IMSE Milestone 3 - Documentation

Eylül Gökce Harputluoglu 11728483 & Roman Schneglberger 01127050

Reasons for designing the NoSQL database

What we know that a relational database is not always the best choice to use in a system. Our system is based on renting cars, which means every day we gain more costumers, more cars and with all this, more rents which includes bills and insurances for each of them. Therefore, we need more space to store all this data. What we want is that a larger system which can analyze the customer activity and tracks our cars rental (reservation) status. We should be able to see these statically (e.g. frequently preferred locations of costumers) and adapt the new costumers/cars in the system. Generally, as a rental company, we want to be prepared for a big amount of data and dare to switch the system as early as possible.

Why to use NoSQL documents and Collections

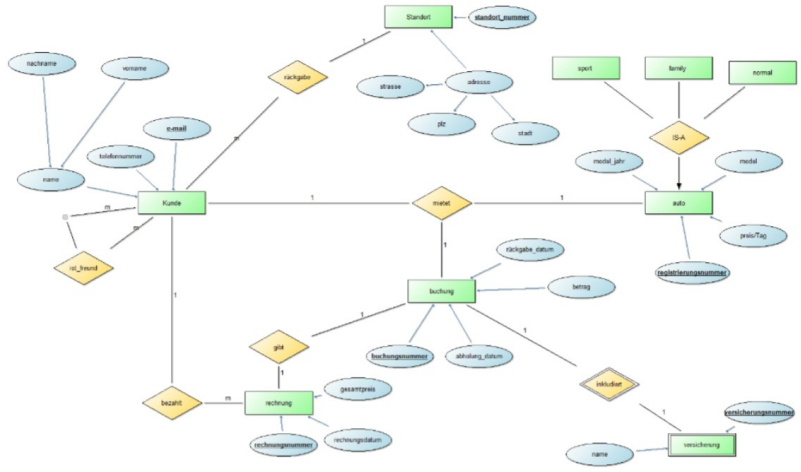
The document is the unit of storing data in a MongoDB database and it uses JSON format for storing data, which is a lightweight, thoroughly explorable format used to interchange data between various applications. Documents are analogous to the records of an RDBMS. We can show the difference as following:

RDBMS	MongoDB
Table	Collection
Column	Key
Value	Value
Records/Rows	Document/Object

Insert, update, and delete operations can be performed on a collection**.**
A collection may store several documents. A collection is analogous to a table of an RDBMS and it may store documents those who are not same in structure. This is possible because MongoDB is a Schema-free database (We describe schema as the structure of data in RDMS). In the other hand MongoDB doesn’t require such a set of formula defining structure of data.

How we designed our system

Before we created our RDMS we designed our renting database as you can see in the ER-Diagram below:



From this diagram, we decided the relations for our NoSQL database as following:

(Booking == Reservation)

- Booking <-> Insurance: One-To-One
- Booking <-> Car: One-To-Many (few)
- Booking <–> Bill: One-To-One
- Customer <–> Costumer (AreFriends): Many-To-Many
- Customer <-> Location: One-To-Many
- Costumer <–> Bill: One-To-Many (actually many)

And we decided to create 4 collections in our NoSQL database. Our collections are *Costumer collection, Reservation collection, Car collection and Location collection*.

In Costumer and Reservation collections, we created sub documents with using Join operations in SQL at the migration stage to have a more structured design.

In the Costumer collection, we find the related bill information for the specific costumer and we create costumer sub document to determine the discounts for this costumer (see Costumer<->Costumer relation below), a costumer gets an discount if he brings a new costumer. Therefore we store these costumers as friends of our existing costumer. And we decided to store the location information also in Costumer collection to find out the statistically more preferred locations from costumers (see Costumer<-> Location relation).

E.g.

Costumer {
"EMAIL": "Chester_Benfield5546@twace.org",
"FIRSTNAME": "Aggi",
"SURNAME": "GARCIA",
"PHONE_NUMBER": "511997205",
"LOCATIONID": "28",
"BILLING": [{
"BILLING_NUMBER": "50",
"TOTAL_PRICE": "453",
"BILLDATE": "2020-06-18",
"C_EMAIL": "Chester_Benfield5546@twace.org"}],
"LOCATIONS": [{
"LOCATION_ID": "28",
"ZIP_CODE": "1210",
"STREET": "c4CMvjrBX3lDIq",
"CITY": "Vienna"}],
"FRIENDS": [{
"FRIENDS": "Sage_Wise4536@ubusive.com"}]
}

In the Reservation collection, we added all related information about the rent, so that we can find the all information about one reservation/bill/car faster. The bill information is stored both in reservation and costumer, which is good to find the connection of the rent. For example, after a successfully reservation, we want to calculate, if the total price (it is written on the bill) is higher than average, or we can see the average days which a car (car information) has been rented (reservation information).

E.g. (here we see that this reservation has no data about car → it is because we generate the data randomly and not every time it is logically correct output, you can check out our foreign key constraints)

Reservation {
"RESERVATION_NUMBER": "100",
"FROM_DATE": "2020-06-18",
"RETURN_DATE": "2020-06-18",
"AMOUNT": "187",
"BILLING": {
"BILLING_NUMBER": "52",
"TOTAL_PRICE": "700",
"BILLDATE": "2020-06-18",
"C_EMAIL": "Miriam_Newman8501@supunk.biz"}
}

The other 2 collections (car and location) is created to insert the new car/loc into the system. Because after import we only have existing data.

Booking <-> Insurance: One-To-One

An insurance is depending on a booking, which means if a booking exists that a unique insurance is creating for the specific booking. If there are 0 bookings in the DB then we cannot show an insurance. The collection “Booking” should include insurance. Therefore, we decided the correct relation would be One-To-One.

Costumer <-> Bill: One-To-Actually Many

We know that one costumer can rent many cars separately. For this reason, it would be a good way if we store the bills in the Costumers. With this relation we will be able to see all bills generated for that costumer. But we should also consider that this kind of a big data need more space in the disk and cost much more time if we update/search it.

Booking <-> Car: One-To-A-Few

With a booking, it is possible to rent a few cars. We decided to use one-two-a-few relation in order to determine which cars are rented with a specific bill. For that, we should store carIDs as a list (Array) in Booking. A little issue here would be a little more storage place in the disk.

Costumer <-> Costumer: Many-To-Many

Costumers can be friends with each other. In order to extend our business, we want to use these friendships to gain more costumers and we want to offer our already existed costumer 10% discount for every costumer they bring. Therefore, we should remember this relation in the system and we decided to use here a documentation as an array of the costumers to find out which costumer is friends with each other.

Costumer <-> Location: One-To-Many

Costumers locate the car which they rented in different locations. In our systems there are already various locations and we want to store this information in a separate collection as well as in the costumer collection in order to calculate the ratio of a costumer, that shows us their favorite locations or their preferred locations to hand over the car. With this, we can analyze the most preferred locations and we will be able to show these statically in the system.

Booking <-> Bill: One-To-One

After a costumer booked a car/cars, it generates a bill automatically. For each booking, it created a bill, which includes the total price. Thus, our choice is to use one-to-one relation in this case. And we use bill info in Costumer as well as in Reservation collections.

Inherited Relations (Sport-, Normal-, Family car)

We made our decisions that we store the cars attributes in each inherited entity under the name of “Cars”.

Why storing and not referencing

We are aware of that storing needs more space in the memory and a reference may be faster in some cases but we still small company and at this time we can store the data and because there is not a lot of data in the system, we can easily and faster access to the data we want.

Our Sharding Strategy: Hashed Sharding

As we mentioned before, we want to expand our company and with using NoSQL Database, Sharding became even more important. The advantages of this strategies are followings:

- Monotonically Changing Values
- High Shard Key Frequency which supports compound keys
- Chunk size can be configured via Hash function

With this strategy, we have to use a hast function and every collection is stored with a key in the system, which makes it possible to search or update a document really fast.

How to index the data in the MongoDB: Multi Key Index

Our choice for our MongoDB design is the Multi Key Index. It is because we are using arrays in our collections and to access an item in the list, we must use an additional key for list index. We know that These multikey indexes support efficient queries against array fields. Multikey indexes can be constructed over arrays that hold both scalar values and nested documents. MongoDB automatically creates a multikey index if any indexed field is an array therefore, we did not need to explicitly specify the multikey type. Besides MongoDB keeps track of which indexed field or fields cause an index to be a multikey index. Tracking this information allows the MongoDB query engine to use tighter index bounds.

Time Report

Romans Schneglberger

Setting up Ubuntu up because Docker wont run on Fedora	2d
Docker compose file	4h
setting up apache-php with mongo driver	4h
setting up mongo db container	1h
Writing mongoDb migration app	5h + 2h (alone)
testing	2h
sql db queries	1h

creating jar file out of mongo app	1h

Eylül Gökce Harputluoglu

Documentation	6h
Presentation slides + work protocol	1h
Data migration (together)	5h+2h(alone)
java structure	1h
list conversion for MongoDB collections	1h