Queen Mary University of London

ECS650/ECS789 Semi-structured data and advanced data modelling
Coursework 2: MongoDB design and implementation

CW2 Group 7:
Govor, Natalia (Student Id: 170658393)
Lyapina, Ekaterina (Student Id: 160984099)
Novinfard, Soheil (Student Id: 170801896)

Supervisor:
Anthony Stockman

During this coursework, the database for a taxi company has been designed and built. The system stores information about the taxi company, drivers, bookings and payments made by customers. At the beginning of the database design the relational model was used and UML diagram describing relations between entities was created. Then, the relational model was denormalised and adopted to NoSQL. The coursework also includes a list of subject area assumptions made before the implementation of the system. Moreover, examples of using performance monitoring tools are provided in the end of the report.

1. Assumptions

Assumptions, which were made for this implementation:

1) Each driver has one shift.
2) Taxi company operates in one timezone.
3) Taxi driver is also an owner of the car.
4) Taxi driver can have several cars.
5) Each car MOT test result is valid for 1 year.
6) Regular bookings contains time of the day when the car should arrive.
7) In the relational design the majority of tables have surrogate unique key - Id. This key helps to link table with others.

2. Database design (relational approach)

Database design was started from creating a UML diagram which describes information about entities of the taxi company and represents relations between them. The created diagram is shown on Figure 1.

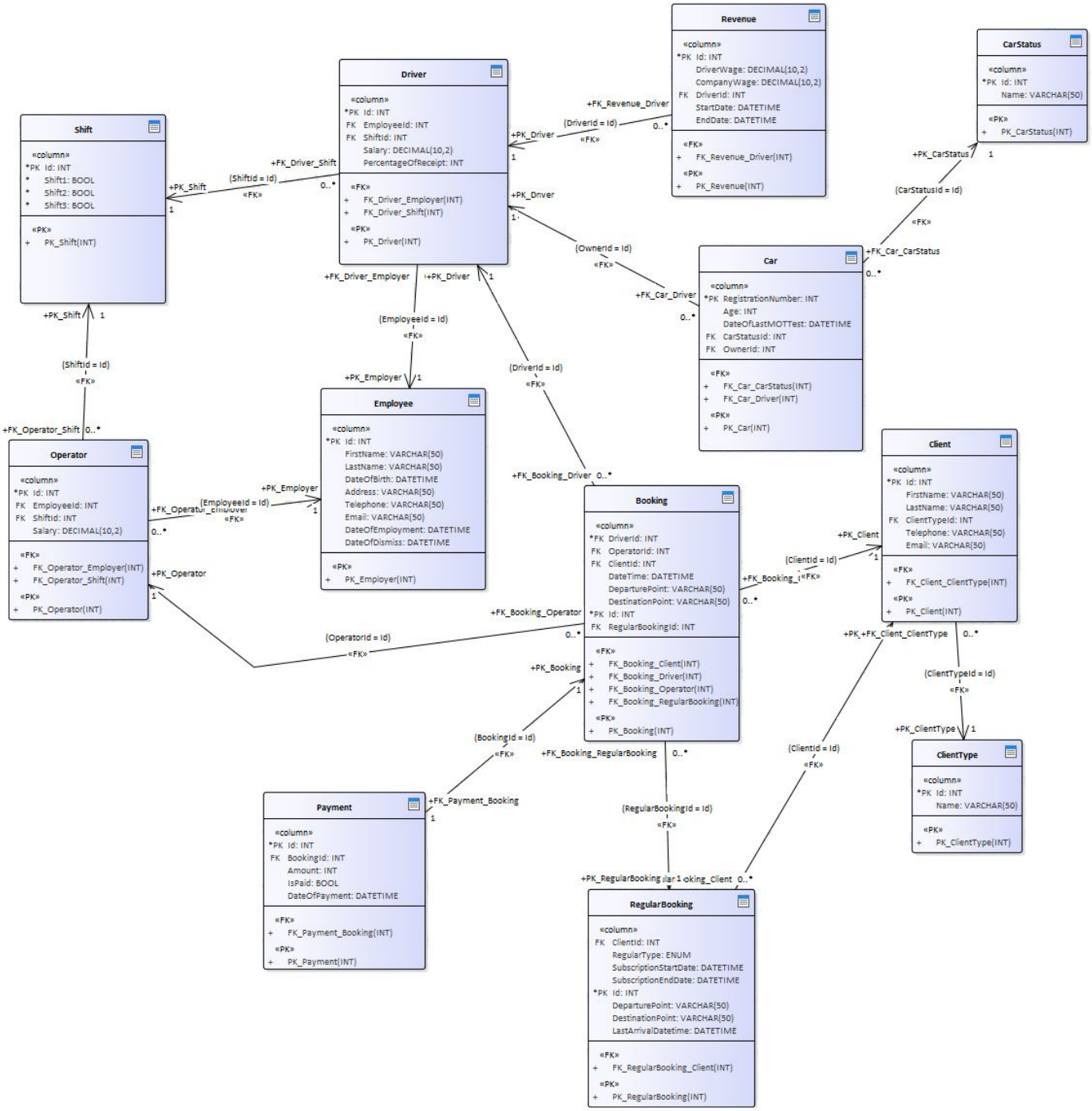

Figure 1. UML diagram of taxi company database

Relational database design consists of the following tables, representing subject area, according to the following logic:

1) Employee table: details of the employees, their accommodation, contact details and dates of employment and dismiss. The details of employee columns are displayed in Table 1.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
FirstName	**Varchar**	
LastName	**Varchar**	
DateOfBirth	**DateTime**	
Address	**Varchar**	
Telephone	**Varchar**	
Email	**Varchar**	
DateOfEmployement	**DateTime**	
DateOfDismiss	**DateTime**	

Table 1. Employee table

2) Driver table: information about driver's rate type (fixed salary or percentage of receipts basis), driver's shift, foreign key to driver's information in Employee table. The details of driver columns are displayed in Table 2.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
EmployeeId	**Int (Foreign key)**	This column is linked with Employee table, one-to-many relationship type.
ShiftId	**Int (Foreign key)**	Each driver has one shift. This column is linked with Shift table, one-to-many relationship type.
Salary	**Decimal**	One of these 2 columns is optional. This means that driver uses the certain scheme for rate type and
PercentageOfReceipt	**Int**	

		the existence of exact column value (another is NULL) reference to a certain type of rate: fixed salary or percentage of receipts basis.

Table 2. Driver table

3) Operator table: operator's shift, operator's salary, foreign key to operator information in Employee table. Details of operator columns are shown in Table 3.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
EmployeeId	**Int (Foreign key)**	This column is linked with Employee table, one-to-many relationship type.
ShiftId	**Int (Foreign key)**	This column is linked with Shift table, one-to-many relationship type.
Salary	**Decimal**	

Table 3. Operator table

4) Shift table: during what shift the employee is available. The drivers and operators work a shift system in order to ensure the Company is effective over 24 hours. The details of Shift columns are provided in Table 4.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
Shift1	**Bool**	Shift 1 is between 08:00 and 16:00.
Shift2	**Bool**	Shift 2 is between 16:00 and 00:00.
Shift3	**Bool**	Shift 3 is between 00:00 and 08:00.

Table 4. Shift table

5) Car table: details of the registration number, age, date of last MOT test, status of cars, details of who owns the car. The details of car columns are provided in Table 5.1.

Name of the column	Data type	Comment
RegistrationNumber	**Int (Primary key)**	
Age	**Int**	
DateOfLAstMOTTest	**DateTime**	
CarStatusId	**Int (Foreign key)**	This column is linked with CarStatus table, one-to-many relationship type.
OwnerId	**Int (Foreign key)**	This column is linked with Driver table, one-to-many relationship type. We assume, that taxi driver is also an owner of the car. We assume, that driver can have several cars.

Table 5.1. Car table

CarStatus is a lookup table which contains names of car statuses. Details are provided in Table 5.2.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
Name	**Varchar**	Column has the values: roadworthy, in for service, awaiting repair, written off.

Table 5.2. CarStatus table

6) Client table: information about client. Details of Client columns are provided in table 6.1.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
Firstname	**Varchar**	
LastName	**Varchar**	

ClientTypeId	**Int (Foreign key)**	This column is linked with Client table, one-to-many relationship type.
Telephone	**Varchar**	
Email	**Varchar**	

Table 6.1. Client table

ClientType is a lookup table which contains names of client types (corporate and private). Details of columns are provided in Table 6.2. Corporate clients can make regular bookings.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
Name	**Varchar**	This column has two possible values: “private”, “corporate”

Table 6.2. ClientType table

7) Booking table: details of all bookings (taken over the phone and performed regular bookings). Columns details are provided in Table 7.1.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
DriverId	**Int (Foreign key)**	This column is linked with Driver table, one-to-many relationship type.
OperatorId	**Int (Foreign key)**	This column is linked with Operator table, one-to-many relationship type.
ClientId	**Int (Foreign key)**	This column is linked with Client table, one-to-many relationship type.
DateTime	**DateTime**	
DeparturePoint	**Varchar**	
DestinationPoint	**Varchar**	
RegularBookingId	**Int (Foreign key)**	This column is linked with

		RegularBooking table, one-to-many relationship type. If column is NULL then it is not regular booking.

Table 7.1. Booking Table

Clients can make regular bookings (daily bookings, once a week, etc.), the details of which must be stored.
RegularBooking table: information about regular booking. Columns details are shown in Table 7.2.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
ClientId	**Int (Foreign key)**	This column is linked with Client table, one-to-many relationship type.
RegularType	**Enum**	Regular bookings have type which describes how often the booking should occur. It will be stored as unstructured text (for instance “each week”, “each month”). This column is recommended to be analysed in an logic tier of architecture. Possible values: daily bookings, once a week, etc.
SubscriptionStartDate	**DateTime**	Date from which a contract for regular bookings was subscribed.
SubscriptionEndDate	**DateTime**	Date from which a contract for regular bookings will expire.
DeparturePoint	**Varchar**	
DeatinationPoint	**Varchar**	
LastArrivalDatetime	**DateTime**	This column contains the last date and time when the car arrived. Date of new

		arrival will be calculated with regard to this column and RegularType.

Table 7.2. RegularBooking table

Comments: The values of RegularType column in RegularBooking table are strictly defined in a system and these values are considered not to be changed. Because of that assumption, the data type enum is used in relational database for the table RegularBooking. In contrast, for CarStatus and ClientType lookup tables are created because we assume that those entities might be changed more frequently.

8) Payment table: information about the payment related to booking. Columns details are provided in Table 8.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
BookingId	**Int (Foreign key)**	This column is linked with Booking table, one-to-one relationship type.
Amount	**Int**	
IsPaid	**Bool**	
DateOfPayment	**DateTime**	

Table 8. Payment table

9) Revenue table: details about revenue calculated for specific period, the amount of money earned by drivers and the Company. Columns details are shown in Table 9.

Name of the column	Data type	Comment
Id	**Int (Primary key)**	
DriverWage	**Decimal**	
CompanyWage	**Decimal**	
DriverId	**Int (Foreign key)**	This column is linked with Driver table, one-to-many relationship type.

StartDate	**DateTime**	Start of the period for which revenue was calculated.
EndDate	**DateTime**	End of the period for which revenue was calculated.

Table 9. Revenue Table

3. NoSQL database design

The NoSQL database was built using MongoDB RDBMS. A MongoDB script *create_db.js* inserts the initial documents into the collections. The parts of this script and explanation of its work and created collections are provided below.

Before creating the new database it should be ensured that there is no database with the same name:

```
conn = new Mongo();
db = conn.getDB('taxiCompany');

// drop the previous database with the same name if exists
db.dropDatabase();
```

The script created the following collections: drivers, operators, clients, bookings, regularBookings and revenue.

Drivers collection:

Name of the field	Data type	Comment
firstName	**String**	
lastName	**String**	
dateOfBirth	**Date**	
address	**Object**	This **Object** field contains fields: country (**String**), city (**String**), street (**String**) and flat (**Int**).
contactDetails	**Object**	This **Object** field contains fields: telephone (**String** because of different possible formats) and email (**String**).
dateOfEmployement	**Date**	
dateOfDismiss	**Date**	Optional field. When there is no dismiss date yet, this means that the driver is currently working in the company.
salary	**Int**	One of these 2 fields is optional. This mean that driver use the

percentageOfReceipt	**Int**	certain scheme for salary and the existence of exact field reference to a certain type of salary: fixed monthly salary or percentage of receipt basis.
shift	**Object**	This **Object** field shows which shift is used for a driver. Contains fields: startTime (**String**) and endTime (**String**). For correct comparison of those strings in queries the shift endTime should have two values for noon: 00:00 and 24:00. Therefore, the shift variants are the following: 00:00-08:00, 08:00-16:00, 16:00-24:00.
cars	**Array**	This **Array** might contain several **Objects** inside, because taxi driver can own several cars (assumption). Each object contains: registrationNumber (**String**) - unique field for each car, age (**Int**), dateOfLastMOTTest (**Date**) and carStatus (**String**, possible values: roadworthy, in for service, awaiting repair, written off).

Example:

```
db.drivers.insert({
    firstName: 'Andrey',
    lastName: 'Newman',
    dateOfBirth: new Date('1987-12-12'),
    address: {
        country: 'UK',
        city: 'London',
        street: 'Oxford street',
        flat: 12
    },
    contactDetails: {
        telephone: '+44 020 7033 3920',
        email: 'andrey.newman@gmail.com'
    },
    dateOfEmployment: new Date('2008-01-12'),
   //dateOfDismiss: new Date('2008-01-12'), // if there is no such field then the employee
is currently working
     //salary: 20000, // if there is no such field then the driver receives percentage from
each ride
      percentageOfReceipt: 40, // if there is no such field then the driver receives fixed
salary
    shift: {
       startTime: '08:00',
       endTime: '16:00'
},
    cars: [
        {
            registrationNumber : '123AJOO22MLG',
            age : 3,
            dateOfLastMOTTest: new Date('2017-07-11'),
             carStatus : 'roadworthy' //roadworthy, in for service, awaiting repair, written
off
```

```
        }
    ]
});
```

Operators collection:

Name of the field	Data type	Comment
firstName	**String**	
lastName	**String**	
dateOfBirth	**Date**	
address	**Object**	This **Object** field contains fields: country (**String**), city (**String**), street (**String**) and flat (**Int**).
contactDetails	**Object**	This **Object** field contains fields: telephone (**String** because of different possible formats of telephone input) and email (**String**).
dateOfEmployement	**Date**	
dateOfDismiss	**Date**	Optional field. When there is no dismiss date yet, this means that the operator is currently working in the company.
salary	**Int**	
shift	**String**	This field shows which shift is used for an operator. Contains fields: startTime (**String**) and endTime (**String**). For correct comparison of those strings in queries the shift endTime should have two values for noon: 00:00 and 24:00. Therefore, the shift variants are the following: 00:00-08:00, 08:00-16:00, 16:00-24:00.

Example:

```
db.operators.insert({
    firstName: 'Anny',
    lastName: 'Winehouse',
    dateOfBirth: new Date('1997-01-06'),
    address: {
        country: 'UK',
        city: 'London',
        street: 'Regent street',
        flat: 2
    },
    contactDetails: {
        telephone: '+44 020 8659 3794',
        email: 'anny1997@gmail.com'
    },
    dateOfEmployment: new Date('2016-05-22'),
    // dateOfDismiss: new Date('2008-01-12'),
```

```
    salary: 20000,
    shift: {
              startTime: '08:00',
              endTime: '16:00'
        }
});
```

Clients collection:

Name of the field	Data type	Comment
_id	**String**	This field is manually specified and it is represented by the actual telephone number.
firstName	**String**	
lastName	**String**	
clientType	**String**	This field has two possible values: “private”, “corporate”.
email	**String**	

Example:

```
db.clients.insert({
    _id: '+44 020 8659 9094',
    firstName: 'Jack',
    lastName: 'Theripper',
    clientType: 'corporate', // private
    email: 'jacktheripper@gmail.com'
});
```

Regular bookings collection:

Name of the field	Data type	Comment
clientId	**String**	Client ID is the actual telephone number of corporate client, which can be got from the Client collection.
regularType	**String**	Possible values: daily bookings or once a week
subscriptionStartDate	**Date**	Date, when this subscription for a regular bookings has been started. We need this field for understanding of all active regular bookings and performing statistical reports on a historical data.
subscriptionEndDate	**Date**	Date, when this subscription for a regular bookings has been ended. We need this field for understanding of all active regular bookings and performing statistical reports on a historical data.
departurePoint	**String**	
destinationPoint	**String**	

lastArrivalDateTime	**Date**	This field contains the last date and time when the car arrived. Date of new arrival will be calculated with regard to this field and regularType.

Example:

```
db.regularBookings.insert({
    clientId: '+44 020 8659 3794',
    regularType: 'once a week', // daily bookings
    subscriptionStartDate: new Date('2017-09-30'),
    subscriptionEndDate: new Date('2018-09-30'),
    departurePoint: 'Heathrow Airport Terminal 3',
    destinationPoint: 'The Royal London Hospital',
    lastArrivalDateTime: new Date('2017-11-25T15:30:00Z')
});
```

Bookings collection:

Name of the field	Data type	Comment
driver	**Object**	**Object** field consists of several fields, including: driverId (MongoDB **ObjectId**, generated for each driver automatically in Driver collection), salary (**Int,** from the Driver collection), percentageOfReceipt (**Int,** from the Driver collection), carRegistrationNumber (**String**, from the Driver collection, from a certain **Object**). Only the most important driver fields are included in this embedded document.
operatorId	**ObjectId**	MongoDB **ObjectId**, generated for each operator automatically in Operator collection.
clientId	**String**	Client ID is manually generated and it is represented by the actual telephone number, which can be got from the Client collection.
dateTime	**Date**	
departurePoint	**String**	
destinationPoint	**String**	
regularBookingId	**ObjectId**	MongoDB **ObjectId**, generated for each regular booking automatically in regularBookings collection. This field is optional. If it is absent, this means that this exact booking is not regular.
amount	**Int**	The cost of booking.
dateOfPayment	**Date**	This field is optional. If it is absent, this means that this exact booking is not paid yet.

For creating booking collection, *create_db.js* retrieves information about driver, operator, client and regular booking to use appropriate information from those documents in the *db.bookings.insert* operation.

```
var driver = db.drivers.findOne();
var operator = db.operators.findOne();
var client = db.clients.findOne();
var regularBooking = db.regularBookings.findOne();
```

Example:

```
db.bookings.insert({
    driver: {
                driverId: driver._id,
                // salary: driver.salary, // presense of the field depends on the driver
rate type
                percentageOfReceipt: driver.percentageOfReceipt, // presense of the field
depends on the driver rate type
                carRegistrationNumber: driver.cars[0].registrationNumber
        },
    operatorId: operator._id,
    clientId: client._id,
    dateTime: new Date('2017-09-30T15:30:00Z'),
    departurePoint: regularBooking.departurePoint,
    destinationPoint: regularBooking.destinationPoint,
    regularBookingId: regularBooking._id, // optional field. If do not have, that is not
regular booking
    amount: 120,
    dateOfPayment: new Date('2017-09-30'), // if there is no dateOfPayment then it has not
been paid yet
});
```

Revenue collection:

Name of the field	Data type	Comment
driverId	ObjectId	MongoDB **ObjectId**, generated for each driver automatically in Driver collection.
startDate	Date	Two filed for the period of revenue calculation.
endDate	Date	
driverWage	Int	
companyWage	Int	

Example:

```
db.revenue.insert({
    driverId: driver._id,
    startDate: new Date('2017-09-01'),
    endDate: new Date('2017-09-30'),
    driverWage: 48,
```

```
    companyWage: 72
});
```

Script for adding more data into the MongoDB database can be found in attached files (*initialize_db.js*).

4. Queries

The following queries were executed to prove that the database has appropriate design for typical use cases of taxi company.

1) Operator need to find an available driver for a ride, taking into account driver's shift. There is also a need to check that a driver has an available car. This queries allow to do the mentioned functionality.

```
// find available drivers for a ride at specified time (11:15 in example)
db.drivers.find({
        'shift.startTime': {$lte: '16:15'},
        'shift.endTime': {$gte: '16:15'}}
);

// find available drivers for a ride at the current time
db.drivers.find({
        'shift.startTime': {$lte: new Date().getHours() + ':' + new Date().getMinutes()},
        'shift.endTime': {$gte: new Date().getHours() + ':' + new Date().getMinutes()}
});

// the specified or current time should be within drivers' shifts and drivers should have
available cars
db.drivers.find({
        'shift.startTime': {$lte: '16:15'},
        'shift.endTime': {$gte: '16:15'},
        'cars': {$elemMatch: { 'carStatus': 'roadworthy'}}
});
```

2) Company assigns rides every day. There is a need to find regular bookings for today.

```
// find regular bookings which may happen today.
db.regularBookings.find({
    'subscriptionStartDate': { $lte: new Date(new Date().getFullYear(), new
Date().getMonth(), new Date().getDate())},
    'subscriptionEndDate': { $gte: new Date(new Date().getFullYear(), new
Date().getMonth(), new Date().getDate(), 23, 59, 59)}
});

// find regular bookings which may happen in the specified day
// regularType is unstructured field which makes it hard to anaylise only with query. We
assume that it will be analysed by some program or operator.
db.regularBookings.find({
    'subscriptionStartDate': { $lte: ISODate("2017-11-20T00:00:00.000Z")},
```

```
    'subscriptionEndDate': { $gte: ISODate("2017-12-11T00:00:00.000Z")}
});
```

3) MOT car test should be done every year. Company needs to manage its resources and to know one month in advance which cars will have to do this this test soon. Query helps to recognise whose drivers will soon need MOT test for their cars.

```
// we assume that MOT test is needed soon if the last was at least 330 days ago
db.drivers.find({
    'cars.dateOfLastMOTTest': {"$lte": new Date(new Date().setDate(new Date().getDate() -
330))}
 });

// The second variant. Display only driver Ids and cars for this query (remove distracting
driver's information which is not important in this case).
 db.drivers.find({
    'cars.dateOfLastMOTTest': {"$lte": new Date(new Date().setDate(new Date().getDate() -
330))}
 }, {"_id": 1, "cars": 1});
```

4) Debt collections is an important financial activity of every company. This query allows the company to find all unpaid bookings and count the total amount of clients debts.

```
// how many unpaid rides(bookings) we have?
db.bookings.find({
    'dateOfPayment': { $exists: false }
}).count();

// get all unpaid rides(bookings)
db.bookings.find({
    'dateOfPayment': { $exists: false }
});

// how much money should we receive from unpaid bookings?
db.bookings.aggregate(
    {
        $match: {'dateOfPayment': { $exists: false }}
    },
    {
        $group: { _id : 'totalUnpaidAmount', sum : { $sum: "$amount" }}
    }
);
```

5) Marketing strategy always need some descriptive statistics (for loyalty programs for instance). For this purposes company need to know its customers, divide them into segments and manage key performance indicators. This query allows to find the most expensive bookings in some period.

```
// Find the most expensive bookings
// if the price is the same, latest bookings should be higher in the output result
// the first way
db.bookings.aggregate([
```

```
        { $sort : {amount  : -1, dateOfPayment: -1} },
    { $limit: 10 }
]);

// the second way
db.bookings.find()
     .sort({amount  : -1, dateOfPayment: -1} )
     .limit(10);
```

6) Understanding of daily amount of bookings is useful as part of descriptive statistics for managing business.

```
// how many bookings we have in the specified time interval
db.bookings.find({
    'dateTime': {
        $gte: ISODate("2017-11-01T00:00:00.000Z"),
        $lte: ISODate("2017-11-01T23:59:59.000Z")
        }
}).count();
```

7) Understanding the amount of money which the company pays for all drivers the specified period (e.g. last month) might be one of KPIs to evaluate their success at reaching business targets.

```
// Total drivers' revenue for specified month
db.revenue.aggregate([
    {
        $match: {
            'startDate': { $gte: ISODate("2017-11-01T00:00:00.000Z") },
            'endDate': { $lte: ISODate("2017-11-30T00:00:00.000Z") }
        }
    },
    {
        $group: {
            '_id': {
                'startDate': "$startDate",
                'endDate': '$endDate'
            },
            'totalDriversMonthRevenue': { $sum: '$driverWage' },
        }
    }
]);
```

8) Another KPI - the total amount of earned money.

```
// Total company revenue for the whole period of taxi company existence
db.revenue.aggregate([
    {
        $group: {
            '_id': null,
            'totalCompanyRevenue': { $sum: '$companyWage' },
        }
```

```
    }
]);
```

9) It is also important to identify clients in terms of their profitability for the company.

```
// find the most profitable client
db.bookings.aggregate(
   [
     {
       $group:
         {
           _id: {clientId: '$clientId'},
           totalProfit: { $sum: '$amount' },
         }
     },
     { $sort: {'totalProfit': -1} },
     { $limit: 1 }
   ]
);
```

10) For understanding of drivers performance, the amount of money the specified driver earned for the company by performing rides in the specified dates range is calculated.

```
// drivers id and dates period should be specified by user
// in example we search for driver 0
var driver0 = db.drivers.findOne();
db.bookings.aggregate([
    {
        $match: {
            'driver.driverId': driver0._id,
            'dateTime': {
                $gte: ISODate("2017-11-01T00:00:00.000Z"),
                $lte: ISODate("2017-11-30T23:59:59.000Z")
            }
        }
    },
    {
        $group: {
            '_id': { driverId: '$driver.driverId' },
            'amountOfDriverBookings': { $sum: '$amount' }
        }
    }
]);
```

11) This query allows to know more information about the ride: select an operator who allocated booking and the driver who performed that booking.

```
// Find driver and operator related to booking
// For example, looking for booking from Private drive 10 which was on the 1st Septemper
2017 at 08:30
```

```
var selectedBooking = db.bookings.findOne({
    'departurePoint': 'Private drive 10',
    'dateTime': ISODate("2017-10-02T08:30:00")
 });

 db.operators.find({
    '_id': selectedBooking.operatorId,
 });

 db.drivers.find({
    '_id': selectedBooking.driver.driverId,
 });
```

12) Implement paging system for all bookings.

```
// usage of skip
var pageNumber = 1;
var limitPerPage = 5;
db.bookings.find().skip(limitPerPage * (pageNumber - 1) ).limit(limitPerPage);
```

13) Sometimes the data should be updated. For instance, employee was fired or all drivers passed medical examination.

```
// Update example if we want to fire driver 8
var driver8 = db.drivers.find({})[8];

db.drivers.update({_id: driver8._id}, {$set: {dateOfDismiss: new Date()}});

// update all drivers as they passed medical examination
var dateOfMedicalExamination = new Date();
db.drivers.update({},
    {$set: {dateOfMedicalExamination: dateOfMedicalExamination}},
    {multi: true}
);
```

5. Usage of performance monitoring tools

The bookings collection is assumed to be one of the most frequently used in the current design. After each order in the taxi company a new document is added into the collection. Moreover, for calculating revenue the bookings collection is scanned. Bookings collection could also be used for retrieving a lot of statistical information such as the most profitable clients, the most popular times of bookings, the most effective drivers. Therefore, usage of performance tools are shown for this collection.

Evaluate the performance of a query

The following query retrieves documents where the dateTime field belongs to the day of the 1st of November 2017:

```
db.bookings.find({"dateTime": {
```

```
    $gte: ISODate("2017-11-01T00:00:00.000Z"),
    $lte: ISODate("2017-11-01T23:59:59.000Z")
}})
```

The query returns three documents. Method *explain()* with parameter *executionStats* runs the query optimiser to choose the winning query plan, executes the winning plan and returns statistics describing the execution of the winning plan. (1) Usage of the *explain("executionStats")* method:

```
db.bookings.find({"dateTime": {
    $gte: ISODate("2017-11-01T00:00:00.000Z"),
    $lte: ISODate("2017-11-01T23:59:59.000Z")
}}).explain("executionStats")
```

For this query with no index it returns the following results:

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        ...
        "winningPlan" : {
            "stage" : "COLLSCAN",
            ...
        },
        "rejectedPlans" : []
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 3,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 29,
        "executionStages" : {
            "stage" : "COLLSCAN",
            ...
            },
            ...
        }
    },
    ...
}
```

It can be seen that:

- *queryPlanner.winningPlan.stage* displays COLLSCAN to indicate a collection scan (it means that no index was used).
- *executionStats.nReturned* displays 3 to indicate that the query matches and returns three documents.
- *executionStats.totalDocsExamined* display 29 to indicate that MongoDB had to scan 29 documents (all documents in the collection) to find the three matching documents.

The difference between the number of matching documents and the number of examined documents may suggest that the query might benefit from the use of an index. (1)

Therefore, an index on the *dateTime* field is added:

```
db.bookings.createIndex( { dateTime: 1 } )
```

Now the *explain()* method returns the following results:

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        ...
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "dateTime" : 1.0
                },
                "indexName" : "dateTime_1",
                ...
            }
        },
        "rejectedPlans" : []
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 3,
        "executionTimeMillis" : 1,
        "totalKeysExamined" : 3,
        "totalDocsExamined" : 3,
        "executionStages" : {
            "stage" : "FETCH",
            ...
            "inputStage" : {
                "stage" : "IXSCAN",
                ...
            }
        }
    },
    ...
}
```

From new *explain()* results it can be noticed that:

- queryPlanner.winningPlan.inputStage.stage displays IXSCAN to indicate that index was used.
- executionStats.nReturned displays 3 to indicate that the query matches and returns three documents.
- executionStats.totalKeysExamined display 3 to indicate that MongoDB scanned three index entries.
- executionStats.totalDocsExamined display 3 to indicate that MongoDB scanned three documents.

Also *explain()* method returns total time in milliseconds required for query plan selection and query execution, but in the case of very small collections it is not useful.

After index creation, the query scanned 3 index entries and 3 documents to return 3 matching documents. Without the index, the query had to scan the whole collection (29 documents) to return 3 matching documents. Therefore, it is proved that efficiency is improved by using index in this case.

To provide more informative results, additional one thousand randomly generated bookings were added into Bookings collection. After that find query was run with and without index. The code which adds random bookings (*addBookings.js*) and results of *explain()* method are provided below:

```
function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

var drivers = db.drivers.find();
var operators = db.operators.find();
var clients = db.clients.find();

var randBookings = [];

for (var i=0; i < 2000; i++) {
    var date = new Date(2017, getRandomInt(8, 11), getRandomInt(1, 31), getRandomInt(0,
23), getRandomInt(0, 59));
    var randDriver = drivers[getRandomInt(0, drivers.length()-1)];
    var randOperator = operators[getRandomInt(0, operators.length()-1)];
    var randClient = clients[getRandomInt(0, clients.length()-1)];
    randBookings.push({
        'driver': {
            'driverId': randDriver._id,
            'percentageOfReceipt': randDriver.percentageOfReceipt,
            'salary': randDriver.salary,
            'carRegistrationNumber': randDriver.cars[0].registrationNumber
        },
        'operatorId': randOperator._id,
        'clientId': randClient._id,
        'dateTime': date,
        'departurePoint': 'Departure ' + i,
        'destinationPoint': 'Destination ' + i,
        'amount': getRandomInt(20, 150),
        'dateOfPayment': date,
        'testBooking': true
    });
}

db.bookings.insertMany(randBookings);
```

Query with no index:

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        ...
        "winningPlan" : {
            "stage" : "COLLSCAN",
            ...
        },
        "rejectedPlans" : []
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 15,
        "executionTimeMillis" : 1,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 2029,
        "executionStages" : {
            "stage" : "COLLSCAN",
            ...
        }
    },
    ...
}
```

Query with index:

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        ...
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "dateTime" : 1.0
                },
                "indexName" : "dateTime_1",
                ...
                }
            }
        },
        "rejectedPlans" : []
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 15,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 15,
        "totalDocsExamined" : 15,
        "executionStages" : {
            "stage" : "FETCH",
            ...
            "inputStage" : {
                "stage" : "IXSCAN",
```

```
                ...
            }
        }
    },
    ...
}
```

For a larger collection the difference between matched and examined documents in the case of query with and without index is much more significant. It returns 15 matching documents query with no index scanned 2029 documents of the collection, whereas with index it was only 15.

In general, indexes provide significant performance gains for read operations, but the cost is the penalty on write operations. So it is necessary to be careful when creating new indexes and evaluating the existing indexes to ensure that queries actually use these indexes. (2) Evaluation can be done by using the same *explain()* method. The *explain()* method results of update operation with and without index on bookings collection are the following:

```
db.bookings.explain("executionStats").update(
    { "dateTime": ISODate("2017-10-05T23:02:00.000Z") },
    { $set: { "dateTime" : ISODate("2017-10-01T00:00:00.000Z") }}
)
```

Query with no index:

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        ...
        "winningPlan" : {
            "stage" : "UPDATE",
            "inputStage" : {
                "stage" : "COLLSCAN",
                ...
            }
        },
        "rejectedPlans" : []
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 0,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 405,
        "executionStages" : {
            "stage" : "UPDATE",
            ...
            "inputStage" : {
                "stage" : "COLLSCAN",
                ...
            }
        }
    },
```

```
    ...
}
```

Query with index:

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        ...
        "winningPlan" : {
            "stage" : "UPDATE",
            "inputStage" : {
                "stage" : "FETCH",
                "inputStage" : {
                    "stage" : "IXSCAN",
                    "keyPattern" : {
                        "dateTime" : 1.0
                    },
                    "indexName" : "dateTime_1",
                    ...
                }
            }
        },
        "rejectedPlans" : []
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 0,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 1,
        "totalDocsExamined" : 1,
        "executionStages" : {
            "stage" : "UPDATE",
            ...
            "inputStage" : {
                "stage" : "FETCH",
                ...
                "inputStage" : {
                    "stage" : "IXSCAN",
                    ...
                }
            }
        }
    },
    ...
}
```

It can be seen that in the case of using index update query includes in addition FETCH and IXSCAN stages.

Ensure that Queries use indexes

To ensure that query use index *explain()* method might be used. Example of usage compound index on date time and driver in the bookings collection is shown below:

db.bookings.createIndex({'dateTime': 1, 'driver': 1})

Find query with selection filter only by *dateTime*:

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        ...
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "dateTime" : 1.0,
                    "driver" : 1.0
                },
                "indexName" : "dateTime_1_driver_1",
                ...
            }
        },
        "rejectedPlans" : []
    },
    ...
}
```

Find query with selection filter by *dateTime* and *driver.driverId*:

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        ...
        },
        "winningPlan" : {
            "stage" : "FETCH",
            ...
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "dateTime" : 1.0,
                    "driver" : 1.0
                },
                "indexName" : "dateTime_1_driver_1",
                ...
            }
        },
        "rejectedPlans" : []
    },
    ...
}
```

Find query with selection filter only by *driver.driverId*:

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
```

```
        ...
        "winningPlan" : {
            "stage" : "COLLSCAN",
            ...
        },
        "rejectedPlans" : []
    },
    ...
}
```

Taking into account the *explain()* results, it can be seen that index was not used for query which filtered only by *driverId*.

Database Profiler

The database profiler is a profiling system that can help identify inefficient queries and operations. (3) It collects fine grained data about MongoDB write operations, cursors, database commands on a running mongod instance. (4) The database profiler writes data in the system.profile collection. To view the profiler's output normal MongoDB queries are used. (4)

The profiler can be set with different profiling levels:

- 0 - no profiling;
- 1 - includes only "slow" operations, by default "slow" are operations longer than 100 milliseconds;
- 2 - includes all operations.

Example of usage:
Set the profiler with level 2. Then run find query on booking collection. After that examine system.profiler collection.

```
db.setProfilingLevel(2)

db.getCollection('bookings').find({})

db.getCollection('system.profile').find({})
```

system.profiler collection contains information about executed query:

```
{
    "op" : "query",
    "ns" : "taxiCompany.bookings",
    "query" : {
        "find" : "bookings",
        "filter" : {}
    },
    "cursorid" : 169549956109.0,
    "keysExamined" : 0,
    "docsExamined" : 101,
    "keyUpdates" : 0,
    ...
    },
```

```
    "nreturned" : 101,
    "responseLength" : 33024,
    "protocol" : "op_command",
    "millis" : 0,
    "execStats" : {
        "stage" : "COLLSCAN",
        ...
    },
    "ts" : ISODate("2017-12-14T01:48:01.433Z"),
    ...
}
```

By setting the profiler to level 1, only long inefficient queries can be identified. After that, those queries may be investigated using *explain()* method which was described earlier and optimised. In this coursework the database does not contain a lot of documents inside collection, so all queries run for 0-1 millisecond and usage of profiler will not show any slow queries.

References

1. MongoDB, Inc.. db.collection.explain() [Internet]. 2017 [14 December]. Available from: https://docs.mongodb.com/v3.4/reference/method/db.collection.explain/#db.collection.explain
2. MongoDB, Inc.. Write Operation Performance [Internet]. 2017 [14 December]. Available from: https://docs.mongodb.com/manual/core/write-performance/
3. MongoDB, Inc.. Database Profiling [Internet]. 2017 [14 December]. Available from: https://docs.mongodb.com/manual/administration/analyzing-mongodb-performance/#database-profiling
4. MongoDB, Inc.. Database Profiler [Internet]. 2017 [14 December]. Available from: https://docs.mongodb.com/manual/tutorial/manage-the-database-profiler/