UNIVERSITÀ DI PISA

Large Scale and Multi-Structured Databases

Year 2020/2021

RentNGo

Group members:

ANDREA BULLARI,

GIUSEPPE CANCELLO TORTORA,

MICHELANGELO MARTORANA

Summary

1. INTRODUCTION

RentNGo is a web-application for car rental, in which users can rent cars all over Italy thanks to the fact that the target company is distributed with a high number of offices in the country in order to guarantee an high-level service.

Actually, the service consists of 18 offices distributed to serve a very large number of people. Every office has his own car park. Every car park is situated near the 18 major airports, since most customers of the application are people who travel around Italy.

The vehicles present in car parks are many and of the most varied needs. In fact, cars are divided into three main categories: *Small cars*, *medium cars*, *luxury cars* in such a way that customers can choose the preferred and needed ones. The different categories are divided by the power of the cars (kW).

A customer who enters the website for the first time has to sign-up providing some informations such as Name, Surname, E-mail, Password, Date of Birth.

Once logged-in, the user can explore all the functionalities that the application offers him (e.g., Rent a car, View previous orders etc.).

The application offers different views for workers and admins, through which they can manage users' orders and the entire carparks.

As regarding databases organization, it was decided to use *MongoDB* and *LevelDB*:

- ***MongoDB***, to respect the requirements provided by the project specifications.
- ***LevelDB,*** to better support the chosen features:
    - Cart managing
    - Cars availability

2. REQUIREMENTS ANALYSIS

2.1 Application Actors

The actors of the applications are the *Unregistered User*, the *User*, the *Worker* and the *Admin*.

User and Worker are generalizations of the Unregistered User, while the admin is a generalization of Worker.

When someone logs-in, the application automatically recognizes the role. At first, the application checks if the email and password correspond to a User (since the number of users is greater than the number of workers and admins, the probability of being a user is obviously higher). Then, if the logged person is not a user, the application checks if it is a Worker and finally if it is an Admin. If it is none of them, the system returns an error and asks you to try again.

2.2 Functional and Non-Functional Requirements

The *functional requirements* of this application, divided with respect to the different actors, are the following:

- An Unregistered User can only sign-up, log-in or exit the application.

- A User can log-in.
- A User can log-out.
- A User can rent a car.
- A User can view previous orders.
- A User can view his cart.
- A User can add/remove accessories to his orders.
- A User can delete his account.

- A Worker can log-in.
- A Worker can log-out.
- A Worker can search cars by parameters.
- A Worker can search orders by parameters.
- A Worker can modify orders status.
- A Worker can search users.
- A Worker can change car status.
- A Worker can make cars unavailable.

- An Admin can log-in.
- An Admin can log-out.
- An Admin can search cars by parameters.
- An Admin can search orders by parameters.
- An Admin can modify orders status.
- An Admin can search users.
- An Admin can add/remove workers.
- An Admin can add/remove cars.
- An Admin can promote a worker to admin.
- An Admin can modify workers info.
- An Admin can modify cars info.
- An Admin can view statistics.

The *non-functional requirements* of the application are:

- The applications must be *consistent*, in order to provide correct informations to all users.
- The application store informations in a non-relational Database (*MongoDB*).
- The application must provide *consistency* between *MongoDB* and *LevelDB*.
- The application must guarantee *data availability*.
- The application must guarantee *data replication*.
- The application must guarantee *recoverability* of *LevelDB* data.
- The application must ensure *prevention* against server crashes thanks to replicas.
- The application must be easily *scalable*.
- The application must be *reliable*: no system crashes, exceptions are handled etc.

3. UML DIAGRAMS

3.1 Use-Case Diagram

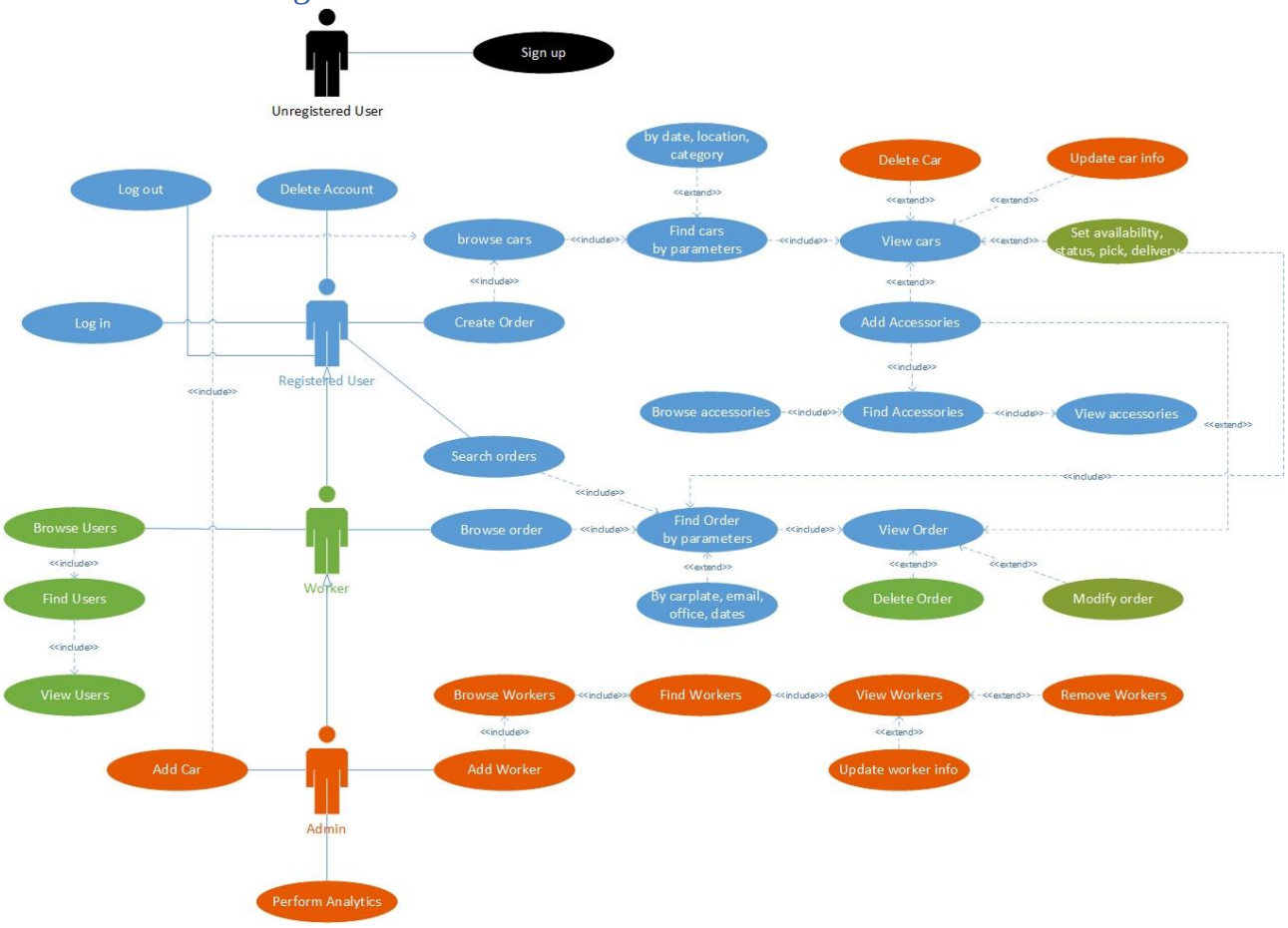


Figure 1: Use-Case Diagram

In the above figure is reported the Use-Case diagram in which we can see: the actors of the application, *Unregistered User*, *User*, *Worker* and *Admin* and their actions**.** The different colors are used to better understand which actions the actors are able to do.

In particular, admins, in addition to *orange* operations, can also perform workers' actions (*green*).

3.2 UML

In the figure 2 are reported the main entities of the application and the relationships between them. *User* and *Worker* are generalized into *Unregistered User*, while *Admin* is generalized into *Worker*.

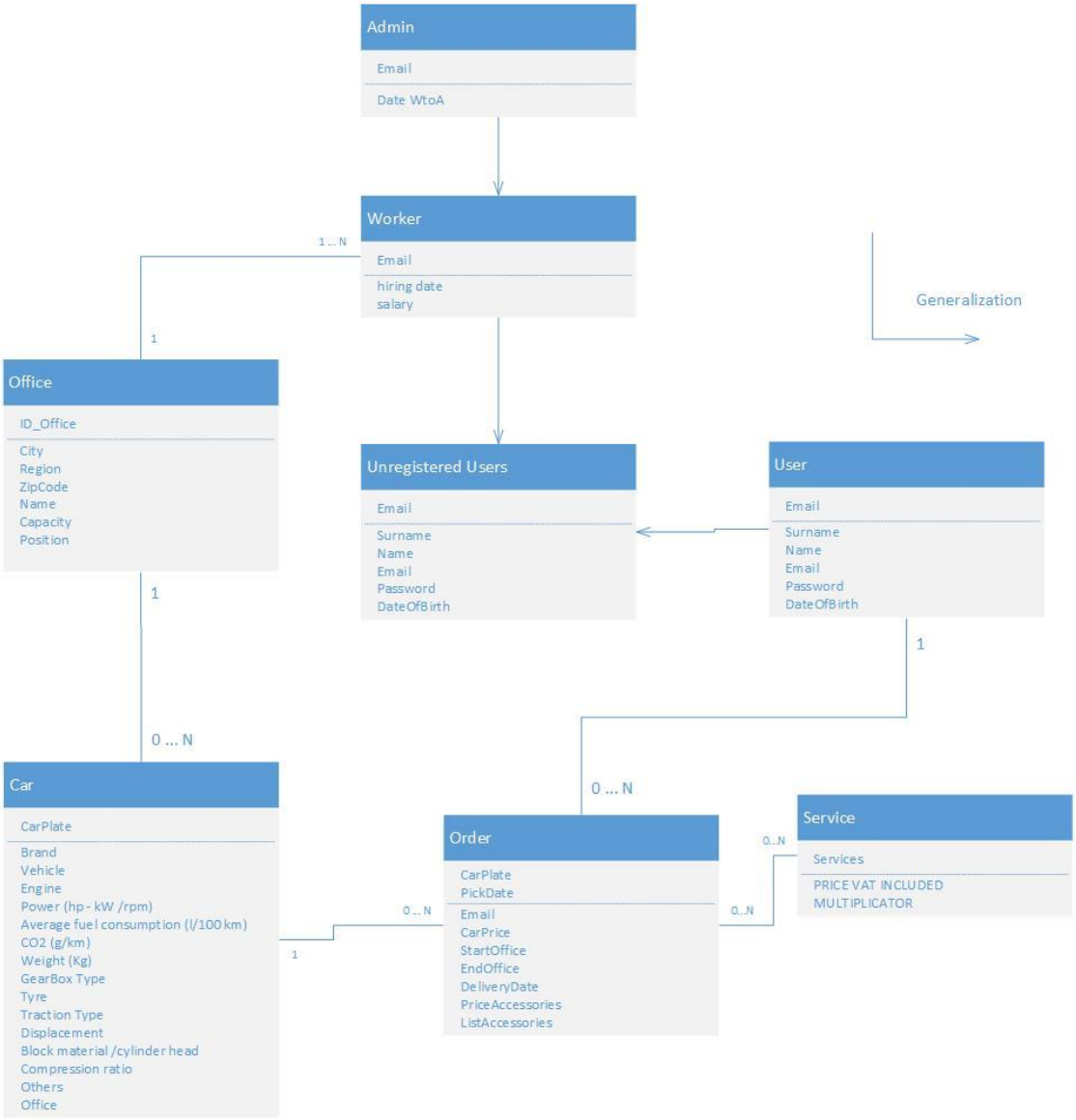


Figure 2: UML Class Diagram

- The carpark's office can contain from 0 to many cars.
- A car, if present, belongs only to a car park.
- A car can be rented zero or many times, so it can be part of zero or N orders.
- The final order can contain only one car.
- An order can be also completed by adding from zero to many additional services.
- An order can be submitted only by a user.

- A user can make zero or many orders.

4. DATABASE ORGANIZATION

4.1 Test Dataset

The Dataset used for RentNGo is an artificial dataset created from various sources. Primarily, the used cars are taken from the official website of the European Environment Agency (EEA). The website contains a dataset of cars with a lot of information about cars. Car Companies are constrained to upload data about co2 emissions of cars for selling them. These cars have been duplicated and random car plates have been assigned to them. After this step, which served to identify uniquely a car, they are distributed to various car parks represented by the office. The offices have been chosen by the major Italy airports.

Another important information about the car's booking are the additional services. They are taken from a real car rental platform.

As regarding users, a dataset of names is used to create sample users. The data about users like the date of birth is computed by choosing a random date in an interval. The emails are created by the concatenation of name, surname and a random email provider. Instead, the users' passwords are created by random choice of alphanumerical characters. Workers and Admins are extracted from the users' dataset and assigned randomly to an office.

Finally, all these data are combined to create about one million orders. Due to privacy, it was not possible to find real data about real car rental transactions, so it was decided to generate them. An order is generated from the extraction of a random user, a random car, random range of dates, random additional services. The city of pick is constrained by the car position. The delivery office is chosen in a random way with a high probability to deliver the car to the same office in which the user picked it.

4.2 MongoDB

MongoDB is used to store all the informations about the application. It consists of 7 collections: *admins, cars, offices, orders, services, users, workers.*

The actors' collections (*admins, users, workers*) contain similar informations, such as *Name, Surname, Date of Birth, Email, Password*, and some others depending on their role (*Salary, HiringDate etc.).*

The main collection is *orders.* It contains several important data about the order, such as:


```
{
  "_id": {
    "$oid": "6002bdff74d0b803341e2baf"
  },
  "CarPlate": "AA491AF",
  "Email": "tinam.henderson@gmail.eu",
  "CarPrice": "22.665663766743602",
  "StartOffice": "Fiumicino",
  "PickDate": 982076400498,
  "EndOffice": "Fiumicino",
  "DeliveryDate": 982767600498,
  "PriceAccessories": "100",
  "ListAccessories": "PAI
  "Status": "Completed"
}
```


Figure 3: MongoDB Order Document

These attributes define totally an order made by a User. In particular:

- The plate of the rented car.
- The email of the user who placed the order.
- The car price per day, depending on the power of the car.
- The office in which the user wants to pick the car.
- The day, expressed in millisecond, in which the user wants to pick the car. This choice was made to simplify DB operations, but this information is hidden to the actors of the application who always use normal format of dates.
- The office in which the user wants to pick the car, if the office is different from the pick one a surcharge is applied.
- The day, expressed in milliseconds, in which the user wants to deliver the car.
- The sum of the price accessories calculated by the price of the single accessory multiplied for his frequency of payment (one time/per day).

- List of accessories chosen by the user in the purchase phase.
- The status of the order in which it is currently set:
    - *Booked*, the user purchases the car.
    - *Picked*, the user physically picks the car from office.
    - *Completed*, the user physically delivers the car to the office.
    - *Deleted*, the order is deleted because the car is in maintenance.

The remaining collections are:

- *Users*, the collection of entire registered users in the application.
- *Workers*, the collection of employees located in the various offices.
- *Admins*, the collection of administrators.
- *Cars*, the collection of entire cars available for rental.
- *Offices*, the collection of the car parks located in Italy.
- *Services*, the collection of the possible accessories which can be chosen by the users in the purchase phase.

The attributes of the previous collection are self-explained and please contact the authors for explanation.

4.3 LevelDB

It was decided to use a Key-Value database, implemented by *LevelDB,* to handle the functionalities chosen in the design phase.

LevelDB is used mainly by the user as cart. The information stored in LevelDB by the user are:

1. All the informations needed to proceed with the order, such as:
    - the *pick office*, the office used to pick the car.
    - the *pick date*, the date in which the user will pick physically the car from the office. The date of pick must be greater/equal than/to the current date.
    - the *delivery office*, the office in which the user will return the car, which can be different from the pick office.
    - the *delivery date*. The date of delivery must be greater than the date of pick.
2. The list of accessories chosen by the user. The user can decide to choose some services like “Road Assistance Plus”, “Additional Driver” etc., while other services are added automatically by the application such as “Young Driver 19/20”, which is added in case the user age is on the range of 18-20 (it’s a cost that the user has to pay due to the low drive experience), or “One Way Same Area”, in case the user returns the car in a different office.

3. The list of the cars selected by the user. At the beginning the system will show all the available cars of the car park of the pick office, selected previously by the user. Only the cars available in the range of dates (pick date and delivery date) will be shown. The system will offer the possibility to choose the category of cars to rent, depending on the power. The application will show 10 cars at a time and will give the possibility to:
    - add a specific car in the cart, to give the possibility to the user to select other cars. When the user decides to proceed with the payment, he has to choose only one car from his cart.
    - Scroll and see other cars.
    - Exit.

These informations about the order will change in case the user decides to create a new order or in case he/she decides to proceed with the payment (in that case the cart becomes empty). The information about the cart is shown only if you choose at least a car.

KEY = EmailUser:*order*

VALUE = PickOffice~PickDate~DeliveryOffice~DeliveryDate

KEY	VALUE
andrea@live.it:*order*	Fiumicino~1614553200000~ Fiumicino~1617228000000
rogerd.mcdaniel@libero.it:*order*	Galileo Galilei~1496158200144~ Galileo Galilei~1496417400144
rickywjr.comer@libero.eu:*order*	Capodichino~1614553200000~ Capodichino~1617228000000

KEY = EmailUser:*cart*

VALUE = CarPlate1~CarPlate2~CarPlate3

KEY	VALUE
andrea@live.it:*cart*	AA016AA
edmunda.adamo@libero.it:*cart*	AA025AA~AA031AA

KEY = CarPlate:*info*

VALUE = Brand1~ Vehicle1~ engine1~ power1

KEY	VALUE
AA016AA:*info*	Volkswagen~Tiguan~2.0L TDI 140 hp 4Motion~140-103/4200
AA025AA:*info*	Mitsubishi~Colt CZ3 Instyle (2004)~1.5L DI-D 95 hp~95-70/4000

KEY = EmailUser:*accessories*

VALUE = Service1, Service2

KEY	VALUE
andrea@live.it:*accessories*	AdditionalDriver,Baby Seat,Young Driver 21/24,
edmunda.adamo@libero.it: *accessories*	Additional Driver,Baby Seat, Vehicle clean,Snow Chain

KEY = EmailUser:*accessoriesPriceDay* (the cost of all the services per day. When you will finalize the order, it will be multiplied for the number of days).

VALUE = value (in double)

KEY	VALUE
andrea@live.it*: accessoriesPriceDay*	10
edmunda.adamo@libero.it: *accessoriesPriceDay*	10

KEY = EmailUser:*accessoriesPriceOneTime* (the cost of all the services per rent. It will be added to the cost of services per day when the order will be finalized).

KEY	VALUE
andrea@live.it: *accessoriesPriceOneTime*	55
edmunda.adamo@libero.it: *accessoriesPriceOneTime*	110

LevelDB is also used to keep track of future orders (considering only the car plate and the dates in which the car is rented in the future, so date of pick and date of delivery). This is done to prevent a user to rent a car in a period in which that car is already rented. Every time a user wants to proceed with the payment, the system checks if, in that range of dates, the specific car is already rented. If the car is available, the system will add the dates of rent in LevelDB preventing someone else to rent that car in those specific dates and then will proceed with the payment.

KEY = CarPlate:*availability*

VALUE = DatePick1,DateDelivery1~DatePick2,DateDelivery2

KEY	VALUE
AA016AA:*availability*	1225297800114,1224520200114
AA025AA:*availability*	1387090800798,1387436400798~ 1449045000271,1449217800271

5. INSTRUCTION MANUAL

As soon as a user enters the website, he can choose whether to log-in (if he is already registered) or to register on the application.

If the user enters the application for the first time, he has to fill a form providing his first name, his last name, his email address and his date of birth.

After logging-in, the application automatically recognizes the role (e.g., User, Worker, Admin).

Depending on the role, the application shows different main menus with the functionalities that the actor is able to do.

5.1 How to use RentNGo – User

A User can:

- Exit the application.
- Create new order.
- Show previous orders.
- Show his cart.
- Delete his account.
- Add/Remove accessories to his order.

Create new order

The application asks user to insert informations about pick office, date of pick, delivery office, date of delivery.

Then the user must select the category of car (small, medium, luxury) and has to choose the cars he wants to add to his cart. In the case the user has not preferences, he is able to see all the cars of all categories.

Show cart

It shows all the selected cars with the corresponding price per day. It also shows all the order informations. The system asks the user if he wants to proceed with the payment.

In the payment phase, the user has to select one car and submit the order.

Add/remove accessories

The application allows user also to add/remove car accessories to his cart.

5.2 How to use RentNGo – Worker

A Worker can:

- Exit the application.
- Search cars.
- Search orders.
- Show user informations.
- Change car status after pick.
- Change car status after delivery.
- Make car unavailable.
- Show cars in maintenance.

Search cars

The worker can search cars by different parameters, such as *Carplate* or *Brand*.

Search orders

The worker can search orders by different parameters, such as *Email*, *Carplate* or *PickOffice* and *Date of pick.*

Search users

The worker can search a user by his *Email.*

Change car status after pick

When a customer rents a car, the status of the car is set to 'Booked' until the user arrives to the office to pick the car. In this moment, the worker changes the status of the car from *'Booked'* to *'Picked'*.

Change car status after delivery

When the customer delivers the car to the office, the worker must check some informations. The worker must find the order by inserting *Carplate*, *Email* and *DeliveryDate* (the date selected in the purchase phase).

If the customer returns the car after the *DeliveryDate*, the application will add a tax of 50.0€ for each day of delay.

Finally, the worker changes the status of the car from *'Picked'* to *'Completed'*.

Make car unavailable

The worker can make a car unavailable. For example, when a car is in very bad conditions and needs some repairs, the worker can temporarily suspend the usage of that car for a range of dates. If the unavailable car was previously rented in the range

of dates in which the car cannot be used, the application will show all the users who rented those cars in that period. In this way, the worker can inform the customers to choose another car.

Show cars in maintenance

The worker can see all the cars in maintenance.

5.3 How to use RentNGo – Admin

An Admin can:

- Exit the application.
- Modify cars.
- Add/Remove cars.
- Find workers.
- Add/Remove workers.
- Promote Worker to Admin.
- Modify workers informations.
- Remove users.
- Show analytics.

The admin can also perform worker operations, such as Search cars, Search orders and Show user informations.

Modify cars

If some office needs to have a larger number of cars, a car can be moved from an office to another one. So, the admin changes the office which a car belongs to.

Add/Remove cars

Admin can decide to replace cars with new ones.
For example, admin can also decide to add new cars with a low level of emissions (CO2) to better respect the environment. These informations are provided by Analytics.

Find Workers

The admin can search worker by his email.

Add/Remove Workers

The admin can decide to hire new workers or can decide to dismiss someone of them. So, he can add/remove workers to/from the system.

Promote Worker to Admin

The admin can decide to promote a *Worker* to *Admin*.

Modify workers informations

The admin can modify workers informations. He can change the salary of the worker or change the office where the employee works in.

Remove users

The admin can decide to remove user's account.

Show analytics

The admin can perform analytics to check the general trend of application.

6. AGGREGATIONS AND INDEXES

6.1 Aggregations

Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together and can perform a variety of operations on the grouped data to return a single result.

The first proposed aggregation is very useful to find the most used cars per office. This information can be used to decide which type of cars to buy.

The first *match* operator matches all the orders whose date is greater than the date passed as parameter. Then, the *sort* operator sorts them in a descending order. The *group* operator counts how many times that car is rented for a specific office. The *project* operator projects them. Finally, with the *limit* operator, the query prints the most three used cars per office.

```
public void getMostUsedCarsPerOffice(String startOffice, long date) {
        Consumer<Document> printFormattedDocuments = new Consumer<Document>() {
            @Override
            public void accept(Document document) {
                System.out.println(document.toJson(JsonWriterSettings.builder().indent(true).build()));
            }
        };

        MongoCollection<Document> myColl = db.getCollection("orders");
        Bson sort = sort(descending(Arrays.asList("count")));
        Bson group = group(Arrays.asList("$StartOffice", "$CarPlate"), sum("count", 1));
        Bson match= match(and(eq("StartOffice", startOffice), ne("Status", "Maintenance"),
                gt("PickDate", date)));
        Bson project = project(fields(include("_id", "count")));
        Bson limit = limit(3);
        myColl.aggregate(Arrays.asList( match,group, sort, limit))
                .forEach(printFormattedDocuments);
    }
```

Figure 4: getMostUsedCarsPerOffice

The second aggregation is useful to find the less eco-friendly office. This query can be used by the admins to change old cars and replace them with new generation cars (e.g., hybrid, electric). This information is also important to act given the continuous changes of restrictions in the country laws.

The *group* operator groups cars by office and by CO2 emissions and then they are limited to find the 3 less eco-friendly office.

```
public void getLessEcoFriendlyOffice(){
        Consumer<Document> printFormattedDocuments = new Consumer<Document>() {
            @Override
            public void accept(Document document) {
                System.out.println(document.toJson(JsonWriterSettings.builder().indent(true).build()));
            }
        };
        MongoCollection<Document> myColl = db.getCollection("cars");
        Bson sort = sort(descending("AvgCO2"));
        Bson group = group("$Office", avg("AvgCO2", "$CO2"));
        //Bson project = project(fields(include( "AvgCO2")));
        Bson limit = limit(3);

        Bson lookup = lookup(
                "offices",
                "_id",
                "Position",
                "Office"
        );
        Bson project = project(fields(include("AvgCO2", "Office.City", "Office.Region", "Office.Name"), excludeId()));

        myColl.aggregate(Arrays.asList( group, sort, limit, lookup, project))
                .forEach(printFormattedDocuments);
    }
```

Figure 5: getLessEcoFriendlyOffice

The third proposed aggregation is very useful to find the users who in a specific year rented cars several time whereas, in next year, they submitted few orders. The application will propose these users some discounts in order to incentivize people to re-use the application.

The first *group* operator checks how many orders a user submitted in the first selected year.

The second *group* operator checks how many orders a user submitted between the second selected year and the first selected one.

Then, with the *merge* operator they are compared.

Finally, the query shows users whose difference in the number of orders between second year and first year is greater than 5.

```
public void searchUserForDiscount(long currentDate, long lastYearDate){
        Consumer<Document> printFormattedDocuments = new Consumer<Document>() {
            @Override
            public void accept(Document document) {
                System.out.println(document.toJson(JsonWriterSettings.builder().indent(true).build()));
            }
        };
        Bson match = match(ne("Status", "Maintenance"));
        Bson group = group("$Email", sum("countCurrent", 1));
        Bson group2 = group("$Email", sum("countPrev", 1));
        Bson matchCurrent = match(gt("PickDate", currentDate));
        Bson matchPrev = match(and(gt("PickDate", lastYearDate),
                lt("PickDate", currentDate)));

        MongoCollection<Document> collection = db.getCollection("orders");

        collection.aggregate(Arrays.asList(
                match,
                matchCurrent,
                group,
                out("currentYear"))).toCollection();

        collection.aggregate(Arrays.asList(
                match,
                matchPrev,
                group2,
                out("prevYear"))).toCollection();

        Bson merge = merge("prevYear");

        MongoCollection<Document> myColl = db.getCollection("currentYear");
        MongoCursor<Document> cursor = myColl.aggregate(Arrays.asList( merge)).cursor();
        while(cursor.hasNext()){
            Document doc = cursor.next();
            if(doc.getInteger("countPrev") != null && doc.getInteger("countCurrent")!=null
                    && (doc.getInteger("countPrev") - doc.getInteger("countCurrent")) > 4){
                System.out.println("User: "+ doc.getString("_id")
                 + ", Current Year Rent: "+ doc.getInteger("countCurrent") + ", Last Year Rent: "
                        + doc.getInteger("countPrev"));
            }
        }
    }
```

Figure 6: searchUserForDiscount

6.2 Indexes

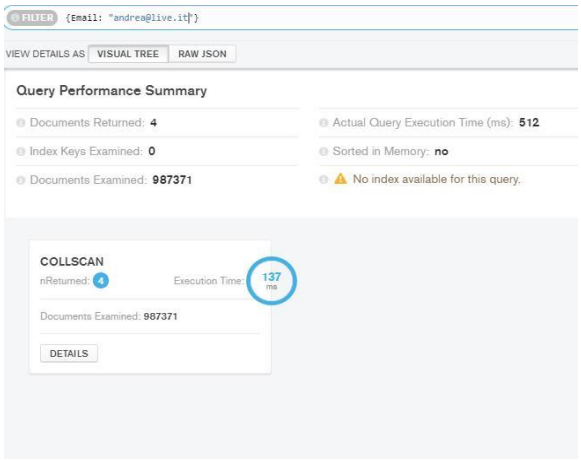
To support the efficient execution of queries in MongoDB, *RentNGo* makes use of indexes.

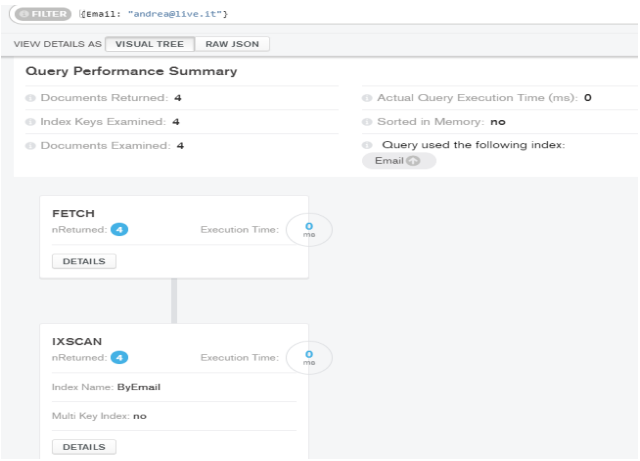
Without use of indexes, certain MongoDB queries should scan every document in a collection to select those documents that match the query statement. Instead, with an appropriate choice of indexes, MongoDB has to scan a limited number of documents to perform queries.

Indexes for collection *Orders*:

- "Email"
- "Delivery Date"

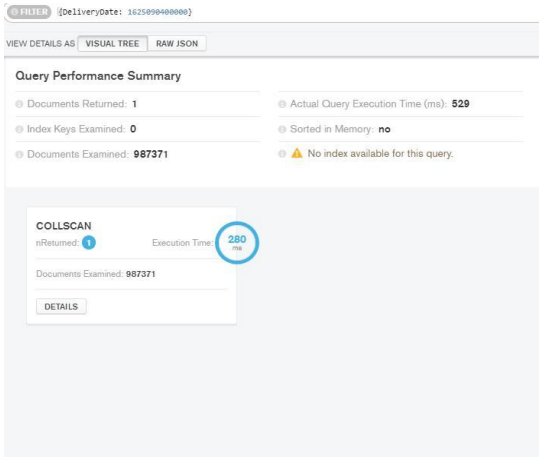
The index *"Email"* is used to scan orders carried out by a single user. In this way, when a worker, or the user himself, want to see past orders, *MongoDB* does not scan the entire collection.

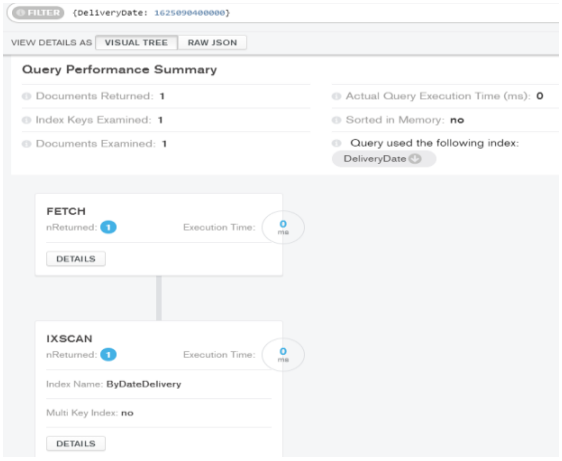




On the left, there is the performance without the usage of index. On the right, there is improvement provided by the usage of index.

The index *"DeliveryDate"* is used for searching the most recent orders.

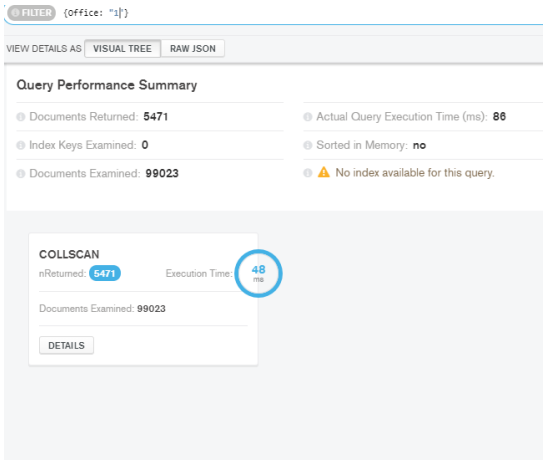


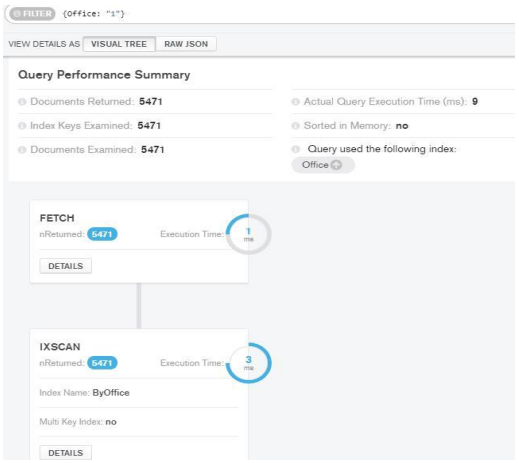


Index for collection *Cars*:

- “Office”

The index *“Office”* is used to find all the cars belonging to a specific office.





7. REPLICAS

In order to simulate a realistic scenario, the application was deployed on virtual machines provided by the *University of Pisa*. 3 VMs were provided to deploy the database as you can see in the following figure.

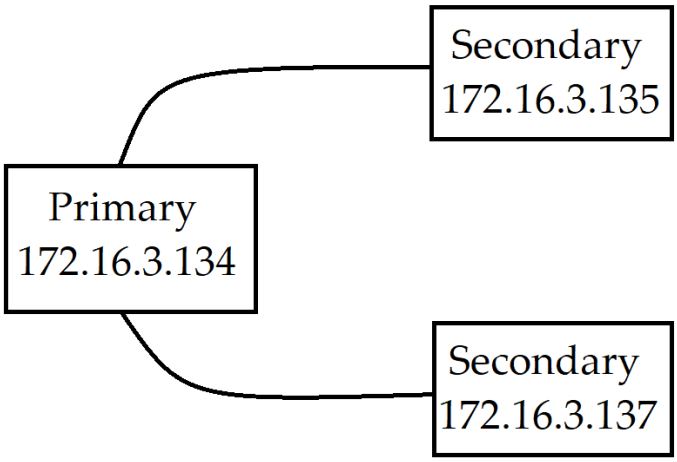


Figure 7: Replica Sets

A *replica set* in MongoDB is a group of mongod processes that maintain the same data set. Replica sets provide redundancy and high availability and are the basis for all production deployments.

Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.

A replica set contains different nodes: only one of these is deemed the primary node, while the other nodes are deemed secondary nodes.

The primary node receives all write operations. A replica set can have only one primary capable of confirming writes with {w: "W2} write concern.

As regarding read preferences, it was decided to use "*Nearest*" to read data from any member node from the set of nodes which respond the fastest.

The secondaries replicate the primary and apply the operations to their data sets such that the secondaries' data sets reflect the primary's data set.

8. CONCLUSIONS

In the actual version of the application, LevelDB is not distributed. For a future implementation it is possible to use Redis that is a distributed version of Key-Value databases.

Other possible improvements, not related to this project purpose, are the addition of specific information about users and cars in MongoDB. For example, telephone number for the users/workers/admins and the current KMs travelled by cars to manage the sales and purchases of cars in car parks.

The code project is available on GitHub:

https://github.com/mickrew/RentNGo.git

To connect with the servers:

- *Username*: workgroup-16
- *Password*: gruppo16

To connect with MongoDB Compass:

mongodb://172.16.3.134:27022, 172.16.3.135:27022, 172.16.3.137:27022/