

Skynet

Leonardo Bacciottini

Leonardo Cecchelli

Zaccaria Essaid

Filippo Guggino

June 2, 2020

Contents

1	Introduction	1
1.1	Abstract	1
1.2	Dataset	1
2	Requirements	2
2.1	Functional Requirements	2
2.1.1	Airport	2
2.1.2	Route	2
2.1.3	Airline	3
2.1.4	Administrator	3
2.2	Non-Functional requirements	3
3	UML Use Cases Diagram	5
4	UML Analysis Class Diagram	6
5	Architecture	7
5.1	Write Oriented Database	7
5.2	Read Oriented Database	8
5.3	Modules Definition	9
5.4	Client	9
5.5	Server	10
5.6	MongoDB	10
5.7	Neo4j	12
5.7.1	Entities	12
5.7.2	Relationships	12
5.8	Technologies	13
6	Admin-Protocol Definition	14
6.1	Authentication	14
6.1.1	Client	14
6.1.2	Server	14
6.2	Request message	14
6.3	Acknowledgement	15
7	Implementation	15
7.1	Server	15
7.1.1	Database interaction	15
7.1.2	Updater	16
7.1.3	Admin Protocol Server	17
7.1.4	Scraper	18
7.2	Client	21
7.2.1	Admin Protocol Client	21
7.2.2	Cache	21
7.2.3	Services	22
8	User guide	24
8.1	Initial screen	24
8.2	Overall statistics screen	25
8.3	Airport statistics screen	26
8.4	Airline statistics screen	26
8.5	Route statistics screen	27
8.6	Administrator log in screen	28
8.7	Reserved area screen	29

1 Introduction

1.1 Abstract

Every day about 20000 flights sail through the skies of the United States. Online a huge amount of data is available regarding each one of these flights, including information on delays, cancellations and their causes. From this data it's possible to compute a large number of different statistics on mean delays, probability of cancellation, most probable delay cause etc. based on a single airport, route or airline. This would be useful for all people willing to fly across the U.S. in order to choose the best airline or route or at least to know what performances they can expect from their flight.



Figure 1: Example of daily air traffic over the U.S.

The application's aim is to support the collection and the processing of this data, producing results that are very easy to understand to everyone. Users interactively read the information they want about any airport, airline or route. Queries always aggregate data of thousands of flights and may be very heavy on the server, moreover their latency must be very low. To achieve a good tradeoff between performances, reliability and availability it's been necessary to use different database technologies and software modules that are widely discussed in the rest of this paper.

1.2 Dataset

The data about all these flights is directly provided by the U.S. Department of Transportation at this URL: <https://transtats.bts.gov/DataIndex.asp>. The scraper embedded into the server is simple as it only has to periodically check for new data and compile the correct form for the data. What's noticeable is that this dataset has the velocity property, since information become less significative as it's older. This is taken into account while performing queries (see how in Section 7). Data is also partly variable, because it doesn't come from different sources, but some fields aren't available for all flight records due to unpredictable missing information, so that the system has to handle data with variable structure.

2 Requirements

Let's start from the actor definition: the standard actor is simply called user and may represent any person willing to use the application services. Users can be passengers willing to have an idea about the probability of having their flight delayed or canceled, or also airline managers checking if their company offers the best quality of service to the customers. A particular kind of user is the system administrator. This actor has some additional use cases that are also described in this paragraph.

2.1 Functional Requirements

When a user accesses the service, a screen shows some overall statistics and invites him/her to choose the subject of their queries, that may be:

- An airport
- A route
- An airline

The overall statistics shown are rankings of airports and airlines based on their QoS performances. The user may click on any of the airports or airlines shown to switch to one of the following menus. At the end of this screen, a menu offers the possibility to lookup for a specific airline or airport (To select a route the user can choose an airport and then select a destination airport). Let's see what services each of these choices does offer:

2.1.1 Airport

At this point the user is asked to specify the name of the airport through a text input or by clicking on a few suggestions. Once the specific airport is chosen, the application must provide the following services.

1. Browse the most served routes from this airport. The user may click on a route and switch to the Route menu.
2. Browse the most served airlines in this airport (maybe in a cake diagram). The user may click on an airline and switch to the Airline menu.
3. Get an importance percentage, i.e. an indicator about how many flights involve this airport over the total number of flights in the U.S.
4. Get the probability of having a delay higher than 15 minutes for departing flights.
5. Get the most likely cause of delay for departing flights.
6. Get the probability of having a departing flight canceled.
7. Get the most likely cause of departing flights cancellations.
8. Get the number of airports that can be reached in at most two hops from this airport. [Section 7.2.3]
9. Select a destination airport and switch to the Route menu.

2.1.2 Route

At this point the user is asked to specify the source and destination airport through a text input or by clicking on a few suggestions. Once the specific route is chosen, the application must provide the following services.

1. Know if this route isn't served. In this case the system shows a list of alternatives with same destination and origin airport in the same U.S. state. [Section 7.2.3]

2. Browse an airline ranking (in terms of delay performances) for this route. The user may click on an airline and switch to the Airline menu.
3. Get the probability of having a delay higher than 15 minutes.
4. Get the most likely cause of delay.
5. Get the mean delay on this route.
6. Get the probability of cancellation.
7. Get the most likely cause of cancellation.
8. Click on any of the source and destination airport and switch to the Airport menu.

2.1.3 Airline

At this point the user is asked to specify the name of the airline through a text input or by clicking on a few suggestions. Once the specific airline is chosen, the application must provide the following services.

1. Browse the most served airports (maybe with a cake diagram). The user may click on an airport and switch to the Airport menu.
2. Get a QoS index, based on both delays and cancellations of the flights.
3. Get the probability of having a delay greater than 15 minutes.
4. Get the mean flight delay.
5. Get the probability of having the flight canceled.
6. Get the number of times this route is the best carrier for a route vs the total number of routes served by this airline. [Section 7.2.3]

2.1.4 Administrator

An administrator has all these previous use cases and can also manage the status of the databases. The meaning of the following services will be clearer after the Architecture paragraph. The admin can access his own menu through a dedicated button in the initial screen. This leads to an authentication interaction with username and password that, in case of success, grants access to these features:

1. Change administrator credentials (only one administrator account is allowed in the system).
2. Force the update of the whole statistics set.
3. Set limits on storage space. The server will delete old records if needed.
4. Set starting year. The server will delete all records previous to this year.
5. Force the scraping for new data.

2.2 Non-Functional requirements

Being interactive, the application must be projected to be very responsive to users queries. This means that the analytics that are requested each time must be ready to show. It's not particularly important that the database is updated real time with live flights. What's basic is the scraping to be performed periodically so that there's always recent data into the database. Not even consistency is so important: statistics shown to users can be the ones computed for not updated data during the time necessary to process the new information. Reliability is not a big issue either: losing data is to be avoided but it's not a catastrophe since it's still available to be downloaded from the U.S. Department of Transportation. Of course replication must be used as it's always faster and reliable to recover from a replica already set on the server. A mandatory

characteristic is availability: the server should always be able to accept users requests, and this means that a distributed server is to be used.

3 UML Use Cases Diagram

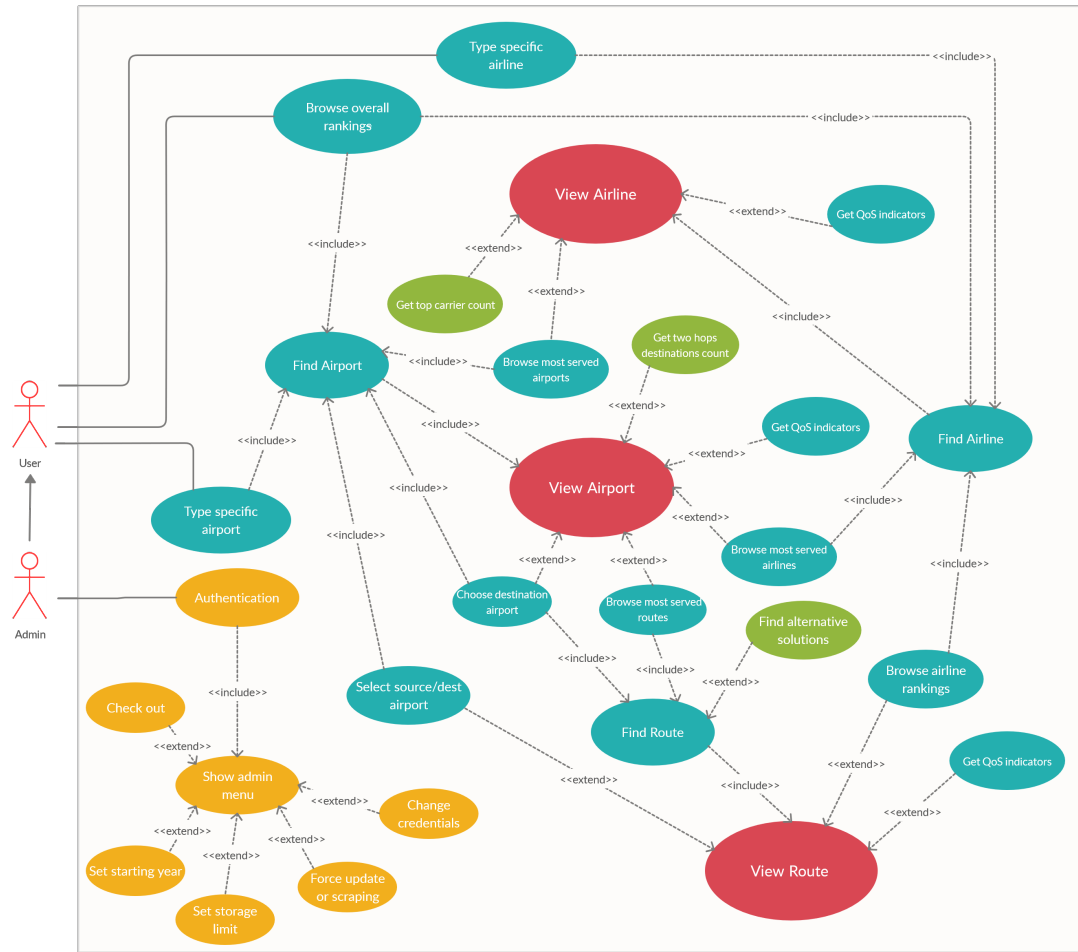


Figure 2: Use Cases diagram for the application

This diagram formally plots functional requirements explained in the previous section. For the sake of readability all probabilities and numeric indicators of performance for airports, airlines and routes are summed up in a single extension called *Get QoS indicators*.

1. Yellow use cases represent administrator interactions.
2. Red use cases are simply for the sake of readability and highlight the three main use cases of the application.
3. Green use cases are standard use cases, but they are implemented through graph oriented queries on the Neo4j database [Section 5.7]
4. Blue is the default color for remaining use cases.

4 UML Analysis Class Diagram

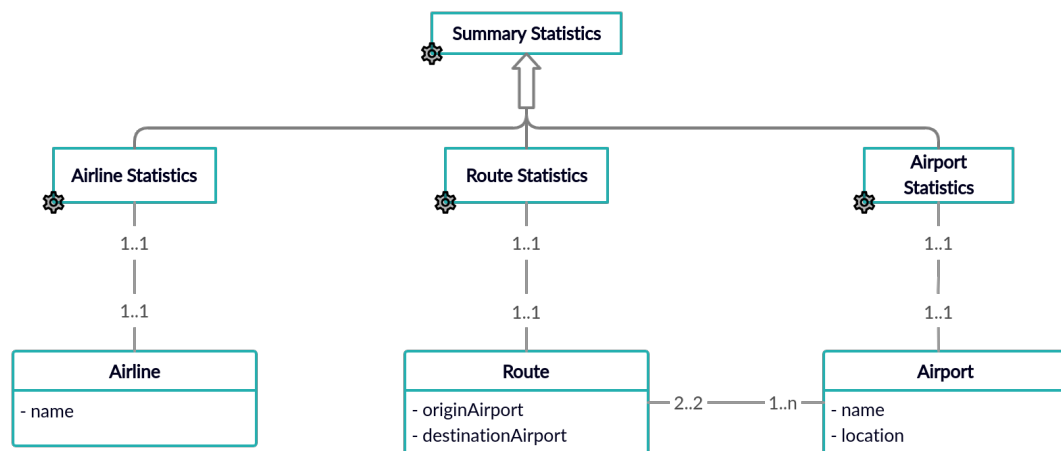


Figure 3: Analysis Class diagram for the application

Here is shown a basic skeleton of the classes, and their relationship, used by the Client. Particular attention was paid in order to highlight the "Inter Usage" between classes previously shown in the Use Cases Diagram (Section 8.1). Mind that this type of diagram is not bonded by the technology which will actually be implemented, as it only strives to throw a basic structure for the future application.

5 Architecture

It's time to choose the structure and technologies to make the database server as efficient as possible.

Talking about the server of the application, it must be able to collect data about flights (obtained by a web scraper), analyze them and make the results available to an interactive client, which could request statistics about an airline, a route or an airport. It would be computationally unacceptable to evaluate these statistics in real time every time they are requested, so a reasonable solution is to use two databases: a write-oriented one, to collect and store data for each flight, and a read-oriented one, in charge of storing aggregated statistics and send them to the client.

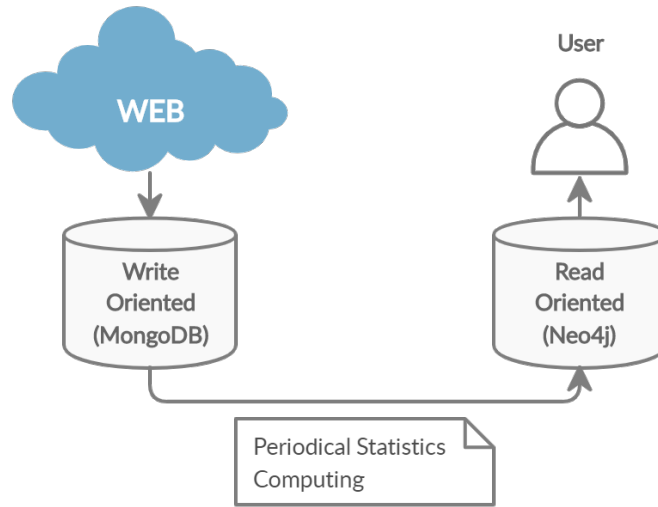


Figure 4: General architecture of the application.

Of course in order to let the two Databases communicate between each other, some kind of middle-ware has to be implemented. Nonetheless a server application (back end) has to keep information located in the read oriented database updated. No particular attention to data confidentiality and integrity will be put as this is not the scope of this project. This means that the task of communication between the two databases will be carried out by a dedicated server application, connected with both technologies, resident on an always-on machine. It periodically checks the update status of the read oriented database and if it's too old, it proceeds with computing new statistics, for a full disclosure of the applications involved see Client [Section 5.4] and Server [Section 5.5].

Since it's very unlikely that a new airport will be built in the near future we can make the hypothesis that the number of nodes inside the graph database will grow very slowly and by that there's no need to consider any problem caused by memory limitations whatsoever.

Before deepening the knowledge of the architecture, let's see what are the choices made for both the database technologies, with proper motivations.

5.1 Write Oriented Database

This database has the following requirements (deduced from the ones in Section 2:

1. It must store an amount of data that can grow very much over time (dozens of Gigabytes per year).

2. Write operations may be frequent, especially if scraped website is often updated with new flights information.
3. Read operations are not frequent, because users don't have access to this store. Later will be explained exactly which software module interacts with this store.
4. Flight data format is variable, since many flights lack of some fields of information depending on what has been recorded by airport and airline employees.
5. The database should be able to perform aggregated analytic queries on the stored data. Latency is not a problem because queries are requested as a batch process.
6. The database should also securely keep some information about administrator credentials (that may change over time).
7. Concluding, the database should have Consistency and Partition protection over CAP theorem. Availability is not a critical issue since this store isn't directly accessed by users.

From all these considerations it's quite straight forward that a document database is suitable for this module, as long as indices defined are a few and limited to enhancing heaviest aggregating queries. *MongoDB* is then a wise choice to be used here.

5.2 Read Oriented Database

This database has the following requirements (deduced from the ones in Section 2):

1. The database stores data organized into airports, airlines and routes between two airports.
2. It must contains an amount of data that can be stored on a single machine and doesn't grow in size over time.
3. Users continuously perform simple and interactive read operations on the database. Latency must be as low as possible.
4. The database is periodically updated with batch operations with aggregated data from the other store.
5. Availability is a basic feature of the database, as it may be concurrently accessed by many users. Then Consistency and Availability over CAP theorem are a good choice (no Partition protection since data can be stored on a single device).

A Graph Database is the best choice here, because the structure of data may easily be seen as a graph (routes are edges between airports), and this kind of database meets all these listed requirements, especially because user queries are very simple in a graph domain. Then, *Neo4j* is the technology for this store. A basic representation can be seen in Figure 5.

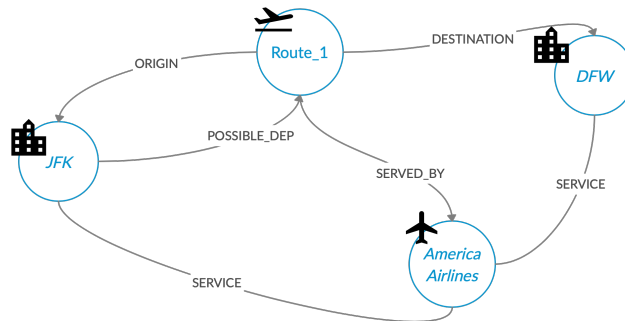


Figure 5: Basic representation of the graph database with nodes and edges.

5.3 Modules Definition

In Figure 6 we may see a more specific diagram of the system architecture, that is now dependent on the design choices. There are two different processes, running on different machines, that from now on will be called Client and Server. They will be now described by listing their tasks.

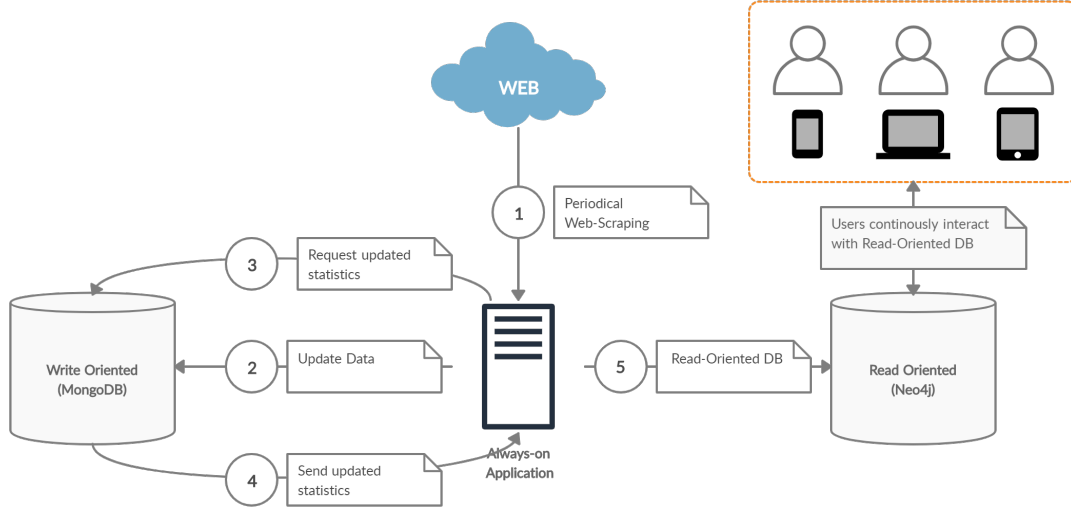


Figure 6: General architecture of the application, considering the interactions between different technologies.

5.4 Client

The client is basically a GUI interface between the final user and the architecture. By that it needs to be as simple as possible:

1. No authentication nor registration is required in order to use the service.
2. For the sake of readability very few text will be inserted, privileging the use of charts and summary statistics.
3. The user will be given the choice to receive very detailed statistics about airline, airport and route. A simple form will be introduced in order to let the user navigate to the information he wants.

Note that the same interface is used by an administrator, who will have to sign in by using credentials in order to manage certain tasks, such as:

1. Manual update of the Read-Oriented database.
2. Manual scraping of new data and therefore update of the Write-Oriented database.
3. Deletion of some data in both Read-Oriented and Write-Oriented databases.

Obviously before introducing the administrator there would be no need for the server to be always on, as it wouldn't receive any requests. It may very well turn on on a specific predetermined time, scrape new data, update the two databases and then repeat. Instead an administrator may start an update request at any time, and by that the need for the server to be always on. In conclusion, there may be multiple instances of the Client simultaneously running at the same time on different machines, and each one of them will establish a connection with the *Neo4j* database to retrieve the necessary data when they are requested. This system architecture will make all these concurrent queries as fast as possible.

5.5 Server

The server is an always-on, single instance application, totally autonomous, whose primary task is the management of databases update and configuration. The machine on which this process is executed may be the same of any of the server in which the two databases reside or an independent one. There are no limitations but being connected to both databases. Only one client at a time can be logged in as an administrator, and one of the server tasks will be to guarantee this feature and handling administrator requests. The main goals of the server are the following two:

1. Periodically run a scraping algorithm to look for new flight data to add to the collections.
2. Keep track of the status of the read oriented database. Inconsistency between the two databases is not an issue, as it's not a requirement that statistics must be updated in real time with new data arrival. So, periodically, the server will proceed with querying *MongoDB* for updated analytics and reload a new updated image on *Neo4j*. The reasonable size of the read database allows global updates without replacing only old data, as the overhead of finding only not updated information would be greater than replacing the whole content.

In conclusion, it's necessary to say that the IP address of the server must be known to all clients, because if one of them logs in as an administrator, it will have the need to send requests to the server through a TCP socket following a very simple application layer protocol defined later in this paper.

5.6 MongoDB

Here we will describe the structure of the MongoDB database. The collection that will track all the informations of every flight will be called *Flights* and will include all the following fields:

- Quarter: quarter of the year (1-4).
- FlightDate: (yyyymmdd)
- Reporting_Airline: unique ID of a specific airline.
- CRSDepTime: scheduled departure time, in minutes.
- DepTime: actual departure time, in minutes.
- DepDelay: difference in minutes between scheduled and actual departure time. Early departures show negative numbers.
- DepDel15: departure Delay Indicator, 15 Minutes or More (1=Yes).
- CRSArrTime: scheduled arrival time, in minutes.
- ArrTime: actual arrival time, in minutes.
- ArrDelay: difference in minutes between scheduled and actual arrival time. Early arrivals show negative numbers.
- ArrDel15: arrival Delay Indicator, 15 Minutes or More (1=Yes)
- Cancelled: cancelled Flight Indicator (1=Yes)
- CancellationCode: specifies The Reason For Cancellation.
- CRSElapsedTime: scheduled elapsed time of flight, in minutes.
- ActualElapsedTime: actual elapsed Time of Flight, in Minutes.
- Distance: total distance travelled.
- CarrierDelay: how much of the total delay is due to the carrier.

- WeatherDelay: how much of the total delay is due to the weather.
- NASDelay: how much of the total delay is due to the National Air System.
- SecurityDelay: how much of the total delay is due to Security issue.
- LateAircraftDelay: how much of the total delay is due to the delay of the previous flight with same plane.

It remains to specify Origin and Destination airport of every flight, one solution could be to introduce a new collection called *Airport* that would indeed allow a small use of storage but also a reduction on the performance of aggregated queries. Since the last is one of the main objectives (if not the one) and from the moment that this is a distributed system, where there's no real limitation in terms of storage, the adopted solution is to implement two nested documents inside *Flights*, one for Origin Airport and the other for the Destination Airport:

- Origin: name of the origin airport.
- OriginAirportID: unique ID of the origin airport.
- OriginCityName: city of the origin airport.
- OriginStateName: state of the origin airport.

And Destination Airport:

- Destination: name of the destination airport.
- DestAirportID: unique ID of destination airport.
- DestCityName: city of the destination airport.
- DestStateName: state of the destination airport.

An example of a document, in JSON format, taken from *Flights* would be:

```
{
  FlightID : 1,
  FlightDate : "20190123", //23th january of 2019
  CRSDepTime : "0600", //6AM
  .
  .
  .
  OriginAirport : {
    Origin: "New York International Airport",
    OriginAirportID : "10067",
    OriginCityName : "New York"
  }
  DestinationAirport : {
    Destination : "Dallas/Fort Worth International Airport"
    DestAirportID : "78169",
    DestCityName : "Dallas"
  }
}
```

For the sake of readability only a few of the total fields are included in the example. In the end, the database is composed of only one collection filled with quite complex flight documents. Aggregating queries will collapse all the records in smaller result sets to be forwarded to the Read Oriented database.

5.7 Neo4j

This section is dedicated to the description of how the graph database is structured, in terms of entities, relationships and their main properties.

5.7.1 Entities



Figure 7: The entity types in the graph database

Each airport and airline has a code, a name and multiple numerical statistics about its QoS. A route has numerical statistics as well and it's uniquely identified through its relationships to the origin and destination airport.

5.7.2 Relationships

- Routes: Each route has an origin and destination airport. Additionally, it's been intro-

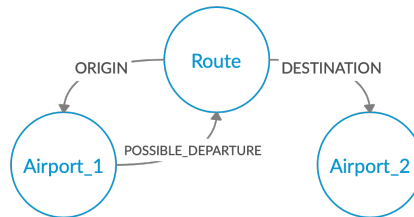


Figure 8: Routes as directional edges in the graph database

duced another relationship from the origin airport to the route. This because it's very common that the database is queried for all possible routes from an airport (or to look for a specific route having that origin airport), and this relationship makes all these queries faster, despite minor storage overhead. Why not representing routes just with an edge between the origin and the destination airport? The reason is that each airline has some dedicated QoS statistics for a route, useful for example to find the best/worst airline for that route, and the best way to store it is through the following kind of relationship.

- Route-Airline service: Of course a route may be served by multiple airlines, and in this



Figure 9: Each route may be served by an airline

case there will be many instances of this relationship for the same route. There are some properties attached to this relationship, representing QoS indicators of that airline for this route. The inverse relationship isn't present because it's not useful to any of the most frequent queries.

- Airport-Airline service: This relationship is very similar to the previous one, but it's

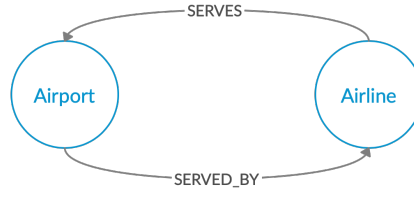


Figure 10: Each airport may be served by an airline

useful to store the percentage of traffic dedicated to each airline inside an airport and vice versa. The two relationships are symmetric and could have been represented through a non directional edge, but since in *Neo4j* this is not possible they are left separated.

In conclusion, here's the possible snapshot of a part of the database, to better understand the whole system:

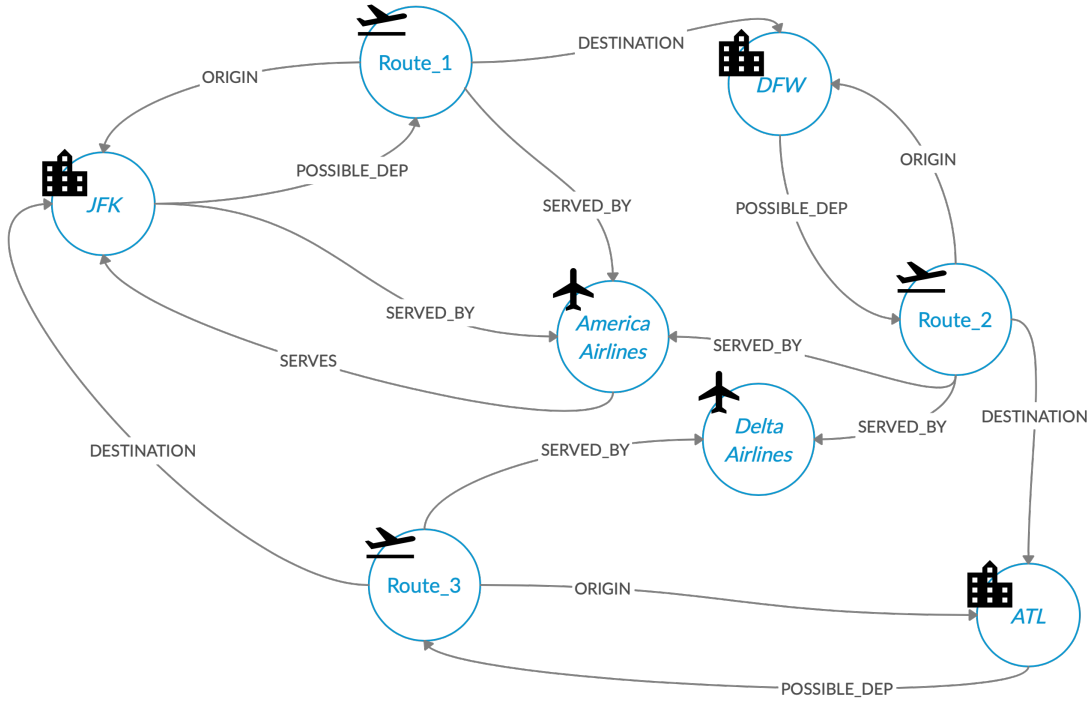


Figure 11: Graph database snapshot

5.8 Technologies

For the implementation, the chosen programming language for both Client and Server is Java. This because it offers the best flexibility for all the different tasks that are requested to both the applications. For example, for the scraping algorithm there's the *JSoup* framework available, and for communication between the applications it's very easy to define TCP sockets in this language. Furthermore, drivers and connectors for both *MongoDB* and *Neo4j* are optimized and highly supported.

6 Admin-Protocol Definition

This section is dedicated to the definition of the application-layer protocol used for the communication between the Client and the Server while fulfilling administration use cases. It relies on SSL (over TCP) transport-layer protocol, so it requires a socket on both the applications. The Admin-protocol is a very simple request-acknowledgement protocol based on text messages.

6.1 Authentication

At the beginning of each interaction with a client, the server asks for an authentication. The messages exchanged are very simple and have this structure:

6.1.1 Client

The client sends an initial request, with the following structure:

Auth <username> <password>

6.1.2 Server

The client sends back a response, that could be a success, or a failure:

- *success*: in this case, the server sends back this message:

Auth successful

- *wrong credentials*: in this case, the server sends back this message:

Auth denied errcode 1

- *already logged*: in this case, the server sends back this message:

Auth denied errcode 2

6.2 Request message

Requests reflect the administration use cases, so that there is a different type of request for each use case. The general structure of a request message is the following:

Request <Request-Type> <comma separated parameters>

The request type field may be one of the following:

1. *credentials*: This request asks for the update of administrator credentials. The parameters are two strings respectively containing new username and password (e.g. *Request credentials admin,my-password*).
2. *update*: This request has no parameters and forces the update of the read oriented database as soon as possible.
3. *limit*: This request sets a storage limit, it has only one parameter that is the maximum size of the storage space (e.g. *Request limit 10*).
4. *replicas*: This request sets replication level on the database. The only parameter is a number taking integer values from 1 to 3, where 1 means a low level of replication and 3 means a high level of replication (e.g. *Request replicas 1*).
5. *year*: Set the starting year of stored data. Older data will be dropped as soon as possible. It takes only one integer parameter representing the year (values between 1970 and 2019).
6. *scrape*: force data scraping algorithm to run as soon as possible. Doesn't take any parameters.
7. *checkout*: Checks out the logged administrator.

6.3 Acknowledgement

The server receives requests and sends back acknowledgements if the request was correctly received. An acknowledgement has this simple structure:

Ack <Request-Type>

If a non authenticated host sends a request to the server, then it will simply ignore the request.

It's important to note that there is at most one admin logged at a time and it sends requests sequentially. The client can't send more requests if it didn't get an acknowledgement back. Another thing is that the server sends the acknowledgement before having fulfilled the request: the acknowledgment means that the server took in charge the command and will process it as soon as possible.

7 Implementation

All concepts and ideas shown in previous sections are applied in this synthetic implementation report which acts as a guide to help the reader understanding the quite large amount of code produced for this application. Reproducing the path followed to describe the application architecture, this section covers both the Server and the Client code, as well as implementation specific topics: databases configuration, interface and tutorial guide. As specified in Section 5.8, the programming language chosen for both Server and Client is Java, so this is from now on taken for granted.

Before passing to the details, a consideration is necessary. Flights data intrinsically has the velocity property: recent flights carry more significant data than old flights. This is the reason why all analytics performed on *MongoDB* have an initial stage which associates a weight to each flight, consisting of the inverse of the distance (in months) between the flight date and the current date. It is used in all averages to give more importance to recent flights, with a linear decay in the weights of flights in statistics with their age.

7.1 Server

The focus in this report is about giving an idea to the reader about the structure of the Server source code, by providing some details about its most important tasks and what classes implement them. To recap, a Server is basically composed of three independent modules:

- An Updater
- An Admin-Protocol Server
- A Scraper

Before providing some details about each one of these software components, it's necessary to explain how the interaction with both *MongoDB* and *Neo4j* databases is handled.

7.1.1 Database interaction

To make the code as simple as possible, for each database technology there is only one Java class, called manager, dedicated to all its interactions. Any module of the Server willing to use a service which will interact with a database needs to invoke a method of its dedicated manager. Because in this application there are two database technologies, two manager classes are defined:

- *MongoDBManager*: an active instance of this class will create a connection with the cluster. All queries interacting with *MongoDB* are realized as a method of this class. This is why Scraper, Updater and Admin-Protocol Server all use this class in their methods.

- *Neo4jManager*: Similarly to the previous one, this is delegated to produce a sort of API for all interactions with *Neo4j* cluster. Mainly it is important only for the Updater module, since it has to periodically refresh the status of the database.

One last thing to note is that these classes do not follow the Singleton pattern. This because the Server application is multi threaded and the best choice is to delegate a dedicated instance to each thread. This enhances data level parallelism and avoids useless concurrence within threads.

7.1.2 Updater

This class is the most complex module, but its code is just a few lines long. This because it consists only of two complex services offered by database managers, which are *MongoDBManager.getUpdatePacket* and *Neo4jManager.update*. The idea is as simple as possible: the first method retrieves all the information to generate a new snapshot of the *Neo4j* database and returns it as an instance of the *UpdatePacket* class, which is defined only for this purpose. The second method takes in input an update packet and proceeds by flushing and recreating the read oriented database. Here is shown the full code of the class:

```
public class Updater extends TimerTask {
    private MongoDBManager mongomanager;
    private Neo4jDBManager neomanager;

    public Updater(MongoDBManager mongomanager, Neo4jDBManager neomanager){
        this.mongomanager=mongomanager;
        this.neomanager=neomanager;
    }
    public void run(){
        UpdatePacket updatePacket=mongomanager.getUpdatePacket();
        neomanager.update(updatePacket);
    }
}
```

The code of the two methods is very long and out of the purpose of this short report. If the reader is interested he/she can look at the source code. In particular *MongoDBManager.getUpdatePacket* contains all analytic queries performed on *MongoDB* for the sake of simplicity, here is shown only one representative aggregation query in mongo shell language, which is present (properly translated in Java) in the code.

```
[
{$addFields: {
  weight: {
    $trunc: {
      $divide: [
        {
          $subtract: [
            '$$NOW',
            {
              $dateFromString: {
                dateString: '$FL_DATE'
              }
            }
          ]
        },
        2592000000
      ]
    }
  }
}},
{$group: {
```

```

    _id: {
      airline: '$OP_UNIQUE_CARRIER',
      origin: '$ORIGIN_AIRPORT'
    },
    serviceCount: {
      $sum: {
        $divide: [
          1,
          '$weight'
        ]
      }
    }
  }
},
{$project: {
  _id: true,
  serviceCount: true
}}, {$sort: {
  "_id.airline": 1,
  "serviceCount": -1
}}
]

```

This aggregation query retrieves, for each airline, a ranking of the most served airports, with the percentage of traffic of that airline dedicated to each airport. The reader can note that the first stage adds the weight field which will be used in the group stage to perform a weighted sum of values.

7.1.3 Admin Protocol Server

This module is a multi thread server which always listens for connections and handles connected clients on separated threads. Here is shown the pseudo code of the class, just to give the reader an idea about how the server actually works:

```

public class Admin_Protocol_Server implements AutoCloseable, Runnable {
    /**
     * Generate a new listening thread
     */
    private void newListener() {
        (new Thread(this)).start();
    }

    public void run() {
        SSLSocket sock;
        try {
            System.out.println("New listener ready for connections...");
            sock = (SSLSocket) server.accept();

            newListener(); //place another thread listening to new connections

            if(!waitForAuth(sock)) {
                sock.close();
                return;
            }

            //Since this point the admin is logged in
            mongomanager = new MongoDBManager();

            do {}
            while(handleRequest()); //handle client requests until a checkout
        }
    }
}

```

```

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Wait for and handle any kind of incoming request.
     * @return false when there's a checkout request.
     */
    private boolean handleRequest() {
        //wait for a request from the client
        //if the request format is correct then parse its parameters and call a more
        //specific handler
        //if the request is a checkout return false, otherwise return true
    }
}

```

As it can be seen in this code snippet, basically the server continuously keeps a thread open to listen for new connections (*newListener()*) and another one to handle an open connection. The client authentication is performed at application level inside the *waitForAuth()* method, where the client has to send the correct admin credentials to be logged in. Of course all accesses to shared variables are made in a thread safe manner using synchronized methods. Before passing to the next component, there's one important thing to say: even though security is out of the purpose of this work, this protocol implements basic security features because it authenticates the server through SSL protocol, which also encrypts the communication. Admin credentials then don't travel in clear and cannot be sniffed by an intruder. Moreover, on the server credentials aren't stored on the server: it stores the hash function of credentials together with a nonce, to protect them even if a hacker manages to steal the credentials store on the server. A dedicated class, called *PasswordManager* implements all of these security functionalities introduced here.

7.1.4 Scraper

This module is the one responsible of gathering raw data from the US Flight Bureau website and parsing them in order to be inserted inside of the Mongo database. The workflow of the Scraper can be divided in three fundamental steps:

- Downloading data
- Decompressing and parsing data
- Inserting data

Downloading data During this step the Scraper connects to the US Flight Bureau website and issues a download request for a certain zip file. The entire data set available on the website is grouped in zip files per each month of the year, so we need to specify in a form which month of each year we want to download. To achieve this, the scraper queries the MongoDB server in order to retrieve the last year and the last month that has been inserted. Once he has this information it calculates the next month and year that has to be download and checks if the corresponding zip file is available on the website by issuing an HTTP request. If the file is available he downloads the file and continues with the next step otherwise it exit the scraping process and the scraping task is rescheduled for the next month.

```

public boolean startScraping() {

    System.out.println("<----- SCRAPING AVVIATO ----->");

    updateScraperViaMongo();
}

```

```

System.out.println("Ultimo anno scaricato: "+lastUpdatedYear+", ultimo mese
    scaricato: "+lastUpdatedMonth);

//check if zip is available

String requestedYear;
String requestedMonth;

if (lastUpdatedYear.equals(Integer.toString(-1))){
    lastUpdatedYear = "2018";
    lastUpdatedMonth = 1;
}
else {
    if (lastUpdatedMonth == 12) {
        lastUpdatedMonth = 1;
        int newYear=ParseInteger(lastUpdatedYear)+1;
        lastUpdatedYear=Integer.toString(newYear);
    } else {
        lastUpdatedMonth = lastUpdatedMonth + 1;
    }
}

requestedYear = lastUpdatedYear;
requestedMonth = Integer.toString(lastUpdatedMonth);

System.out.println("Anno da scaricare: "+requestedYear+", mese da scaricare:
    "+requestedMonth);

String preparedUrl = "https://transtats.bts.gov/PREZIP/On
    _Time_Reporting_Carrier_On
    _Time_Performance_1987_present_"+requestedYear+"_"
        +requestedMonth+".zip";
int code = 0;
try {
    URL u = new URL(preparedUrl);
    HttpURLConnection huc = (HttpURLConnection) u.openConnection();
    huc.setRequestMethod("GET"); //OR huc.setRequestMethod ("HEAD");
    huc.connect();
    code = huc.getResponseCode();
} catch (Exception e) {
    e.printStackTrace();
}

if (code==404){
    System.out.println("Download still not available");
    return false;
}
else {
    scrape();
    return true;
}
}

```

As we can see in the code snippet if the http request sends us back a 404 error we stop the scraping otherwise we call the function `scrape()` that will handle the next steps.

Decompressing data During this second step the scarper unzip the file and then the CSV file inside is analyzed. Note that after completing this phase the scarper deletes all the downloaded files in order to save up space on the disk of the server.

Inserting data and parsing data This step is very important because we want to take just the information we need without wasting space for useless data. The scraper contacts the MongoDB server and insert all the documents gathered from the CSV file. In order to not overwhelm the Mongo server we decided to organize the insert process in batches, each batch containing the insert of 1000 documents. At the end of this phase scraping is complete and even the scraped CSV files is deleted.

```
private void scrape() {
    String requestedYear = lastUpdatedYear;
    String requestedMonth = Integer.toString(lastUpdatedMonth);

    //Retrieve raw data from the web
    try {
        getZipFile();
        unzip(System.getProperty("user.dir")+"/scraperDownloads/"+"report_"
            +requestedYear+"_"+requestedMonth+".zip");
    } catch (Exception e) {
        e.printStackTrace();
        System.err.format("Something went wrong during the retrieve of the file from
            the website");
        return;
    }

    // remove useless files
    try {
        //removes zip file
        cleanDownloads("report_"+requestedYear+"_"+requestedMonth+".zip");
        //removes readme
        cleanTrash();
    } catch (Exception e){
        System.err.format("Something went wrong when trying to delete trash files");
    }

    String documentName =
        "On_Time_Reporting_Carrier_On_Time_Performance_(1987_present)"
        +requestedYear+requestedMonth;

    // Elaborates the raw data in a MongoDB support db and insert it in the final DB
    elaborateDocument(documentName);
    cleanCsv(documentName+".csv");
}
}
```

Note that the inserting process is done inside of the elaborateDocument() function, so that we can do parsing and inserting at the same time.

Once a scraping process has been successfully completed the scraper reschedules itself for another scraping task after 30 days. When this event occurs the scraper will basically start doing again this three steps. We also included a bit more advanced scheduling mechanism, actually if the scraper sees that the date of the MongoDB server is more than 1 month old, it will try to scraper until he reaches the current date. Of course when it finds out that a zip file is not available on the website, it stops this continuous process and reschedules the task to 30 days. Finally we also added the possibility for the administrator of the system (so by using the admin protocol) to start manually the scarping task with a sort of "on demand" option, so that in case there is a problem with the automatic rescheduling procedure an administrator can force it.

7.2 Client

This is a single thread program running on end devices willing to use the services offered by the application. It's out of the scope of this work to explain in detail the whole *JavaFX* GUI, but for the usage of the interface the reader can consult the guide in ???. The description of the client is for some parts the natural complement to what has been shown in Section 7.1, but it also has some aspects that require to be analyzed from scratch. Before listing the main modules, it's important to note that in the client the communication with the database is delegated to a manager (*Neo4jManager* class). Since there's only one thread, just one instance of the manager can be shared among the application, and this is why in this case *Neo4jManager* is indeed implemented following the Singleton pattern.

7.2.1 Admin Protocol Client

When a user wants to log in as an admin, an instance of this class offers an API to easily perform authentication and requests with the server. It uses SSL sockets and it's the complement of *Admin_Protocol_Server*. For a correct usage, the programmer has to invoke *startAuthHandshake()* and if it succeeds then he/she is free to send any request to the server.

7.2.2 Cache

A human user is most likely to ask for the same item several times in the same session. To avoid multiple queries for the same object (and since there's no inconsistency issue) an in memory cache has been defined. In the program there are three instances of this class, one for airports, one for airlines and one for routes. Since it can support three different classes, it's been defined with a generic T parameter:

```
public class Cache<T> extends ArrayList<T> {
    /**
     * Check if there is already an Airport/Airline/Route with a specific identifier
     * into the cache, in that case also update
     * @param tmp Object of type Airport/Airline/Route containing ONLY the identifier
     * @return Complete Object if hit, null if miss
     */
    public T checkCache(T tmp){
        int ind;

        //check for a hit on the cache, and update
        if((ind = this.indexOf(tmp)) != -1) {
            tmp = this.remove(ind);
            this.add(tmp);
            return tmp;
        }

        return null;
    }

    public void updateCache(T tmp){
        if(
            (tmp instanceof Airport && this.size() >= 10) ||
            (tmp instanceof Airline && this.size() >= 10) ||
            (tmp instanceof Route && this.size() >= 30))
        {
            this.remove(0);
        }
        this.add(tmp);
    }
}
```

As it's shown in this code snippet, the class itself is very simple, but the mechanism to support it isn't so, for a main reason: a route is related to two airports, and each airports has statistics about many airlines and so on, creating a potential infinite chain of dependencies that would lead to upload in the cache the whole database each time an object is read. To avoid this the best trade off is the following:

- The identifier of an object is always eagerly loaded (IATA code for airports, unique id for carriers and the two airports for routes)
- When an object is loaded to be put in cache, it has to be complete: completeness means that all of its fields are eagerly fetched.
- In order to be complete, an object must also have its statistics rankings initialized, but all objects in those rankings are instead lazily fetched (i.e. only their identifier is loaded)
- When a field of an object is read the program checks if it's loaded. If not, the object is fetched from the database and completed:

```
public String getName() {
    if(name == null)
        checkCompleteAndFetch();
    return name;
}
```

This is why cached objects have to be complete: otherwise when the user asks for any info about that item the system would have to query the database to complete it, making the cache useless. Note that the cache implements a LRU replacement policy.

7.2.3 Services

This paragraph is dedicated to explain better what are the services offered by the client application to users. Of course all of requirements defined in Section 2.1 are offered (see ?? to know how), but there are some aspects to be clarified and some further services which have been introduced in the implementation phase to make the user life easier:

- What are the benefits of a graph database? Of course the data model is highly suitable for a graph structure, but in particular there are some queries that in a document or relational database would be much more difficult to write or perform. and they are listed here:
 1. Given a route, find all other possible routes having the same destination and an origin airport in the same state.
 2. For each airport, find how many airports it can reach with at most two hops.
 3. Given an airline, find for how many routes it is the best carrier vs the total number of routes it serves.

Despite the simplicity of definitions, these queries would be very difficult to write and perform on a document database. keeping the focus on the third query, let's see how it can be written in a few lines of cypher:

```
MATCH (airline:Airline)
MATCH (airline)<-[:SERVED_BY]-(r:Route)
WITH airline, count(r) AS totalRoutesServed

MATCH (airline)<-[:SERVED_BY]-(r:Route)
OPTIONAL MATCH (r)-[:SERVED_BY]-(otherSb:Airline)
WHERE otherSb.qosIndicator > airline.qosIndicator
WITH airline, totalRoutesServed, otherSb AS candidateRoute, size(collect(otherSb))
AS betterAirlines
```

```
MATCH (airline)<-[candidateRoute]-()
WHERE betterAirlines = 0
RETURN airline, count(betterAirlines) AS bestCarrierCount, totalRoutesServed
```

Note also that with *Neo4j 4.0* it would be even shorter (just 6 lines) thanks to the introduction of sub queries.

- For users it would be very annoying if they have to type the exact name or identifier of an airline or an airport each time they look for some information. For this reason it's been created a service of smart suggestions that use the user input to find the most probable matches basing on the name or the identifier of the item searched. This can very easily tested directly using the application itself.
- Availability has been pointed out as one of the most important non functional requirements of this application. How to implement it on a graph oriented database whose size can stay on a single machine? The answer is simple: *Neo4j Enterprise Edition* offers the possibility to deploy the server over a cluster of nodes, where one is the master and others are followers. All can reply to read operations, while only the master can reply to write operations. The correct way to connect the client to the server would be to use virtual IP addresses into the DNS server, so that load balance and fault tolerance are handled properly. During the test phase it couldn't be possible to use virtual IP addresses, so this is left to further improvements. The server though is effectively deployed on a cluster of three nodes. Sadly the enterprise edition is not open source since *Neo4j 3.5*, so the version installed is the 3.4, which hasn't great limitations for our purpose except the absence of sub queries, that would have been useful as specified at the previous point.

8 User guide

8.1 Initial screen

As the application starts the following window shows up:

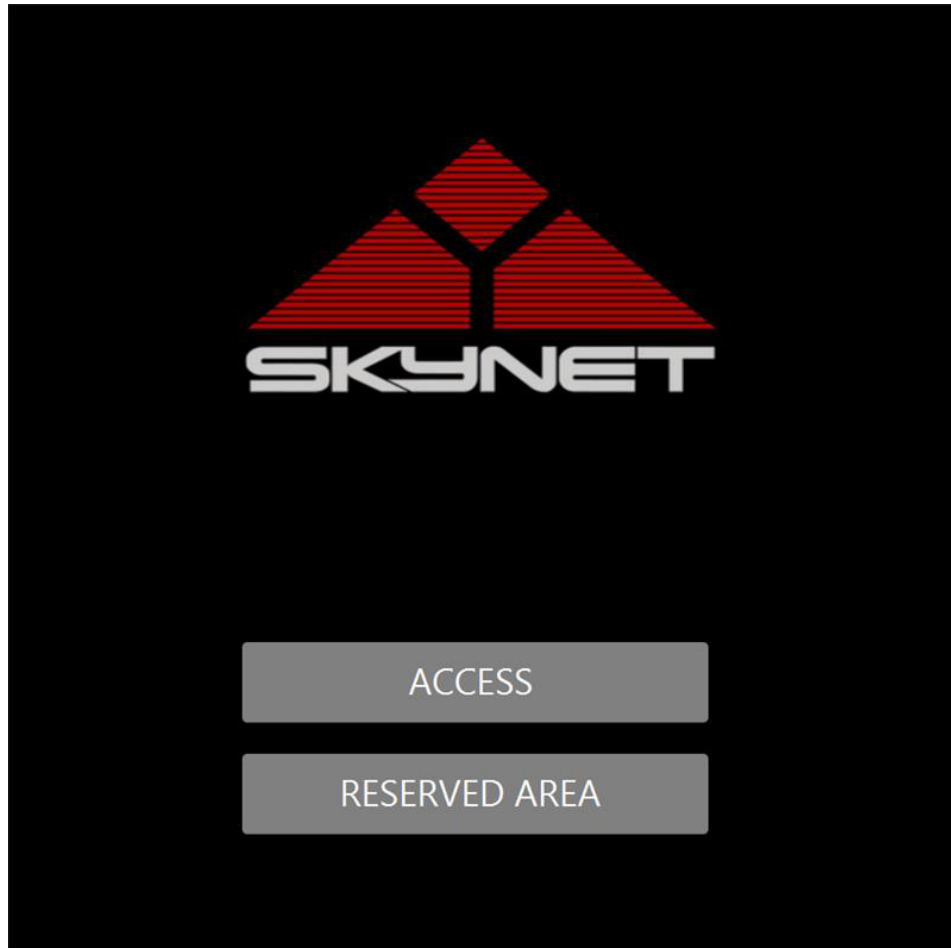


Figure 12: Initial screen interface

1. ACCESS BUTTON: after clicking on this button the user accesses as a normal user.
2. RESERVED AREA BUTTON: after clicking on this button the user accesses to the administrator login screen.

8.2 Overall statistics screen

After clicking on ACCESS button the following window shows up:



Figure 13: overall statistics screen interface

The interface shows two charts plotting the top airlines(on the left) and airports(on the right) ordered by QoS. The user is able to click on the the elements in the charts to access to the statistics screen of the targeted airline(or airport). Below the charts there are some inputs:

1. SELECT AN AIRPORT: the user must insert a valid airport name.
2. SELECT AN AIRLINE: the user must insert a valid airline name.
3. SELECT A ROUTE: the user must insert the origin airport on the left input and the destination airport on the right input.

As the user inserts characters in each input a set of suggested results shows up. After selecting a valid input the user can access to the statistics screen by typing the ENTER key. An error message appears under the inputs if the user inserts an empty value or if the airport/airline doesn't exist. If the route doesn't exist a window with a set of possible alternative routes shows up. The alternative routes are chosen by the system based on the airports selected by the user.

8.3 Airport statistics screen

After selecting an airport the following window shows up:

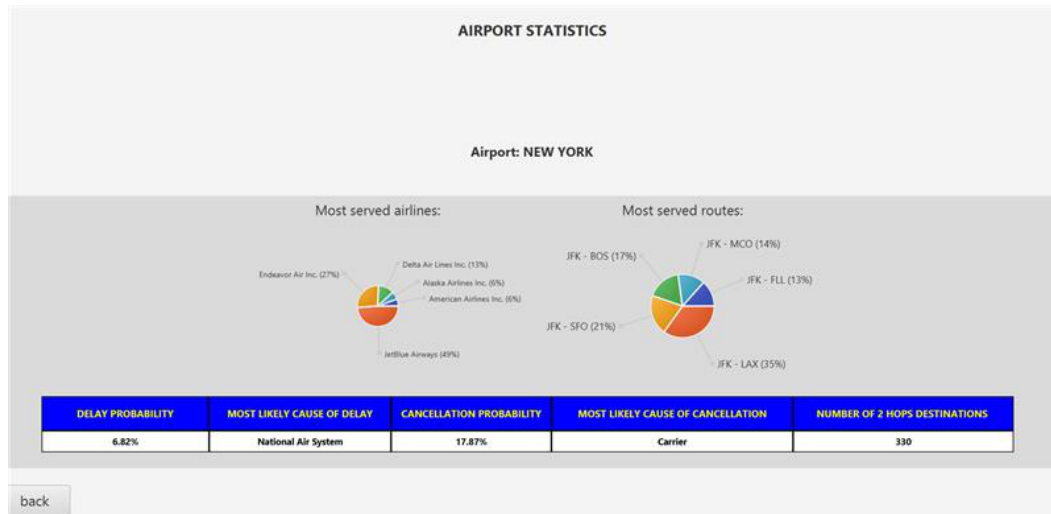


Figure 14: Airport statistics interface

The interface shows two pie charts plotting the most served airlines (on the left) and routes (on the right) by this airport. The user is able to click on the elements in the pie charts to access the statistic screen of the targeted airline (or route). Below the pie charts there are the airport statistics.

8.4 Airline statistics screen

After selecting an airline the following window shows up:

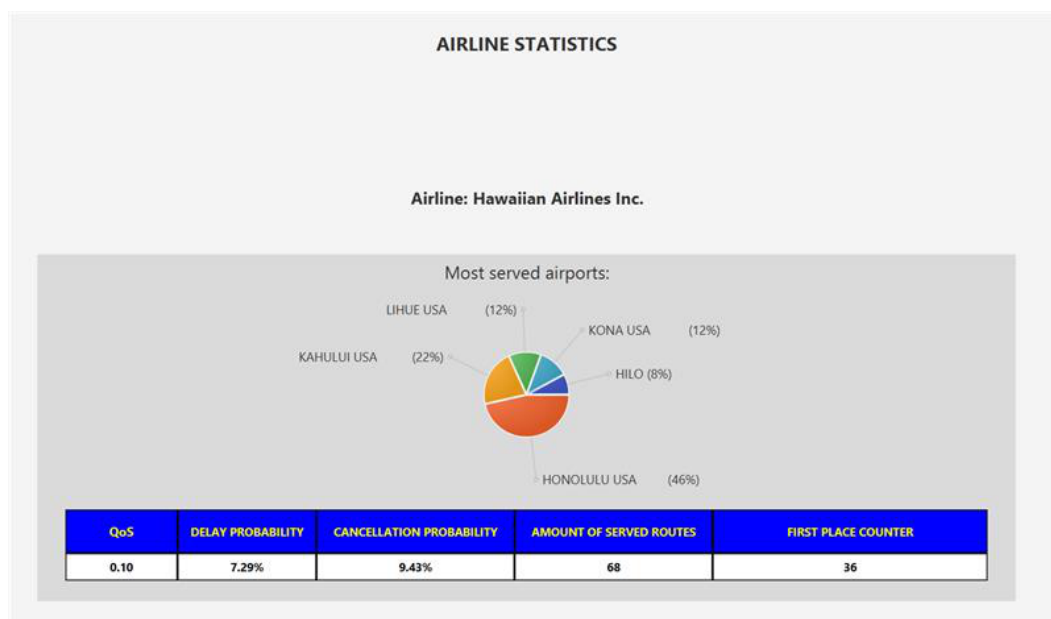


Figure 15: Airline statistics interface

The interface shows a pie charts plotting the most served airports by this airline.

The user is able to click on the the elements in the pie charts to access to the statistic screen of the targeted airport. Below the pie charts there are the airline statistics.

8.5 Route statistics screen

After selecting a route the following window shows up:

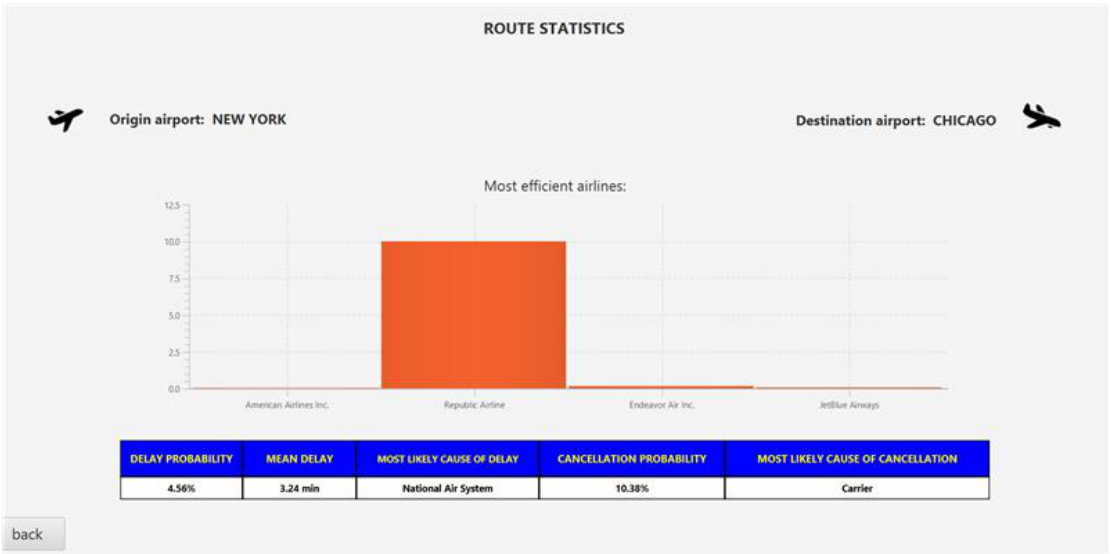
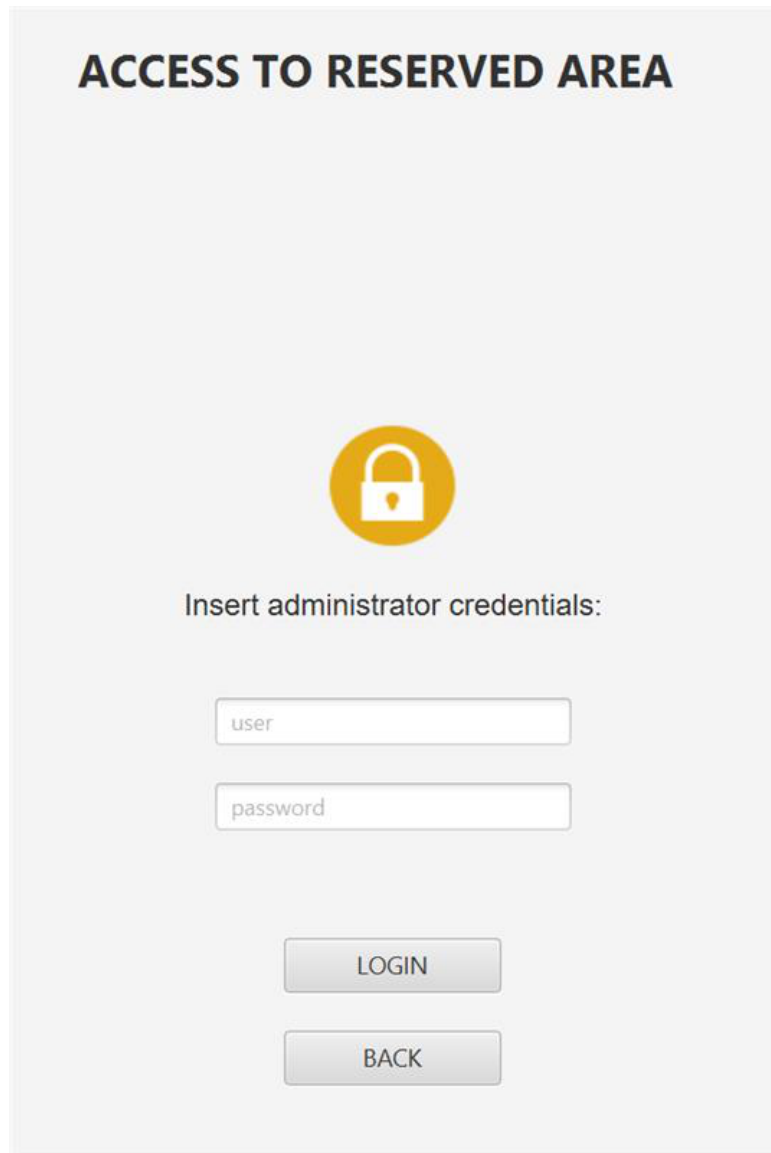


Figure 16: Route statistics interface

The interface shows a chart plotting the top airlines for this route ranked by delay performance. The user is able to click on the the elements in the chart to access to the statistic screen of the targeted airline. Below the chart there are the route statistics.

8.6 Administrator log in screen

After clicking on RESERVED AREA button the following window shows up:



The image shows a login window titled "ACCESS TO RESERVED AREA". It features a yellow padlock icon in the center. Below the icon, the text "Insert administrator credentials:" is displayed. There are two input fields: the first is labeled "user" and the second is labeled "password". At the bottom, there are two buttons: "LOGIN" and "BACK".

Figure 17: Administrator authentication interface

The interface shows two inputs where the administrator can insert the credentials. If the inserted credentials are wrong an error message appears.

8.7 Reserved area screen

After clicking on LOGIN button the following window shows up:

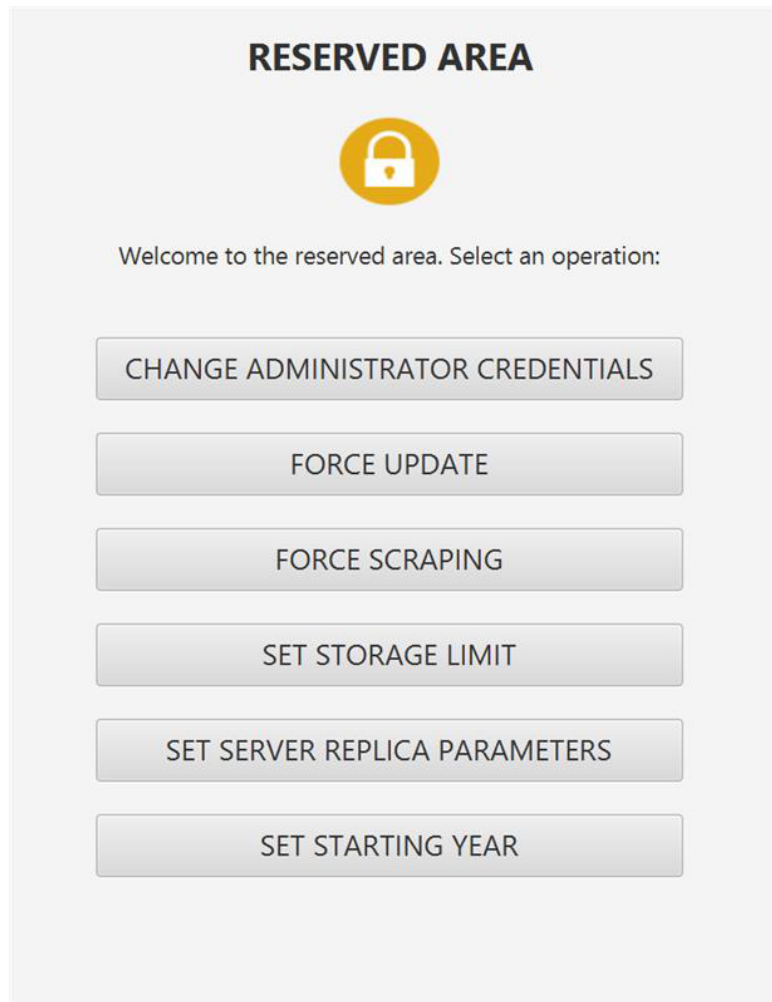


Figure 18: Reserved area interface

The interface shows six buttons:

1. **CHANGE ADMINISTRATOR CREDENTIALS:** a window appears and the administrator can choose new access credentials.
2. **FORCE UPDATE:** the administrator can force the update of statistics in Neo4j database.
3. **FORCE SCRAPING:** the administrator can force the scraping of new data in MongoDB database.
4. **SET STORAGE LIMIT:** a window appears and the administrator can set a new limit in database storage.
5. **SET SERVER REPLICA PARAMETERS:** a window appears and the administrator can set the replication level of the database.
6. **SET STARTING YEAR:** a window appears and the administrator can set a new starting year. The server will delete all records previous to this year