

Progetto MongoDB (traccia 3)

Gabriele Savoia

Il contesto applicativo dei dati scelti è quello riferito all’ambito **car sharing**. In particolare, si fa riferimento alla tipologia **free-floating** in cui l’azienda che eroga il servizio permette ai propri clienti di noleggiare un certo veicolo dando la possibilità di terminare la corsa in un luogo qualsiasi, a differenza della tipologia *roundtrip* (o station-based) in cui gli utenti sono vincolati a terminare il loro viaggio nella stessa stazione iniziale in cui è iniziato il noleggio.

1. Dataset

Per questo progetto è stato scelto di utilizzare un dataset *creato da zero*, prendendo comunque spunto (modificando ed arricchendo) i dati di uno use-case reale rappresentato dal lavoro riportato nel [paper](#) “Decentralized Social Media Applications as a Service: a Car-Sharing Perspective” ed accessibili direttamente su [Zenodo](#). In particolare è stato presentato e descritto il modello di car sharing ARTICONF che punta a creare una piattaforma decentralizzata basata su blockchain e smart contract in modo da poter gestire un servizio che possa permettere agli utenti l'utilizzo on-demand dei veicoli pagando sulla base dei chilometri percorsi o sulla base del tempo di percorrenza. Il car sharing data model di ARTICONF è composto dai seguenti modelli :

<div>Travels</div> <pre>{ "id": "646736...", "carLicensePlate": "3dfet...", "users": [{ "userId": "Albert", "passengers": 2}, ...], "offerId": "ff746374...", "startPlace.latitude": 59.4487214, "startPlace.longitude": 17.8043161, "endPlace.latitude": 49.93775, "endPlace.longitude": 120.9476, "startDate": 1564185146, "endDate": 1566560112, "suggestedEndPlaces": [{ "endPlace": { "latitude": -19.681836, "longitude": -87.98873}, "reward": 92.57, "suggestedBy": "Gerda"}, ...], "km_traveled": 182.87, "status": 2, "priceBalance": 0, "depositBalance": 0, "rewardBalance": 0, "startedBy": [{ "coord": { "latitude": 31.893, "longitude": 120.89373 }, "moment": 123442, "user": null, },], "score": [], "finishedBy": [], "checkedBy": [], "seats": 2, "rentForTime": true, "totalPrice": 50.98, "observation": "travel observation" }</pre>	<div>Cars :</div> <pre>{ "carLicensePlate": "Wb...", "brand": "Audi", "model": "s6", "color": "Khaki", "seats": 4, "year": 1995, "ownerId": "Allayne", "deleted": false, "state": 1, "observation": "note of the car" }</pre> <div>Offers</div> <pre>{ "id": "474654..", "car": "Audi", "priceForKm": 5.6, "priceForTime": 6.25, "startDate": 1564185146, "endDate": 1566560112, "startPlace.latitude": 59.4487214, "startPlace.longitude": 17.8043161, "startPlace.address": "Mcguire", "endPlaces": [{ "latitude": -19.681836, "longitude": -87.98873, "address": "Ju"}, ...], "available": true }</pre>
--	--

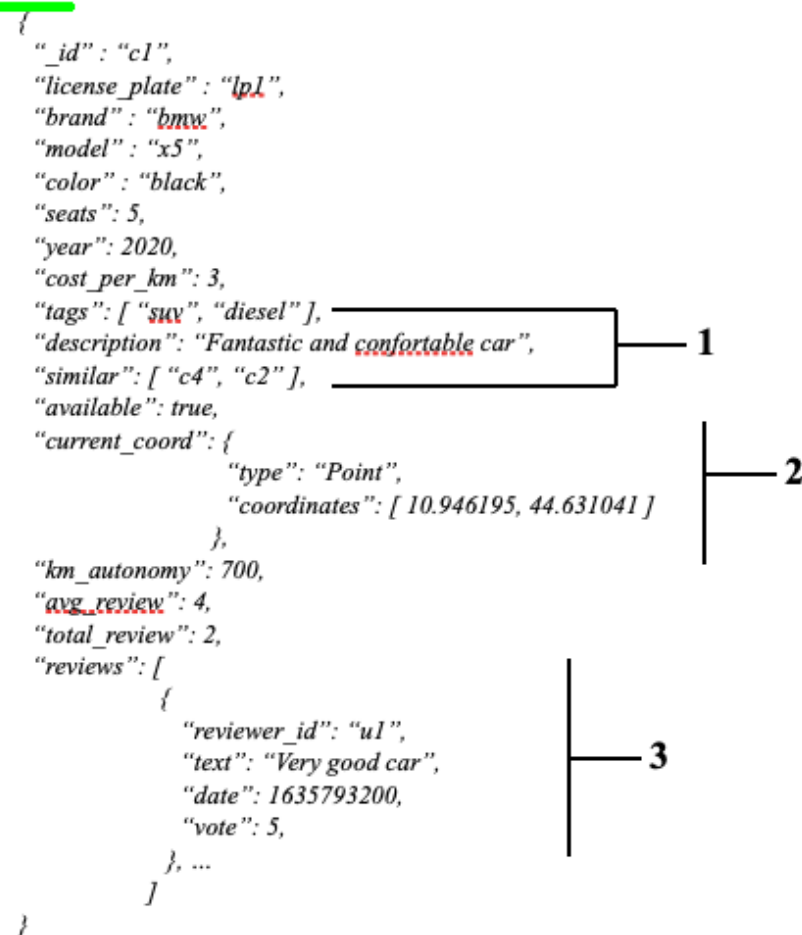
Sulla base di questa tipologia di dati, è stato deciso di creare un **nuovo dataset** in grado di modellare i due aspetti principali di una piattaforma generica di car sharing : le entità **car** e **travel** . Come introdotto in precedenza, il caso d’uso che si vuole gestire riguarda un servizio di car sharing free-floating (simile a quello descritto sopra ma con alcune differenze), in cui le auto a disposizione sono tutte proprietà dell’azienda che eroga il servizio e potenzialmente di marche diverse. Si suppone inoltre che i veicoli siano provvisti di appositi sistemi in grado comunicare informazioni utili ai server dell’azienda (posizione, rilevazione di guasti di diverse tipologie, ...). Gli utenti poi dovranno avere la possibilità di :

- Effettuare il noleggio di auto con eventuali stop senza terminare il servizio;
- Lasciare un feedback sul viaggio effettuato (indicando la qualità del servizio, il rapporto qualità/prezzo e la facilità con cui sono riusciti a trovare l’auto nel punto di partenza del viaggio);
- Effettuare recensioni alle singole auto.

In particolare, di seguito sono riportate le strutture per la rappresentazione dei veicoli e dei viaggi rispettivamente riportate nei documenti `./data/cars.json` e `./data/travels.json`.

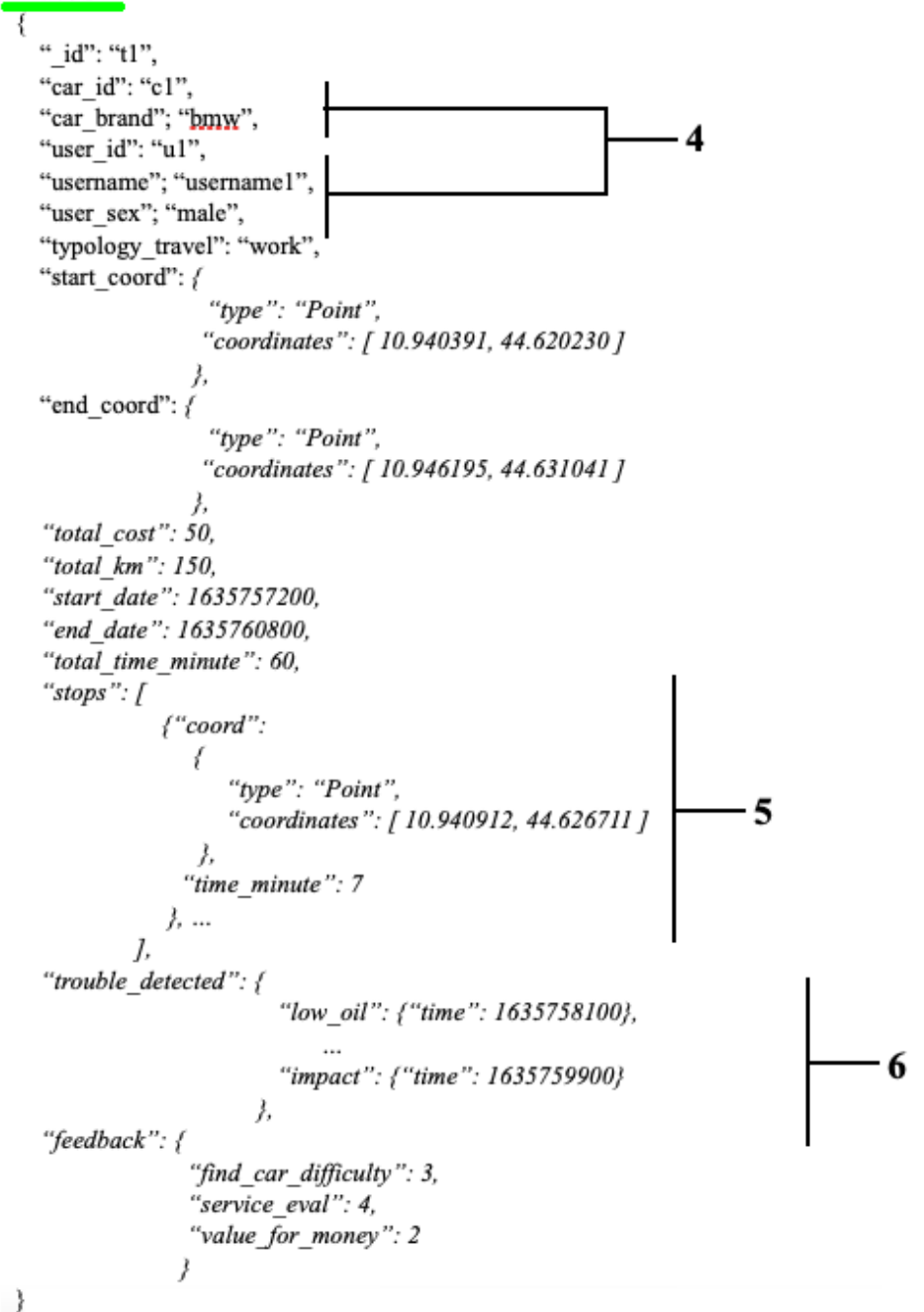
Cars

```
{
  "_id": "c1",
  "license_plate": "lp1",
  "brand": "bmw",
  "model": "x5",
  "color": "black",
  "seats": 5,
  "year": 2020,
  "cost_per_km": 3,
  "tags": [ "suv", "diesel" ],
  "description": "Fantastic and comfortable car",
  "similar": [ "c4", "c2" ],
  "available": true,
  "current_coord": {
    "type": "Point",
    "coordinates": [ 10.946195, 44.631041 ]
  },
  "km_autonomy": 700,
  "avg_review": 4,
  "total_review": 2,
  "reviews": [
    {
      "reviewer_id": "u1",
      "text": "Very good car",
      "date": 1635793200,
      "vote": 5,
    }, ...
  ]
}
```



Travels

```
{
  "_id": "t1",
  "car_id": "c1",
  "car_brand": "bmw",
  "user_id": "u1",
  "username": "username1",
  "user_sex": "male",
  "typology_travel": "work",
  "start_coord": {
    "type": "Point",
    "coordinates": [ 10.940391, 44.620230 ]
  },
  "end_coord": {
    "type": "Point",
    "coordinates": [ 10.946195, 44.631041 ]
  },
  "total_cost": 50,
  "total_km": 150,
  "start_date": 1635757200,
  "end_date": 1635760800,
  "total_time_minute": 60,
  "stops": [
    { "coord": {
      "type": "Point",
      "coordinates": [ 10.940912, 44.626711 ]
    },
      "time_minute": 7
    }, ...
  ],
  "trouble_detected": {
    "low_oil": { "time": 1635758100 },
    ...
    "impact": { "time": 1635759900 }
  },
  "feedback": {
    "find_car_difficulty": 3,
    "service_eval": 4,
    "value_for_money": 2
  }
}
```



Per quanto riguarda il modello **cars** è possibile fare le seguenti considerazioni :

1. *tags* e *similar* sono rappresentati tramite liste i cui elementi variano a seconda della vettura considerata. Nel caso del campo *similar* gli elementi sono il link agli *_id* di altre vetture considerate simili;
2. Utilizzo del formato GeoJSON per effettuare query geo-spaziali in cui il documento innestato contiene un campo *type* che indica il tipo di oggetto GeoJSON (*Point*, *LineString*, *Polygon*, ...) ed un campo *coordinates* che contiene i valori di longitudine e latitudine;
3. Modellazione delle *reviews* come lista di *embedded documents* così soddisfare il *data locality* e migliorare le performance delle query.

In riferimento al modello **travels** invece :

1. E' stato scelto di riportare, insieme al campo *car_id* anche il field *car_brand* del veicolo noleggiato in modo tale da evitare l'operazione di join per quelle query riferite ai travel a cui interessa anche il brand della vettura. Questa scelta porta quindi ad avere performance migliori in fase di interrogazione, a discapito di una denormalizzazione dei dati. E' comunque importante sottolineare che il brand di una vettura ha una probabilità molto bassa di variare nel tempo e quindi di essere aggiornato. Lo stesso discorso è valido anche per i campi *username* e *user_sex* riferiti all'utente *user_id*;
2. Per poter tenere traccia degli stop effettuati durante un noleggio è stato introdotto il campo *stops* identificato tramite una lista di documenti innestati ciascuno dei quali contiene la posizione e la durata della sosta. Si suppone in questo caso che gli stop siano memorizzati in ordine temporale nella lista : primo stop in posizione 0, secondo stop in posizione 1, ... Il campo *total_time_minute* è il tempo totale del viaggio (maggiore della somma dei tempi di stop);
3. Supponendo che le vetture siano in grado di rilevare e comunicare con il server determinati eventi potenzialmente dannosi, il campo *trouble_detected* è stato modellato come un documento innestato la cui chiave rappresenta il tipo di evento e il valore è a sua volta un nested document in quanto ciascun evento, oltre alla data in cui è verificato, potrebbe avere anche altri campi più dettagliati.

2. Operazioni di gestione

Dopo aver installato e fatto eseguire MongoDB in locale, tramite il seguente codice python è creato il relativo client che permette di connettersi al database di nome *BigDataMongo*.

In questa sezione è riportato il codice python relativo alle operazioni più comuni di gestione (inserimento dati, modifica ed eliminazione) rispetto al dataset descritto in sezione 1.


```
        ]
    }
    inserted_car = insert_doc(car_data, 'cars', db)
    print('Inserted car id: '+str(inserted_car))

# Single insert for travel
travel_data = {
    "_id": "t1000",
    "car_id": "c1000",
    "car_brand": "range_rover",
    "user_id": "u1000",
    "username": "username1000",
    "user_sex": "male",
    "typology_travel": "study",
    "start_coord": { "type": "Point",
                     "coordinates": [10.911969, 44.623007]
                   },
    "end_coord": { "type": "Point",
                   "coordinates": [10.946195, 44.631041]
                 },
    "total_cost": 33.0,
    "total_km": 19,
    "start_date": 1635991200,
    "end_date": 1635994800,
    "total_time_minute": 60,
    "stops": [ {"coord":
                 { "type": "Point",
                   "coordinates": [10.929042, 44.625857]
                 },
                "time_minute": 10
              },
              ],
    "trouble_detected": { "low_oil": { "time": 1635758100 }
                        },
    "feedback" : { "find_car_difficulty": 3,
                   "service_eval": 4,
                   "value_for_money": 2
                 }
    }
    inserted_travel = insert_doc(travel_data, 'travels', db)
    print('Inserted travel id: '+str(inserted_travel))
```

Inserted cars ids: ['c1', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'c9', 'c10']
Inserted travels ids: ['t1', 't2', 't3', 't4', 't5', 't6', 't7', 't8', 't9', 't10', 't11', 't12', 't13', 't14', 't15', 't16']
Inserted car id: c1000
Inserted travel id: t1000

2.1 Modifica / Scrittura

Per la modifica, anche in questo caso, sono possibili aggiornamenti mirati ad un singolo o ad un insieme di documenti.

2.1.1 Rendere la vettura di id c1000 NON disponibile per il noleggio

In [5]:

```
query_filter = {"_id": "c1000"}
update = {"$set": {"available": False}}
db.cars.update_one(query_filter, update)

cursor = db.cars.find(query_filter, {"_id": 1, "available": 1})
pd.DataFrame(list(cursor))
```

Out[5]:

	_id	available
0	c1000	False

2.1.2 Per tutti quei viaggi effettuati dall'utente u1000 dal 3 novembre 2021 in poi, impostare la tipologia di viaggio a work

In questo caso l'aggiornamento funzionerebbe su più documenti, ma nel caso di esempio solo 1 fa match con la query di modifica.

In [6]:

```
time_filter = int(datetime(2021, 11, 3, 0, 0, 0).replace(tzinfo=timezone.utc).timestamp())
query_filter = {
    "user_id": "u1000",
    "start_date": {"$gte": time_filter}
}
update = {"$set": {"typology_travel": "work"}}
db.travels.update_many(query_filter, update)

cursor = db.travels.find(query_filter, {"_id": 1, "typology_travel": 1})
pd.DataFrame(list(cursor))
```

Out[6]:

	_id	typology_travel
0	t1000	work

2.1.3 Per tutti quei veicoli di marca *range_rover* e modello *evoque* aggiungere la vettura di id *c19* alle auto simili (SOLO se non presente)

Tramite *\$addToSet* aggiungo l'elemento alla lista solo se non presente.

```
In [7]: query_filter = {
        "brand": "range_rover",
        "model": "evoque"
      }
update = {"$addToSet": {"similar": "c19"}}
db.cars.update_many(query_filter, update)

cursor = db.cars.find(query_filter, {"_id": 1, "similar": 1})
pd.DataFrame(list(cursor))
```

Out[7]:

	_id	similar
0	c1000	[c18, c19]

2.1.4 In riferimento al viaggio di id *t1000*, aggiungere COME PRIMO STOP uno stop localizzato in [10.930907, 44.639626] di durata 30 minuti. (Incrementare quindi il tempo totale di durata di 30 minuti)

Si suppone che gli stop siano memorizzati in ordine (primo stop in posizione 0, secondo stop in posizione 1, ...). Di default il *\$push* inserisce in coda, è necessario quindi specificare *\$position*. In output è riportato prima il tempo totale del viaggio (aggiornato con il nuovo stop inserito che tiene conto sia delle fermate fatte che del tempo di guida) e poi gli stop del viaggio.

```
In [8]: stop_to_add = {
        "coord": {
          "type": "Point",
          "coordinates": [10.930907, 44.639626]
        },
        "time_minute": 30
      }
query_filter = {"_id": "t1000"}
update = {
  "$push": {"stops": {"$each": [stop_to_add], "$position": 0 } },
  "$inc": { "total_time_minute": 30 }
}
db.travels.update_one(query_filter, update)

cursor = db.travels.find(query_filter, {"_id": 1, "stops": 1, "total_time_minute": 1})
for x in cursor:
  print("total time minute : "+str(x['total_time_minute']))
  print("Stops: ")
  for y in x['stops']:
    print(y)

total time minute : 90
Stops:
{'coord': {'type': 'Point', 'coordinates': [10.930907, 44.639626]}, 'time_minute': 30}
{'coord': {'type': 'Point', 'coordinates': [10.929042, 44.625857]}, 'time_minute': 10}
```

2.2 Eliminazione

Come per l'inserimento e la modifica, anche in questo caso sono possibili due tipi di eliminazione :

- Single delete: tramite la funzione **delete_one()** viene eliminato il primo documento che fa match con la query di eliminazione;
- Multiple delete: tramite la funzione **delete_many()** sono eliminati tutti i documenti che fanno match con la query di eliminazione.

2.2.1 Eliminazione della vettura di id *c1000*

```
In [9]: delete_query = {"_id": "c1000"}
db.cars.delete_one( delete_query ).deleted_count
```

Out[9]: 1

2.2.2 Eliminazione di tutti quei viaggi fatti tramite il veicolo di id *c1000*

In questo caso solo un viaggio è stato eliminato, ma in un caso reale potrebbero essercene molteplici.

```
In [10]: delete_query = {"car_id": "c1000" }
db.travels.delete_many( delete_query ).deleted_count
```


Out[10]: 1

3. Query

In questa sezione sono riportate alcune query possibili sui dati di esempio.

3.1 Mostrare la targa e l'anno delle vetture di marchio *toyota* immatricolate dopo il 2005 con almeno 3 recensioni, ordinate dalla più recente alla più vecchia

In [11]:

```
cursor = db.cars.find(
    {
        "brand": "toyota",
        "year": {"$gt": 2005},
        "total_review": {"$gte": 3},
    },
    {"license_plate": 1, "year": 1, "_id": 0}
).sort( "year", -1 )

pd.DataFrame(list(cursor))
```

Out[11]:

	license_plate	year
0	lp9	2016

3.2 Mostrare senza duplicati l'username di quegli utenti che hanno effettuato viaggi senza stop, o se ne hanno fatti, tali che il primo non superi i 5 minuti di attesa

In questo caso, si suppone che gli stop siano salvati in ordine temporale: il primo nella posizione 0 della lista, il secondo nella posizione 1, ...
Nel caso in cui non sono stati fatti stop, la lista è vuota.

In [12]:

```
cursor = db.travels.find(
    {
        "$or": [
            {"stops": { "$size": 0 }},
            {"stops.0.time_minute": { "$lte": 5 }},
        ],
        {"username": 1, "_id": 0}
    ).distinct("username")

pd.DataFrame({"username": list(cursor)})
```

Out[12]:

	username
0	username1
1	username2
2	username3

3.3 Riportare la targa e le coordinate dei suv a diesel, con un prezzo compreso tra i 2 e i 3 euro al chilometro, disponibili per il noleggio e distanti al massimo 300 metri dalla posizione corrente (10.947530, 44.631493). Ritornare i 5 veicoli più vicini ordinati dal più vicino al più lontano

Per poter effettuare query geospaziali con *\$near* è necessario creare un indice di tipo *geosphere* (che corrisponde all' indice *2dsphere* di *mongodb*).
Di default *\$near* effettua l'ordinamento dal più vicino al più lontano (è usato il metro come unità di misura).

In [13]:

```
from pymongo import GEOSPHERE

db.cars.create_index( [ ("current_coord", GEOSPHERE)] )
```

Out[13]:

'current_coord_2dsphere'

```
In [14]: cursor = db.cars.find(
    {
        "tags": { "$all": ["suv", "diesel"] },
        "cost_per_km": {"$gte": 2, "$lte": 3},
        "available": True,
        "current_coord": { "$near":
            {
                "$geometry": { "type": "Point", "coordinates": [ 10.947530, 44.631493 ] },
                "$maxDistance": 300
            }
        },
        {"_id": 1, "license_plate": 1, "current_coord.coordinates": 1}
    ).limit(5)

pd.DataFrame(list(cursor))
```

Out[14]:

	_id	license_plate	current_coord
0	c1	lp1	{'coordinates': [10.946195, 44.631041]}

3.4 Selezionare tutti quei viaggi effettuati dopo il primo novembre 2021 il cui prezzo totale risulti minore di 48 euro, tenendo conto del fatto che per quei viaggi con macchine tesla il prezzo è ridotto a 0.85 del costo totale, mentre per le altre il prezzo è ridotto a 0.95 del costo

Sono considerati i viaggi effettuati dal 2 novembre in poi. Nel DataFrame risultate, la colonna *total_cost* riporta il prezzo originale e non quello scontato. E' possibile vedere come sia stato considerato il viaggio *t12* il cui prezzo di partenza è 50 euro, ma con lo sconto diventa 50*0.95=47.5 che è < 48.

E' utilizzata l'espressione *\$cond* (esplicitando poi il branch condizionale) per gestire i diversi sconti applicati sul prezzo.

```
In [17]: computed_price = {
    "$cond":
        {
            "if": { "$eq": ["car_brand", "tesla"] },
            "then": { "$multiply": ["$total_cost", 0.85] },
            "else": { "$multiply": ["$total_cost", 0.95] }
        }
}

time_filter = int(datetime(2021, 11, 2, 0, 0, 0).replace(tzinfo=timezone.utc).timestamp())

cursor = db.travels.find(
    {
        "start_date": {"$gte": time_filter},
        "$expr": { "$lt": [ computed_price, 48 ] }
    },
    {"_id": 1, "car_brand": 1, "total_cost": 1}
)

pd.DataFrame(list(cursor))
```

Out[17]:

	_id	car_brand	total_cost
0	t2	mercedes	34.0
1	t3	pejeout	35.0
2	t4	bmw	43.0
3	t12	mercedes	50.0
4	t13	pejeout	44.0
5	t15	tesla	37.0
6	t16	pejeout	12.0

3.5 Ritornare senza duplicati gli username di quegli utenti che hanno effettuato più di un viaggio nello stesso giorno (considerando solamente il periodo che va dal 1 novembre 2021 al 14 novembre 2021)

I 2 passaggi principali sono :

1. **Per ricavare gli utenti che hanno effettuato più di 1 viaggio nello stesso giorno** : è stato fatto un raggruppamento sul nuovo campo *day* che è calcolato utilizzando prima la funzione *timestamp_to_date()* che converte il timestamp in una data valida per mongodb, per poi ricavare il giorno dell'anno tramite *dayOfYear*. In questo caso è stato ritenuto sufficiente utilizzare il giorno dell'anno (insieme allo *user_id*) come indice di raggruppamento in quanto la query è limitata ad un intervallo di tempo di 2 settimane e quindi i giorni sono univocamente determinati;
2. **Per riportare gli username senza duplicati** : in questo caso non è possibile utilizzare la *distinct()* come con la find e quindi è stato introdotto un ulteriore raggruppamento per username (riferito al risultato del primo raggruppamento). In questo modo, se nell'arco di tempo dal 1/11/2021 al 14/11/2021, un utente ha effettuato *per più giorni* più di 1 viaggio al giorno, non viene riportato più volte (nei risultati ritornati, l'utente u1 rientrerebbe in questa condizione).

In [19]:

```
def timestamp_to_date(timestamp_field):  
    """  
    Prima è convertito nel formato timestamp in millisecondi, poi nella data di mongodb  
    """  
    return { "$toDate":  
            {  
                "$multiply": [1000, timestamp_field]  
            }  
          }  
  
date_from = int(datetime(2021, 11, 1, 0, 0, 0).replace(tzinfo=timezone.utc).timestamp())  
date_to = int(datetime(2021, 11, 14, 23, 59, 59).replace(tzinfo=timezone.utc).timestamp())  
  
cursor = db.travels.aggregate(  
    [  
        { "$match":  
          {  
              "start_date": { "$gte": date_from, "$lte": date_to },  
          }  
        },  
        { "$group":  
          {  
              "_id":  
                  {  
                      "user": "$user_id",  
                      "day": { "$dayOfYear" : timestamp_to_date("$start_date") }  
                  },  
              "travel_per_day": { "$sum": 1}  
          }  
        },  
        { "$match":  
          {  
              "travel_per_day": { "$gt": 1 },  
          }  
        },  
        { "$group":  
          {  
              "_id": "$_id.user"  
          }  
        },  
        { "$project":  
          {  
              "_id": 0,  
              "user": "$_id"  
          }  
        }  
    ]  
)  
  
pd.DataFrame(list(cursor))
```

Out[19]:

	user
0	u1
1	u2

3.6 Trovare tutte le marche dei veicoli che hanno almeno 2 vetture distanti al massimo 1500 metri dalla posizione corrente (10.947530, 44.631493) e con un' autonomia di almeno 150 km, riportandole in ordine decrescente

In questo caso non è possibile usare *\$near* all'interno della match e per questo è stata utilizzata *\$geonear* i cui campi sono :

- *near* : ovvero il punto su cui calcolare la distanza delle vetture;
- *distanceField* : nome del field di output che contiene la distanza calcolata;
- *maxDistance* : distanza massima rispetto al punto *near*;
- *query* : per filtrare ulteriormente i risultati.

In [20]:

```
cursor = db.cars.aggregate([
    { "$geoNear":
        {
            "near": { "type": "Point", "coordinates": [ 10.947530, 44.631493 ] },
            "distanceField": "distance",
            "maxDistance": 1500,
            "query": {
                "km_autonomy": { "$gte": 150 }
            }
        }
    },
    { "$group":
        {
            "_id": "$brand",
            "count_near": { "$sum": 1 }
        }
    },
    { "$match":
        {
            "count_near": { "$gte": 2 }
        }
    },
    { "$sort":
        {
            "count_near" : -1
        }
    },
    { "$project":
        {
            "_id": 0,
            "brand": "$_id",
            "count_near": 1
        }
    }
])

pd.DataFrame(list(cursor))
```

Out[20]:

	count_near	brand
0	2	tesla
1	2	bmw

3.7 Per ogni utente mostrare la media delle recensioni per marca riordinando l' output, a parità di utente, in ordine decrescente di recensione media

In questo caso il field *total_review* riferito ad ogni vettura non è di aiuto, ma è necessario fare l'operazione di unwind alla lista delle recensioni ed infine raggruppare per utente e brand.

In [477...

```
cursor = db.cars.aggregate([
    { "$unwind":
        {
            "path": "$reviews"
        }
    },
    { "$group":
        {
            "_id": {
                "user": "$reviewsReviewer_id",
                "brand": "$brand",
            },
            "avg_reviews": { "$avg": "$reviews.vote" },
        }
    },
    { "$sort":
        {
            "_id.user": 1,
            "avg_reviews": -1
        }
    },
    { "$project":
        {
            "_id": 0,
            "user": "$_id.user",
            "brand": "$_id.brand",
            "avg_reviews": 1
        }
    }
])

pd.DataFrame(list(cursor))
```

Out [477...

	avg_reviews	user	brand
0	5.00	u1	bmw
1	3.75	u1	tesla
2	2.00	u1	mercedes
3	5.00	u14	maserati
4	5.00	u2	pejeout
5	4.50	u2	fiat
6	3.00	u2	bmw
7	5.00	u21	maserati
8	4.50	u21	toyota
9	3.00	u21	porsche
10	2.50	u21	fiat
11	3.00	u3	mercedes
12	1.00	u3	tesla
13	4.00	u4	pejeout
14	4.00	u5	bmw
15	2.00	u54	toyota
16	5.00	u7	fiat
17	4.00	u7	bmw
18	4.00	u8	toyota
19	5.00	u9	tesla

3.8 In riferimento alla finestra temporale dall'1 novembre 2021 al 7 novembre 2021, riportare per ogni giorno della settimana il numero di viaggi che sono iniziati ad una distanza non superiore a 5000 metri dal punto (10.913326, 44.622403)

Anche in questo caso è necessario definire l' indice per effettuare query geospaziali. L'aggregazione è definita sul nuovo campo *day_of_week* che è calcolato utilizzando prima la funzione *timestamp_to_date()* definita nella 3.5 per poi ricavare il giorno della settimana tramite *\$dayOfWeek* che associa ad ogni giorno un numero (1 per la domenica e 7 per il sabato).

```
In [21]: from pymongo import GEOSPHERE

db.travels.create_index( [("start_coord", GEOSPHERE)] )
```

Out[21]: 'start_coord_2dsphere'

```
In [22]: date_from = int(datetime(2021, 11, 1, 0, 0, 0).replace(tzinfo=timezone.utc).timestamp())
date_to = int(datetime(2021, 11, 7, 23, 59, 59).replace(tzinfo=timezone.utc).timestamp())

cursor = db.travels.aggregate(
    [
        { "$geoNear":
            {
                "near": { "type": "Point", "coordinates": [ 10.913326, 44.622403 ] },
                "distanceField": "distance",
                "maxDistance": 5000,
                "key": "start_coord",
                "query": {
                    "start_date": { "$gte": date_from, "$lte": date_to }
                }
            }
        },
        { "$group":
            {
                "_id": {
                    "day_of_week" : { "$dayOfWeek" : timestamp_to_date("$start_date") },
                },
                "count": { "$sum": 1 }
            }
        },
        { "$sort":
            {
                "_id.day_of_week": 1
            }
        },
        { "$project":
            {
                "_id": 0,
                "day_of_week": "$_id.day_of_week",
                "avg_travels_count": "$count"
            }
        }
    ]
)

pd.DataFrame(list(cursor))
```

Out[22]:

	day_of_week	avg_travels_count
0	1	2
1	2	4
2	3	2
3	4	1
4	5	3
5	6	1
6	7	2

3.9 Mostrare per ogni marca, la percentuale di viaggi in cui è stata rilevata la problematica *engine_problem*, per poi ordinare i risultati in ordine decrescente di percentuale

In questo caso l'idea è quella di effettuare una prima proiezione calcolando un nuovo campo *engine_problem* che assume i valori :

- 1 se il field *trouble_detected.engine_problem* esiste
- 0 se non esiste

Dal momento che dentro alla *\$cond* non è possibile inserire *\$exists* , è stata utilizzata la *\$not* che ritorna 0 se il campo non esiste (danno non verificato) e 1 se il campo esiste (danno verificato).
A questo punto avviene il raggruppamento per marca di auto per poi contare (per ciascuna marca) i viaggi totali ed infine sommare i valori del campo *engine_problem* creato nella prima proiezione così da conteggiare solamente i viaggi in cui si è verificato l'evento *engine_problem*.

```
In [23]: cursor = db.travels.aggregate([
    { "$project":
      {
        "car_brand": 1,
        "engine_problem": { "$cond":
          [
            {"$not": ["$trouble_detected.engine_problem"]},
            0,    # se TRUE    --->  NON verificato engine_problem
            1     # se FALSE   --->  verificato engine_problem
          ]
        }
      }
    },
    { "$group":
      {
        "_id": "$car_brand",
        "travels_count": { "$sum": 1 },
        "engine_problem_counts": { "$sum": "$engine_problem" }
      }
    },
    { "$project":
      {
        "_id": 0,
        "car_brand": "$_id",
        "engine_problem_counts": 1,
        "travels_count": 1,
        "%_engine_problem": { "$multiply":
          [
            { "$divide": ["$engine_problem_counts", "$travels_count"] },
            100
          ]
        }
      }
    },
    { "$sort":
      {
        "%_engine_problem": -1
      }
    },
  ],
)

pd.DataFrame(list(cursor))
```

Out[23]:

	travels_count	engine_problem_counts	car_brand	%_engine_problem
0	1	1	toyota	100.0
1	1	1	maserati	100.0
2	4	3	bmw	75.0
3	2	1	mercedes	50.0
4	3	0	tesla	0.0
5	1	0	fiat	0.0
6	3	0	pejeout	0.0
7	1	0	porsche	0.0

3.10 Trovare il prezzo medio pagato per tutti i viaggi riferiti al noleggio della vettura targata *lp1*

In questo caso, essendo che il campo targa (*license_plate*) non è presente nel documento *travels*, è necessario effettuare un'operazione di *join* (molto costosa).

Inoltre in questo caso il raggruppamento avviene su tutti i documenti filtrati e quindi nel campo *_id* è specificato *None*.

Essendo che per ogni viaggio è associata una sola vettura, allora l'utilizzo dell' *unwind* in questa query non è obbligatorio.

In [24]:

```
cursor = db.travels.aggregate([
    { "$lookup" :
        {
            "from": "cars",
            "localField": "car_id",
            "foreignField": "_id",
            "as": "car_list"
        }
    },
    { "$unwind":
        {
            "path": "$car_list"
        }
    },
    { "$match":
        {
            "car_list.license_plate": "lp1"
        }
    },
    { "$group":
        {
            "_id": None,
            "price_avg": { "$avg": "$total_cost" }
        }
    },
    { "$project":
        {
            "_id": 0,
            "price_avg": 1
        }
    }
])

pd.DataFrame(list(cursor))
```

Out[24]:

	price_avg
0	61.0

In []: