = list(day)
In [44]: df = pd.DataFrame(day)
In [46]: df.count()
Out[46]:
_id                  35705
```

```
data_freshness       35705
date                 35705
delay                20465
etat                  1220
expected_passage_day 35705
expected_passage_time 35705
miss                 35705
request_day          35705
request_time         35705
scheduled_departure_time 20465
station              35705
term                 35705
train_num            35705
trip_id              20465
dtype: int64
```

We see that we could not find trip_id for all trains (20565 out of 35705).
We also see that for all trip_ids we found, we always managed to find scheduled_departure_time and delay (20465 trip_ids => 20465 scheduled_departure_times and delays).

6. Schedules from SNCF website: Example of textfile containing urls of schedules zip files

```
file
https://ressources.data.sncf.com/api/datasets/1.0/sncf-transilien-gtfs/
    images/f1a8e58126fbe2f67dd0512ef74a1124
https://ressources.data.sncf.com/api/datasets/1.0/sncf-transilien-gtfs/
    images/023d3733775238ae2e431e3613812bae
```

7. Statistics

7.1. "Departures" collection: number of items saved per day

```
In [129]: run api_transilien_manager/utils_mongo.py
In [130]: deps = mongo_get_collection("departures")
In [131]: deps_on_day = deps.find({"expected_passage_day":"20170130"})
In [132]: deps_on_day.count()
Out[132]: 647954
```

7.2. "Real_departures" collection (with only trains real departures time in stations): number of items saved per day

```
In [136]: run api_transilien_manager/utils_mongo.py
In [137]: rdeps = mongo_get_collection("real_departures")
In [138]: rdeps_on_day = rdeps.find({"expected_passage_day":"20170130"})
In [139]: rdeps_on_day.count()
Out[139]: 35705
```

Appendix B

Log samples

1. Automated deployment process with fabric: extract of deployment procedure logs when code is updated

We did not show here the deployment from scratch since logs are much longer.

```
| => fab deploy:host=ubuntu@ec2-54-154-184-96.eu-west-1
   .compute.amazonaws.com

[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] Executing task '
   deploy'
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] run: mkdir -p ~/
   tasks/api_transilien/virtualenv
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] run: mkdir -p ~/
   tasks/api_transilien/source
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] run: mkdir -p ~/
   tasks/api_transilien/logs
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] run: mkdir -p ~/
   tasks/api_transilien/taskrunner
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] run: cd ~/tasks/
   api_transilien/source && git fetch
[localhost] local: git log -n 1 --format=%H
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] run: cd ~/tasks/
   api_transilien/source && git reset --hard 8
   db66487c76f5b55cc75862774a602cba7a08c26
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] out: HEAD is now
    at 8db6648 bug correction: delay was int, instead of str to be json
   encoded
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] out:

[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] put: secret.json
    -> /home/ubuntu/tasks/api_transilien/source/secret.json
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] run: ~/tasks
....
....
....
```

```
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] out: Requirement
    already satisfied: pygments in ./tasks/api_transilien/virtualenv/lib/
   python3.5/site-packages (from ipython>=0.10.2->ipdb->-r /home/ubuntu/
   tasks/api_transilien/source/requirements.txt (line 7))
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] out: Requirement
    already satisfied: ipython-genutils in ./tasks/api_transilien/
   virtualenv/lib/python3.5/site-packages (from traitlets>=4.2->ipython>=0
   .10.2->ipdb->-r /home/ubuntu/tasks/api_transilien/source/
   requirements.txt (line 7))
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] out: Requirement
    already satisfied: wcwidth in ./tasks/api_transilien/virtualenv/lib/
   python3.5/site-packages (from prompt-toolkit<2.0.0,>=1.0.3->ipython>=0
   .10.2->ipdb->-r /home/ubuntu/tasks/api_transilien/source/
   requirements.txt (line 7))
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] out: Requirement
    already satisfied: ptyprocess>=0.5 in ./tasks/api_transilien/virtualenv
   /lib/python3.5/site-packages (from pexpect; sys_platform != "win32"->
   ipython>=0.10.2->ipdb->-r /home/ubuntu/tasks/api_transilien/source/
   requirements.txt (line 7))
[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] out:

[ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com] sudo: service
   postgresql start

Done.
Disconnecting from ubuntu@ec2-54-154-184-96.eu-west-1.compute.amazonaws.com
   ... done.
```

2. Logs of api extraction and mongo saving process

```
2017-01-30 15:55:01,927-- __main__ -- INFO -- Beginning single cycle
   extraction
2017-01-30 15:55:01,934-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Extraction of 300 stations
2017-01-30 15:55:02,171-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Parsing
2017-01-30 15:55:05,292-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Saving of 253 chunks of json data (total of 5549 items) in Mongo
   departures collection
2017-01-30 15:55:09,834-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Upsert of 5549 items of json data in Mongo real_departures_2
   collection
2017-01-30 15:56:31,646-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Time spent: 89 seconds
2017-01-30 15:56:31,647-- api_transilien_manager.mod_01_extract_api --
   WARNING -- Chunk time took more than one minute: 89 seconds
2017-01-30 15:56:31,647-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Extraction of 207 stations
2017-01-30 15:56:31,967-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Parsing
2017-01-30 15:56:33,675-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Saving of 143 chunks of json data (total of 3088 items) in Mongo
   departures collection
```

```
2017-01-30 15:56:36,409-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Upsert of 3088 items of json data in Mongo real_departures_2
   collection
2017-01-30 15:57:25,168-- api_transilien_manager.mod_01_extract_api -- INFO
    -- Time spent: 53 seconds
....
....
....
```

3. 'Schedules' files extraction and saving in Postgres:

```
2017-01-25 19:23:54,008-- __main__ -- INFO -- Task: weekly update of gtfs
   files
2017-01-25 19:23:54,008-- __main__ -- INFO -- Download files.
2017-01-25 19:23:54,009-- src.mod_01_extract_schedule -- INFO -- Download
   of csv containing links of zip files, at url https://
   ressources.data.sncf.com/explore/dataset/sncf-transilien-gtfs/download/?
   format=csv&timezone=Europe/Berlin&use_labels_for_header=true
2017-01-25 19:23:56,175-- src.mod_01_extract_schedule -- INFO -- Download
   of https://ressources.data.sncf.com/api/datasets/1.0/sncf-transilien-
   gtfs/images/f1a8e58126fbe2f67dd0512ef74a1124
2017-01-25 19:23:56,595-- src.mod_01_extract_schedule -- INFO -- File name
   is gtfs-lines-last.zip
2017-01-25 19:23:56,595-- src.mod_01_extract_schedule -- INFO -- The 'gtfs-
   lines-last' folder has been found.
2017-01-25 19:23:56,794-- src.mod_01_extract_schedule -- INFO -- Download
   of https://ressources.data.sncf.com/api/datasets/1.0/sncf-transilien-
   gtfs/images/023d3733775238ae2e431e3613812bae
2017-01-25 19:23:57,107-- src.mod_01_extract_schedule -- INFO -- File name
   is gtfs-routes-last.zip
2017-01-25 19:23:57,253-- root -- INFO -- Create flat csv.
2017-01-25 19:24:00,607-- root -- INFO -- Inserting data in postgres
```

4. Logs of step 3: adding trip_id, schedule and delay to items in 'real_departures' collection:

```
Last login: Wed Feb 1 21:12:57 2017 from 88.190.162.211
ubuntu@ip-172-31-47-26:~$ sudo /home/ubuntu/tasks/api_transilien/virtualenv/
   bin/python3 /home/ubuntu/tasks/api_transilien/source/
   api_transilien_manager/task_03_d_match.py
2017-02-01 21:14:19,615-- __main__ -- INFO -- Task: daily update: adds
   trip_id, scheduled_departure_time, delay
2017-02-01 21:14:19,626-- __main__ -- INFO -- Paris yesterday date is
   20170131
2017-02-01 21:14:19,633-- __main__ -- INFO -- Processing station 87113803,
   number 0 out of 507
```

4. Logs of step 3: adding trip_id, schedule and delay to items in 'real_departures' collection:

```
2017-02-01 21:14:20,407-- api_transilien_manager.mod_03_match_collections --
    INFO -- Found 83 elements in real_departures collection for station
   87113803.
2017-02-01 21:14:21,211-- api_transilien_manager.mod_03_match_collections --
    INFO -- Found trip_id for 82 elements.
2017-02-01 21:14:25,322-- api_transilien_manager.mod_03_match_collections --
    INFO -- Real departures gathering finished. Beginning update.
2017-02-01 21:14:25,322-- api_transilien_manager.mod_03_match_collections --
    INFO -- Gathered 82 elements to update.
2017-02-01 21:14:25,322-- api_transilien_manager.mod_03_match_collections --
    INFO -- Processing chunk number 0 (chunks of max 1000) containing 82
   elements to update.
2017-02-01 21:14:27,270-- __main__ -- INFO -- Processing station 87116012,
   number 1 out of 507
2017-02-01 21:14:28,048-- api_transilien_manager.mod_03_match_collections --
    INFO -- Found 85 elements in real_departures collection for station
   87116012.
2017-02-01 21:14:28,851-- api_transilien_manager.mod_03_match_collections --
    INFO -- Found trip_id for 83 elements.
2017-02-01 21:14:32,761-- api_transilien_manager.mod_03_match_collections --
    INFO -- Real departures gathering finished. Beginning update.
2017-02-01 21:14:32,762-- api_transilien_manager.mod_03_match_collections --
    INFO -- Gathered 83 elements to update.
2017-02-01 21:14:32,762-- api_transilien_manager.mod_03_match_collections --
    INFO -- Processing chunk number 0 (chunks of max 1000) containing 83
   elements to update.
...
...
...
```

Bibliography

[1] Advances in big data (2016). proceedings of the 2nd inns conference on big data.

[2] Delay prediction system for large-scale : Railway networks based on big data analytics (2015). Luca Oneto, Emanuele Fumeo, Giorgio Clerico, Renzo Canepa, Federico Papa, Carlo Dambra, Nadia Mazzino, and Davide Anguita.

[3] https://data.sncf.com/.

[4] Prediction of delays in public transportation using neural networks (2016). Jan Peters, Bastian Emig, Marten Jung, Stefan Schmidt.

[5] Railway passenger train delay prediction via neural network model (2012). Masoud Yaghini, Mohammad M. Khoshraftar, Masoud Seyedabadi.

[6] Stochastic prediction of train delays in real-time using bayesian networks (2015). P.Kecman, F.Corman, A.Peterson, M.Joborn.