BEERZONE DOCUMENTATION

Large-Scale and Multi-Structured Databases
2021-2022

Application developed by

Federico Montini, Emanuele Tinghi, Veronica Torraca

Summary

Abstract

The application provides a service named “BeerZone”, in which a registered user can for example browse for beers, look for recipes, add new beer and review them.

This application was developed using Java and Java Swing.

Evaluating the needs of our project, we chose to use Neo4j and MongoDB as non-relational databases.

Neo4J was used to leverage data in a way that would provide users with suggestions based on the user's tastes, namely it was used to develop the social part, taking advantage of its main feature of memorizing relationships between entities.

MongoDB on the other hand was used as main storage, thanks to its sharding capability, and to provide various features such as computing Beers or Brewery statistics.

The whole project, including documentation and code can be found inside the GitHub repository.

Introduction

BeerZone is a Java application that collects data and information mostly about beers but also on users and breweries.

As just mentioned, the application has two types of users:

- A Standard User
- A Brewery User

The two types of users have both common and type-specific functionalities, for example:

- A Brewery User can add, remove or edit a beer (a Standard User can't)
- A standard user can add beers to favorites or review them (a Brewery can't)
- Both can browse beers, see their recipes and score.

Another additional feature is to request and view statistics; each type of user can extract specific statistics that will meet user's interest, like Most Favorite Beers or Top Scored Beers of the last month.

Design

This section describes the main actors and the Functional and Non-Functional requirements of the application.

- Functional Requirements

The Unregistered User can just sign up and become a Standard User or a Brewery User.

A Standard User can

- Sign In the app
- Logout from the app
- Browse beers by name and style
- Browse breweries by name
- See a beer recipe
- Review a beer or delete his review on that specific beer
- Add and remove a beer to its favorites
- See some statistics like: Most Favorite Beers, Most Reviewed beers and Top Scoring beers. Every rank is computed for the last month or year starting from the current date
- Delete or modify his account
- View suggestions based on its tastes

A Brewery User can

- Sign In tthe app
- Logout from the app
- Browse beers by name and style
- Browse breweries by name
- See a beer recipe
- Modify its own beers
- Add/Delete its own beers
- Delete or modify his account
- See some statistics like: Brewery score or Beer score for a specific feature

- Non-Functional Requirements

- The application must be simple and intuitive
- The user email must be unique
- The username for a Standard User must be unique
- Good quality of service must be provided. So, given the fact that this application is a kind of social network, we chose to focus on the A-P side of the CAP triangle: we want to provide good availability and partition tolerance. As far as consistency is concerned, the eventual consistency paradigm will be exploited
- Data must be saved and maintained in order to avoid having beers without a corresponding brewery or favorites related to nobody

- Dataset

The Beer dataset has been taken from two different sources:

- Beer Recipes
- Beers, Reviews & Breweries

The users have been taken from:

- Users

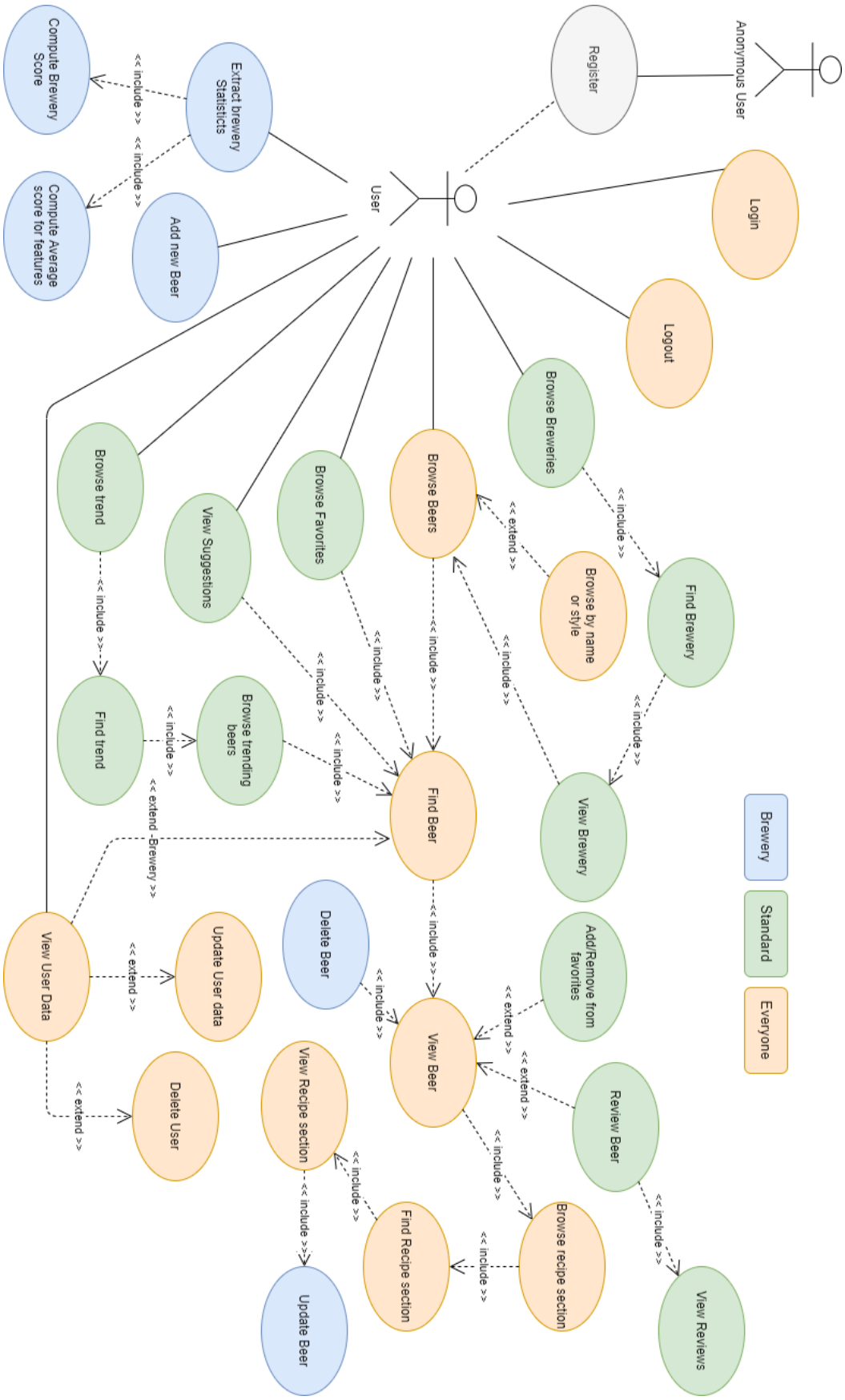
Favorited beers relationships were randomly generated with a random number generator through Python; it chose an existing id among the various users and assigned to it the id of a randomly chosen beer.

It is important to note that the datasets needed a Pre-Processing phase, as the various fields between the two datasets of beers did not match. It was therefore necessary to standardize and make them consistent. Major changes made to the dataset are described below.

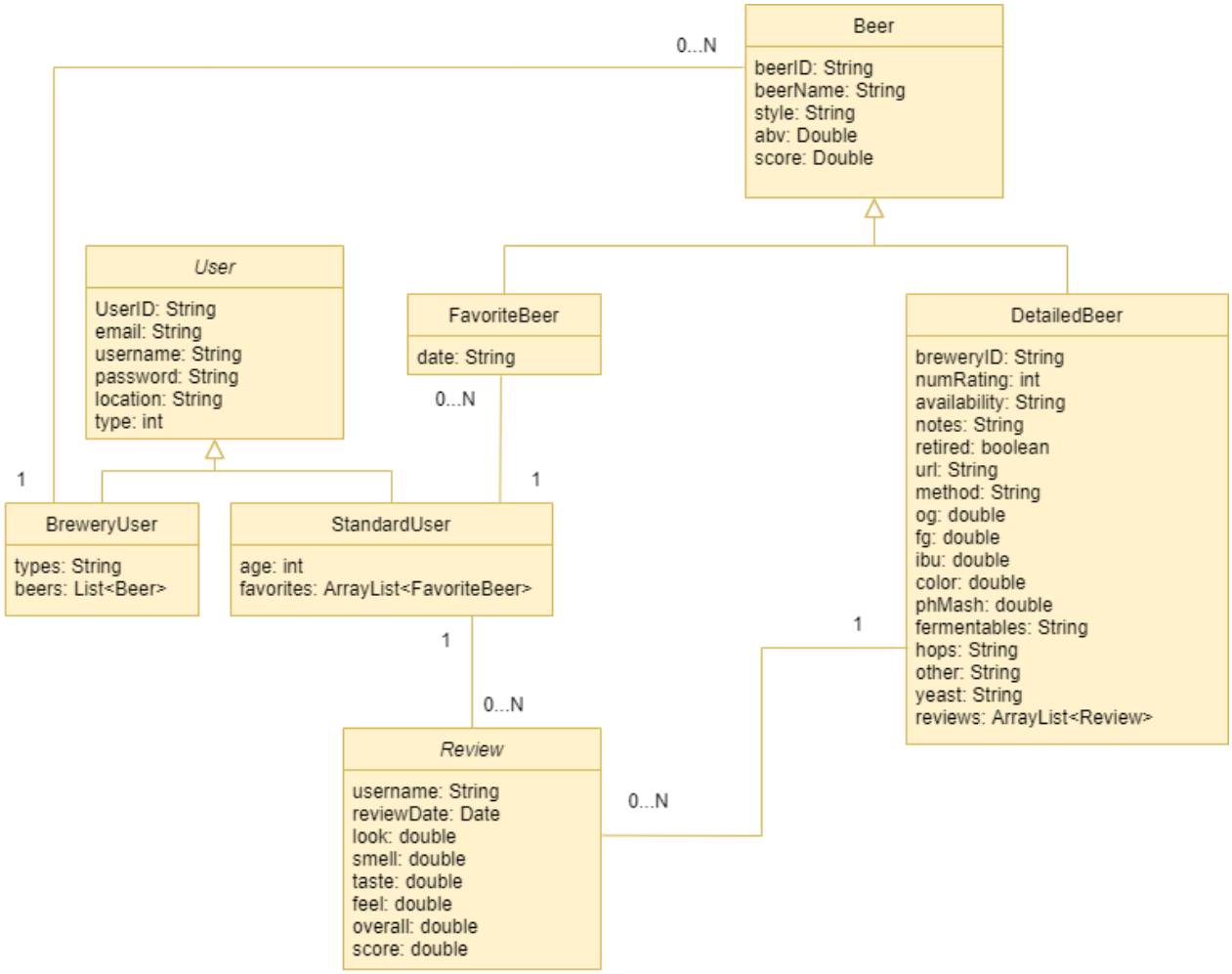
- Missing fields have been added with meaningful values assigned to them, like a brewery associated with a beer or a list of the beers (id – name) produced by each brewery

- Username have been assigned to standard-users, since the initial user dataset did not have a username field; the usernames in the reviews were then randomly associated with standard-users
- Since reviews and beers were in two different files, it was necessary to merge the two datasets creating a single collection of beers in which the reviews were also stored as a list of nested documents

- Use Case


Anonymous User
Register
User
Login
Logout
Add new Beer
Compute Brewery Score
Extract brewery Statistics
Compute Average score for features
<< include >>
<< include >>
Browse Breweries
Browse Beers
Browse Favorites
View Suggestions
Browse trend
Find Brewery
Browse by name or style
<< include >>
<< extend >>
Find trend
Browse trending beers
Find Beer
View Brewery
Add/Remove from favorites
View User Data
Update User data
Delete User
Delete Beer
View Beer
Review Beer
View Recipe section
Find Recipe section
Browse recipe section
View Reviews
Update Beer
Brewery
Standard
Everyone

Class Analysis



- Each StandardUser can add to favorites N beers
- Each Brewery can produce N beers
- Each Beer can be crafted by only one Brewery
- Each StandardUser can review N beers and each beer can be reviewed N times (by different StandardUsers)

- GeneralUser Attributes

Name of the Attribute	Attribute Type	Description
UserID	String	Unique string that identifies a User inside the collection
email	String	User email

username	String	Unique StandardUser identifier or Brewery name
password	String	User's password
location	String	User's location
type	int	User type: 0 -> S.U. 1-> Brewery

- Brewery Attributes

Name of the Attribute	Attribute Type	Description
All the same of the GeneralUser		
types	String	Brewery type
beers	List<Beer>	List of beers produced by the brewery

- StandardUser Attributes

Name of the Attribute	Attribute Type	Description
All the same of the GeneralUser		
age	int	User's age
favorites	ArrayList<FavoriteBeer>	List of beers that has been liked by the user

- Beer Attributes

Name of the Attribute	Attribute Type	Description
BeerID	String	Unique string that identifies a beer inside the collection
beerName	String	Name of the beer
style	String	Beer's style
abv	double	alcohol by volume of the Beer
score	double	Beer's review score

- FavoriteBeer Attributes

Name of the Attribute	Attribute Type	Description
All the same of the Beer		
date	String	Date in which the user put the beer in his favorites

- DetailedBeer Attributes

Name of the Attribute	Attribute Type	Description
All the same of the Beer		
BreweryID	String	ID of the brewery that produces the beer
numRating	int	Number of times the beer has been reviewed
availability	String	Beer availability periods
notes	String	Various annotations on the beer
retired	Boolean	Indicates if a beer is retired from the market
url	String	Link to a web-page that shows all beer's characteristics
method	String	Production method of a beer
og	double	Is the original gravity of the beer
fg	double	Is the final gravity of the beer
ibu	double	Is the International Bitterness Unit
color	double	Indicate the tendency of the beer to be darker
phMash	double	Is the measure of how the beer is acidic or alkaline

fermentables	String	Indicate which fermentables are used
hops	String	Indicate which hops are used
other	String	Others recipe informations
yeast	String	Indicate which yeast are used
reviews	ArrayList<Review>	List of reviews for the beer

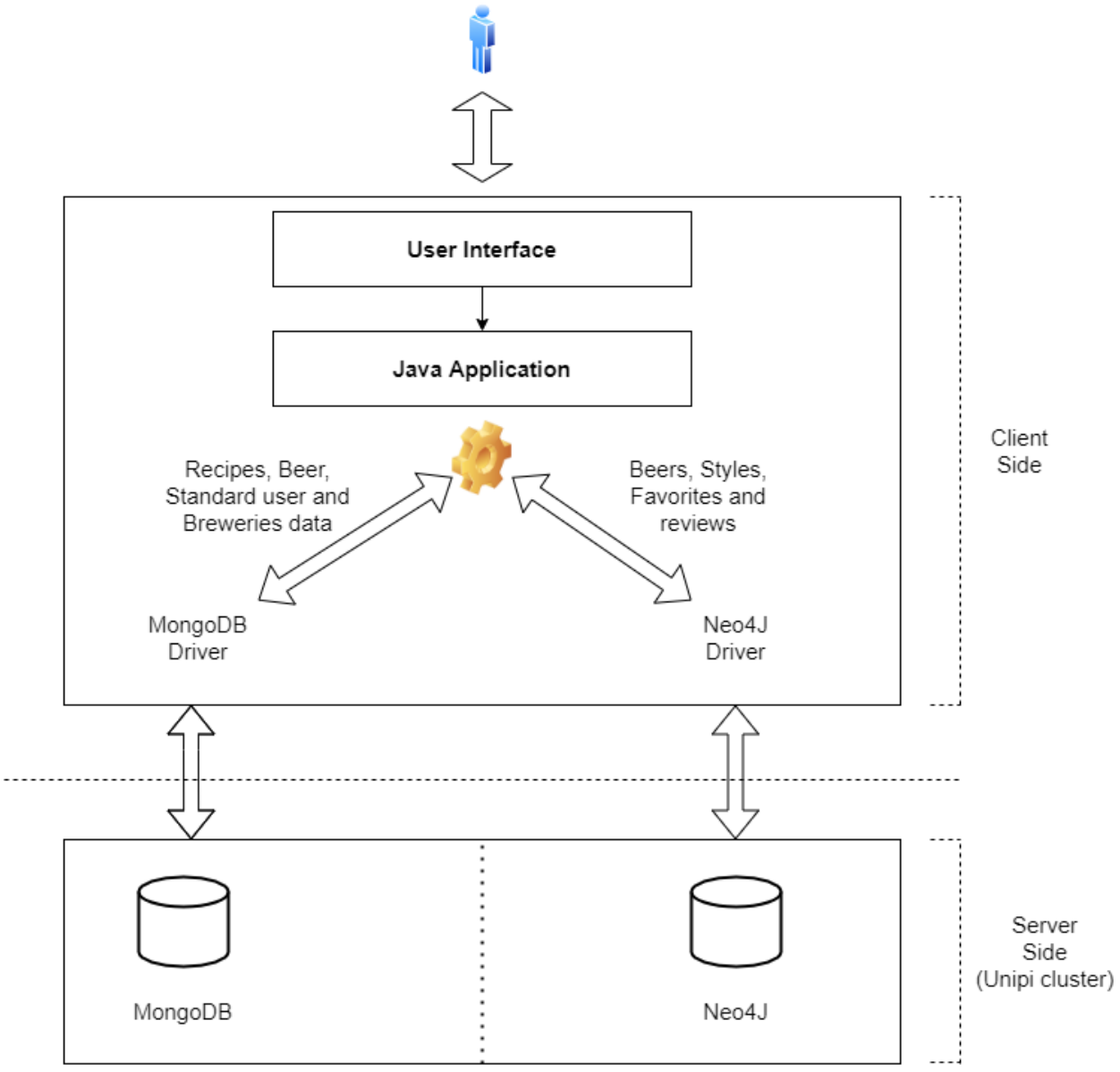
- Review Attributes

Name of the Attribute	Attribute Type	Description
username	String	Name of the user that reviewed the beer
reviewDate	Date	Date of the review
look	double	Beer's look score
smell	double	Beer's smell score
taste	double	Beer's taste score
feel	double	Beer's feel score
overall	double	Beer's overall score
score	double	Beer's total average score

Architecture Model

The application has been separated into two parts:

- A part that plays the role of a client and takes care of displaying the received data by performing queries
- A second part that plays the role of a server and takes care of keeping the service available, maintaining the data and answering the queries of the client instances.



The client-side part is also divided into three parts:

- Frontend: That can be found inside the "gui" folder of the project code. It contains all the Java Swing code needed to show the graphical interface and give a better and interactive experience to the user.
- Middleware: Represented by the entity managers that have the role to keep the consistence between the two remote databases and the local instances of the entities.
- Backend: Represented by the entities and the databases managers, they must create a connection between the local client and the server DBMSs that are on the remote cluster, they also contain all the queries needed to fill the client with real data.

The server-side consists of a cluster of three virtual machines available at 172.16.4.57, 172.16.4.58 and 172.16.4.59:

Server Address	DBMS	Priority
172.16.4.57	MongoDB	It always has the possibility to become the primary server
172.16.4.58	MongoDB and Neo4J	Since it has to handle two different DBMS it become primary only if is the only server online
172.16.4.59	MongoDB	It always has the possibility to become the primary server

The Replica Set has been created to guarantee system availability and avoid single point of failure.

Since the eventual consistency was adopted and the application requires a fast response, we set Write Concern to `W1` (wait for acknowledgement from a single member) and Read Preference to `nearest` (read from the member node which respond fastest).

The configuration of Replica Set is shown below:

```
lsmdb:PRIMARY> rs.conf()
{
        "_id" : "lsmdb",
        "version" : 3,
        "term" : 120,
        "members" : [
                {
                        "_id" : 0,
                        "host" : "172.16.4.57:27020",
                        "arbiterOnly" : false,
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 3,
                        "tags" : {

                        },
                        "secondaryDelaySecs" : NumberLong(0),
                        "votes" : 1
                },
                {
                        "_id" : 1,
                        "host" : "172.16.4.58:27020",
                        "arbiterOnly" : false,
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 1,
                        "tags" : {

                        },
                        "secondaryDelaySecs" : NumberLong(0),
                        "votes" : 1
                },
                {
                        "_id" : 2,
                        "host" : "172.16.4.59:27020",
                        "arbiterOnly" : false,
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 5,
                        "tags" : {

                        },
                        "secondaryDelaySecs" : NumberLong(0),
                        "votes" : 1
                }
        ],
        "protocolVersion" : NumberLong(1),
        "writeConcernMajorityJournalDefault" : true,
        "settings" : {
                "chainingAllowed" : true,
```

Data Model

In this section we will describe the organization of the data in the various DBs.

- MongoDB

MongoDB is the main database we used and is composed by two collections: **users** and **beers**.

- **Users** collection includes data for "standard users", type 0, and for "brewery", type 1, with some different fields. A standard user has an `age` field, whereas a brewery can specify its `types` of venue (like pub or cafe) and maintains an embedded list for the `beers` it produces.
  This collection contains more than 100.000 users, half Standard users and half Breweries, and has a data volume of more than 50MB.

- **Beers** collection stores several information about a beer, like the *name* of the beer, its *style*, a detailed *recipe* with a list of hops, fermentables, yeast and so on, and it also stores an embedded list of all the reviews received by that beer.

  This collection contains about 540.000 beers with 620.000 total embedded reviews, for a total data volume of approximately 350MB.


```
_id: ObjectId("61d2df355bd4d326678c6177")
name: "Vienna Session Lager"
style: "Vienna Lager"
availability: "Year-round"
abv: 4.03
notes: "No notes at this time."
retired: "f"
url: "/homebrew/recipe/view/402438/vienna-session-lager"
method: "BIAB"
og: 1.043
fg: 1.012
ibu: 21.32
color: 11.52
phMash: -1
fermentables: "[[4.536, 'American - Vienna', 35.0, 4.0, 96.8], [0.15, 'United Kingdom..."
hops: "[[42.0, 'Tettnanger', 'Pellet', 4.5, 'First Wort', '0 min', 16.76, 75...."
yeast: "['Wyeast - Munich Lager 2308', '72%', 'Medium', '48', '56', 'No']"
other: "[]"
rating: 2
num_rating: 1
brewery_id: ObjectId("61d32516407f9064b24ab96e")
reviews: Array
  0: Object
      username: "fede"
      date: 2022-01-15T16:24:36.787+00:00
      look: 2
      smell: 2
      taste: 2
      feel: 2
      overall: 2
      score: 2
```


- Neo4J

Regarding Neo4J data organization we choose to create three different types of nodes:

- **User Nodes**: they represent the registered user entity that have at least a relationship with the other nodes; they are needed to keep track of all the interactions between the user and a specific beer.

The attributes that each User node has are Username and ID, needed to identify a specific user. We don't have to explicitly save the user type because Standard Users will be the only type of users in the graph.

- **Beer Nodes**: they represent the beers that are associated to a specific brewery that have at least one relationship with some User nodes.

  The attributes that each Beer node has are Name and ID, needed to show results for the queries and to get more information directly searching for an ID in MongoDB side.

- **Style Nodes**: they represent the currently existing styles for the beers in the dataset. For each beer inside Neo4J we have a relationship that links the beer to its own style that will be used to retrieve suggestions based on the users favorited beers.
  It has a property called 'Name' in which we store the name of the Style it represents.

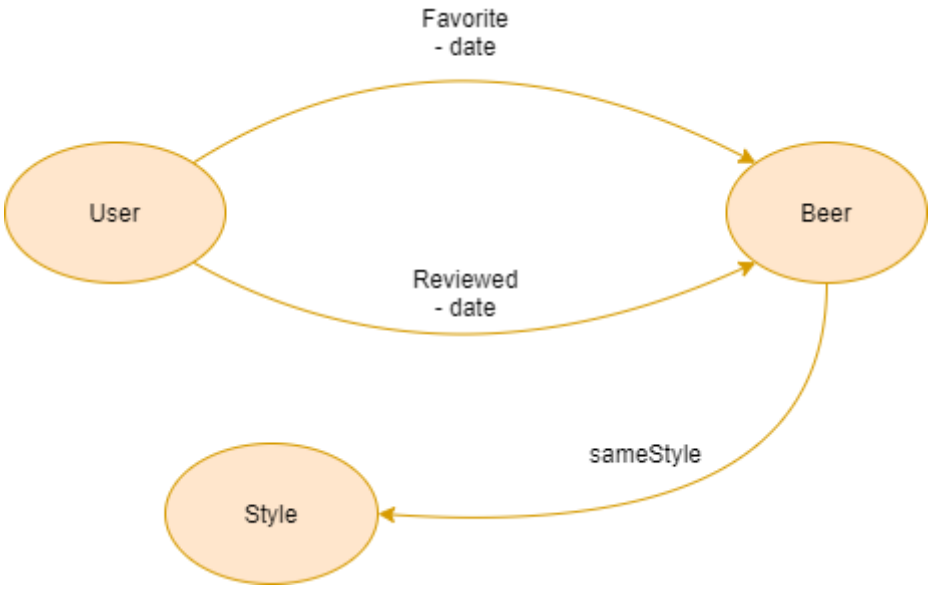
And three different types of relationships:

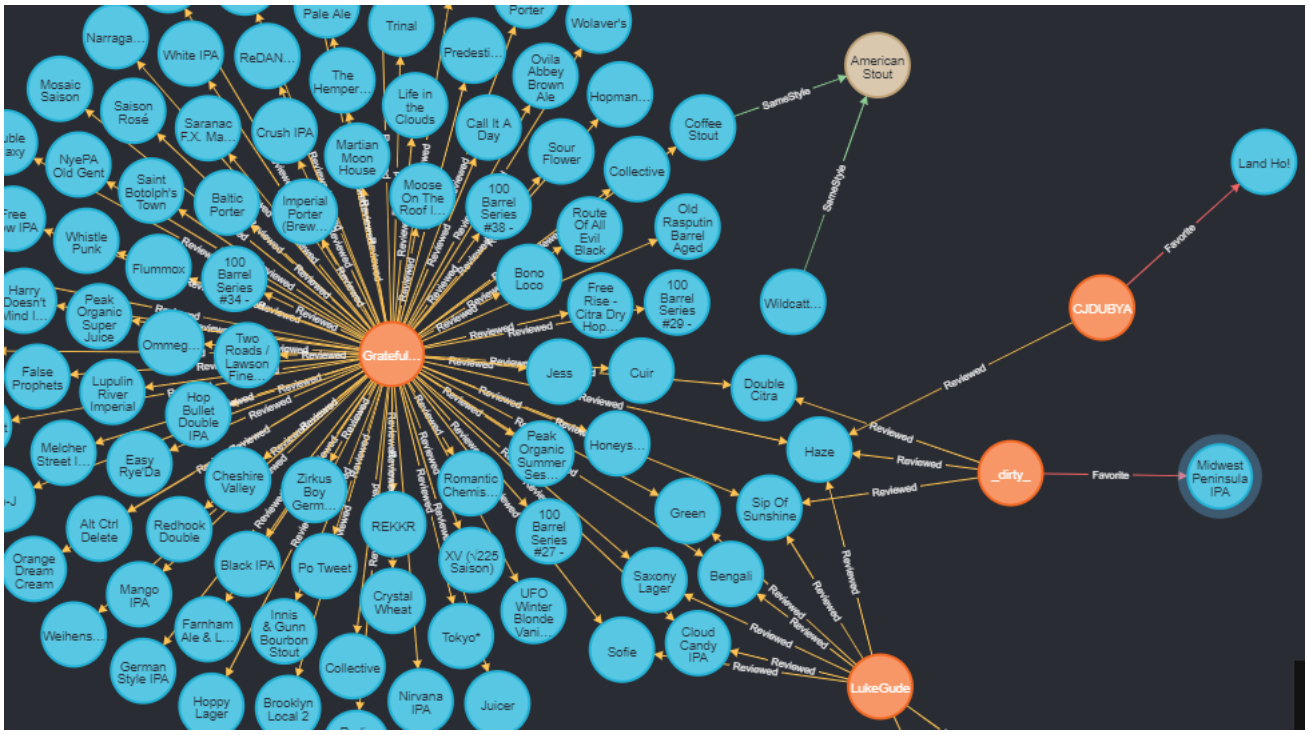
- **Favorite**: Is a relationship between a User node and a Beer node, it represents a user that have added the beer to a favorite. Each time a User try to add a beer to its Favorites at first will be executed a check on the User and Beer existence (since if a user or a beer have not any relationship its node will not be created until something significant must be stored). It has a property called 'date' in which is stored the data (in the format 'yyyy-MM-dd') of the day in which the user put that beer into favorites.

- **Reviewed**: Is a relationship between a User node and a Beer node, is an abstraction to represent the fact that a user has reviewed a specific beer. Even on creation of this relationship a check of beer and user existence is

made. This relationship has a property called 'date' stored in the format 'yyyy-MM-dd' representing the date in which the user reviewed that beer.

- **SameStyle**: Relationship that link a beer to its own Style node, is used to make suggestions based on user favorites. The relationship is created every time a specific beer is inserted in the Neo4J DB. If the style node doesn't exist yet it will be created dynamically with the beer node.





In the above images we can see a generic example of graph implementation in which are showed all the possible relationships between the nodes and an effective one. It's important to notice that these are some relationships not all the relationships for a specific node, so the users may have more favorites or reviews and the beers are linked to their respective style node.

Neo4J Data Volume

Regarding Neo4J we chose to minimize the data that are going to be inside the graph. This choice is given to the fact that bigger is the graph and the bigger is the delay given by each query, since we can't adopt sharding.

Like we said before then we chose to put inside the graph only the nodes that have at least one relationship with some other nodes. A new user won't be inside Neo4J unless it reviews or put a beer in its favorites.

The final volume inside Neo4J DB is:

- A total of 112.833 nodes, divided in:
    - 53.615 Users nodes
    - 58.940 Beer nodes

    - 276 Style nodes
- A total of 708.090 relationships, divided in:
    - 30.016 Favorite relationships
    - 619.740 Reviewed relationships
    - 58.344 same-Style relationships

Analysis

Analysing the Use-case we came up with the following read and write operations that involve MongoDB and Neo4J.

- MongoDB Queries Analysis

Name of the operation	Frequency	Computational cost
Add a User	Low	Average (one read and one write of a new Document)
Find a User by email and password	Average	Low (single read)
Update User data	Low	Average (replace a document)
Delete a Standard User	Low	High (update several embedded document in a collection and delete a document in another collection)
Find Brewery by (part of) name	High	Average (multiple reads)
Compute Brewery score	Average	High (complex aggregation)
Delete a Brewery	Low	High (update two fields in many documents and delete a document)
Add a new Beer	Average	Average (add a new Document in a collection and update another collection inserting an embedded document)
Find a Beer by ID	High	Low (single read)
Find all Beers starting from a part of name or style	High	Average (multiple reads)

Find Beers with the highest score in the last month	Average	Very High (complex aggregation)
Find Beers with a feature score under the brewery score	Low	Very High (complex aggregation)
Update a Beer	Low	Average (replace a document)
Delete a Beer	Low	High (delete a nested document in a collection and an entire document in another collection)
Add a Review	High	Average (one read and one update inserting a new embedded document and updating two fields)
Delete a Review	Average	Low (one update)

- MongoDB CRUD operations

In this section we will show some queries implementing CRUD operations in the application.

Create

Methods called when a Brewery add a new Beer, so the beer is inserted in the beers collection.

```
public void addNewBeerMongo(DetailedBeer beer) {
    try {
        MongoCollection<Document> beersCollection = mongoManager.getCollection( collectionName: "beers");
        Document beerDoc = beer.getBeerDoc();
        beersCollection.insertOne(beerDoc);
        beer.setBeerID(beerDoc.getObjectId( key: "_id").toString());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
public Document getBeerDoc() {
    return new Document("name", beerName)
            .append("style", style)
            .append("abv", abv)
            .append("rating", score);
}
```

Read

Method called when a user searches for a beer inserting a part of its name or style in the field text and clicking the corresponding button, and also when he scrolls through the pages.

```
public FindIterable<Document> browseBeers(int page, @Nullable String name) {
    //check string
    name = name != null ? name : "";
    int limit = 13;
    int n = (page-1) * limit;

    FindIterable<Document> iterable = null;

    try {
        MongoCollection<Document> beersCollection = mongoManager.getCollection( collectionName: "beers");
        iterable = beersCollection.find(or(
                regex( fieldName: "name",  pattern: "^" + name + ".*",  options: "i"),
                        regex( fieldName: "style",  pattern: "^" + name + ".*",  options: "i")))
                .skip(n).limit(limit+1)
                .projection(include( ...fieldNames: "name", "style", "abv", "rating"));

    } catch (Exception e) {
        e.printStackTrace();
    }
    return iterable;
}
```

Another Read operation is performed when an anonymous user wants to subscribe to the application. This query is performed to check if a user with that email or username, the latter only for standard user, already exists in the database.

```java
/* check if an email or a combination of an username/type=0 already exist in the users collection */
public boolean userExists(GeneralUser user) {
    if (user.getUsername().equalsIgnoreCase( anotherString: "deletedUser")
            || user.getUsername().equalsIgnoreCase( anotherString: "deleted_user")
            || user.getUsername().equalsIgnoreCase( anotherString: "deleted user"))
        return true;
    Document doc = null;
    try {
        MongoCollection<Document> usersCollection = mongoManager.getCollection( collectionName: "users");
        doc = usersCollection.find(or(regex( fieldName: "email",  pattern: "^" + user.getEmail() + "$",  options: "i"),
                and(eq( fieldName: "type",  value: 0),
                        regex( fieldName: "username",  pattern: "^" + user.getUsername() + "$",  options: "i")))).first();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return !(doc == null || doc.isEmpty());
}
```

Update

Operation performed when a user wants to review a beer. As the reviews are in the beer collection, a new review is inserted as an embedded document in the review list of the beer it refers to. At the same time, the rating of that beer is updated.

```java
public boolean addReviewToBeersCollection(Review review, DetailedBeer beer, double newRating) {
    if (!existsReview(review.getUsername(), beer)) {
        try {
            MongoCollection<Document> beersCollection = mongoManager.getCollection( collectionName: "beers");
            UpdateResult updateResult = beersCollection.updateOne(eq( fieldName: "_id", new ObjectId(beer.getBeerID())),
                    combine(set("rating", newRating), inc( fieldName: "num_rating",  number: 1),
                            addToSet( fieldName: "reviews", review.getReviewDoc())));
            return updateResult.getMatchedCount() == updateResult.getModifiedCount();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    return false;
}
```

Another update operation takes place when a brewery deletes one of its own beers, so we have to update the beer list in the brewery document.

```
public boolean deleteBeerFromBrewery(DetailedBeer beer) {
    try {
        MongoCollection<Document> usersCollection = mongoManager.getCollection( collectionName: "users");
        UpdateResult updateResult = usersCollection.updateOne(eq( fieldName: "_id", new ObjectId(beer.getBreweryID())),
                pull( fieldName: "beers", eq( fieldName: "beer_id", new ObjectId(beer.getBeerID()))));
        return updateResult.getMatchedCount() == 1;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return false;
}
```

Delete

A delete operation is performed every time a user deletes its own account or when a brewery delete a beer.

```
public boolean deleteUser (GeneralUser user) {
    try {
        MongoCollection<Document> usersCollection = mongoManager.getCollection( collectionName: "users");
        DeleteResult deleteResult = usersCollection.deleteOne(eq( fieldName: "_id", new ObjectId(user.getUserID())));
        return deleteResult.getDeletedCount() == 1;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return false;
}
```

```
public boolean removeBeerMongo(Beer beer){
    try {
        MongoCollection<Document> beersCollection = mongoManager.getCollection( collectionName: "beers");
        DeleteResult deleteResult = beersCollection.deleteOne(eq( fieldName: "_id", new ObjectId(beer.getBeerID())));
        return (deleteResult.getDeletedCount() == 1);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return false;
}
```

- MongoDB Analytics

The aggregation pipelines for MongoDB are shown below. These aggregations are performed to extract statistics about beers or breweries. A snapshot of the piece of code in MongoDB Query language and MongoDB Java Driver is shown for each aggregation.

Top beers of the month

The first aggregation is used to obtain ***the best beers of the month***, those with the highest rating reviewed in the last month. To achieve this, we have to know the time interval in which we want to compute the average score, we select only the reviews whose date is included in that range and group them again by the beer they refer to. We can therefore get the rating of all the beers reviewed in the specific time frame and compute the result sought.

```
[
{$match: {'reviews.date': {
  $gt: ISODate('2021-12-15T00:00:00.000Z'),
  $lt: ISODate('2022-01-15T00:00:00.000Z')
}}},
{$unwind: {path: '$reviews'}},
{$match: {'reviews.date': {
  $gt: ISODate('2021-12-15T00:00:00.000Z'),
  $lt: ISODate('2022-01-15T00:00:00.000Z')
 }}},
{$group: { _id:
  {_id: '$_id', name: '$name', style: '$style', abv: '$abv',rating: '$rating'},
 monthly_score: {$avg: '$reviews.score'}}},
{$sort: {monthly_score: -1}},
{$limit: 10},
{$project: {monthly_score: {$round: ['$monthly_score', 2]}}}
]
```

- **First stage:** *match* -> find only those beers with at least a review received in the last month
- **Second stage:** *unwind* -> deconstruct the array of reviews in the beer collection to obtain a beer document for each review
- **Third stage:** *match* -> only select beers with reviews posted in the last month

- **Fourth stage:** *group* -> group the result of the previous stage by beer, using as _id some useful fields of a beer, in particular `_id`, `name`, `style`, `abv`, `rating`, and compute the average monthly rating of reviews' score
- **Fifth stage:** *sort* -> sort the result obtained by the monthly rating in descending order
- **Sixth stage:** *limit* -> limit the result displaying only the first 10 beers
- **Seventh stage:** *project* -> the project step is used to correctly format the monthly rating, only showing two decimal places

```
public AggregateIterable<Document> getHighestAvgScoreBeers() {
    AggregateIterable<Document> list = null;
    try {
        MongoCollection<Document> beersCollection = mongoManager.getCollection( collectionName: "beers");
        LocalDateTime today = LocalDateTime.now();
        LocalDateTime last_month = LocalDateTime.now().minusMonths(1);
        Bson matchDate = match(and(lt( fieldName: "reviews.date", today), gt( fieldName: "reviews.date", last_month)));
        Bson unwindReviews = unwind( fieldName: "$reviews");
        Bson groupBeer = new Document("$group", new Document("_id",
                new Document("_id", "$_id")
                        .append("name", "$name")
                        .append("style", "$style")
                        .append("abv", "$abv")
                        .append("rating", "$rating"))
                .append("monthly_score", new Document("$avg", "$reviews.score")));
        Bson projectRoundScore = project(new Document("monthly_score",
                new Document("$round", Arrays.asList("$monthly_score", 2))));
        Bson sortScore = sort(descending( ...fieldNames: "monthly_score"));
        Bson limitResult = limit(8);

        list = beersCollection.aggregate(Arrays.asList(matchDate, unwindReviews, matchDate, groupBeer, sortScore,
                limitResult, projectRoundScore));
        /* for (Document d: list) {
            System.out.println(d);
        }
         */
    } catch (Exception e) {
        e.printStackTrace();
    }
    return list;
```

Beers with a feature score under Brewery score

The second aggregation is performed when a brewery wants to know which beers lower brewery's ***average score for a specific feature***. To do that we get

the time frame of the last year and select only the reviews included in this range.

```
[
{$match: {brewery_id: ObjectId('61d32515407f9064b24a58ad'),
 'reviews.date': {
    $gt: ISODate('2021-01-15T00:00:00.000Z'),
    $lt: ISODate('2022-01-15T00:00:00.000Z')}}},
{$unwind: {path: '$reviews'}},
{$match: {'reviews.date': {
    $gt: ISODate('2021-01-15T00:00:00.000Z'),
    $lt: ISODate('2022-01-15T00:00:00.000Z')}}},
{$group: {_id: {
    _id: '$_id', name: '$name', style: '$style', brewery_id: '$brewery_id', abv: '$abv',
    rating: '$rating', num_rating: '$num_rating'},
  avg_feature: {$avg: '$reviews.feel'}}},
{$project: {avg_feature:
  {$round: ['$avg_feature', 2]}}},
{$match: {avg_feature: {
  $lt: brewery_score}}},
{$sort: {avg_feature: 1}}
]
```

- **First stage:** *match* -> find only those beers produced by a specific brewery and containing reviews received in the last year
- **Second stage:** *unwind* -> deconstruct the array of reviews in the beer collection to obtain a beer document for each review
- **Third stage:** *match* -> only select beers with reviews actually posted in the last year
- **Fourth stage:** *group* -> group by beer the result of the previous stage, using as _id some useful fields of a beer, in particular `_id`, `name`, `brewery_id`, `style`, `abv`, `rating`, and compute the average score of the specific feature
- **Fifth stage:** *project* -> the project step is used to correctly format the feature score, rounding the result and only showing two decimal places

- **Sixth stage:** *match* -> take only the beers with the computed feature score under the brewery score
- **Seventh stage:** *sort* -> sort the result obtained by feature score in ascending order

```
public AggregateIterable<Document> getBeersUnderAvgFeatureScore(String breweryID, String feature, double breweryScore){
    AggregateIterable<Document> list = null;
    try {
        MongoCollection<Document> beersCollection = mongoManager.getCollection( collectionName: "beers");
        LocalDateTime today = LocalDateTime.now();
        LocalDateTime past = LocalDateTime.now().minusYears(1);
        Bson dateMatch = match(and(eq( fieldName: "brewery_id", new ObjectId(breweryID)),
                lt( fieldName: "reviews.date", today), gt( fieldName: "reviews.date", past)));
        Bson unwindReviews = unwind( fieldName: "$reviews");
        Bson groupBeer = new Document("$group", new Document("_id",
                new Document("_id", "$_id")
                        .append("name", "$name")
                        .append("style", "$style")
                        .append("abv", "$abv")
                        .append("rating", "$rating"))
                .append("feature_score", new Document("$avg", "$reviews."+ feature.toLowerCase())));
        Bson projectRoundScore = project(new Document("feature_score",
                new Document("$round", Arrays.asList("$feature_score", 2))));
        Bson matchBreweryScore = match(lt( fieldName: "feature_score", breweryScore));
        Bson sortResult = sort(ascending( ...fieldNames: "feature_score"));
        list = beersCollection.aggregate(Arrays.asList(dateMatch, unwindReviews, dateMatch, groupBeer,
                projectRoundScore, matchBreweryScore, sortResult));
    } catch (Exception e) {
        e.printStackTrace();
    }

    return list;
}
```

Brewery score

The third aggregation is used to compute the ***brewery score*** considering the reviews score received by the beers it produces.

```
[
{$match: {brewery_id: ObjectId('61d32515407f9064b24a5fb2'),
  num_rating: {$gt: 0}}},
{$group: { _id: '$brewery_id',
  avg_score: {$avg: '$rating'}}},
{$project: {brewery_score:
  {$round: ['$avg_score', 2]}}}
]
```

- **First stage:** *match* -> find only those beers produced by a specific brewery and who have some reviews (`num_rating > 0`)
- **Second stage:** *group* -> group beers by brewery_id and compute the average of the ratings
- **Third stage:** *project* -> stage used to format the result rounding the rating to two decimal places

```
public Document getBreweryScore(String breweryID) {
    Document doc = null;
    try {
        MongoCollection<Document> beersCollection = mongoManager.getCollection( collectionName: "beers");
        Bson matchBrewery = match(and(eq( fieldName: "brewery_id", new ObjectId(breweryID)),
                gt( fieldName: "num_rating", value: 0)));
        Bson groupBrewery = group( id: "$brewery_id", avg( fieldName: "avg_score", expression: "$rating"));
        Bson projectResult = project(new Document("brewery_score",
                new Document("$round", Arrays.asList("$avg_score", 2))));

        doc = beersCollection.aggregate(
                Arrays.asList(matchBrewery, groupBrewery, projectResult)).first();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return doc;
}
```

Beer Style score

The fourth aggregation is performed when a user want to know the average ***score of a beer style*** in the last year.

```
[
{$match: {num_rating: {$gt: 0},
  style: {$ne: '--'},
  'reviews.date': {
    $gt: ISODate('2021-01-15T00:00:00.000Z'),
    $lt: ISODate('2022-01-16T00:00:00.000Z')
 }}},
 {$group: {_id: '$style',
  total_style_score: {$sum: {
    $multiply: ['$rating', '$num_rating']}},
 total_num_score: {$sum: '$num_rating'}}},
 {$project: {style_score: {
  $round: [
   {$divide: ['$total_style_score','$total_num_score']},2]}
}},
{$sort: {style_score: -1}},
{$limit: 3}]
```

- **First stage:** *match* -> match only those beers with not undefined style and reviewed in the last year
- **Second stage:** *group* -> group by style and computing the total rating obtained by adding the product of the beer rating times the number of reviews for each beer and the sum of the total number of reviews received
- **Third stage:** *project* -> compute and display the result of the division between the total rating and the total number of votes received, computed in the previous step, and round the result to two decimal places
- **Fourth stage:** *sort* -> sort the computed style score in descending order
- **Fifth stage:** *limit* -> limit the result to show to the top 3 styles

```
public AggregateIterable<Document> getTopStyleScore() {
    AggregateIterable<Document> list = null;
    try {
        MongoCollection<Document> beersCollection = mongoManager.getCollection( collectionName: "beers");
        LocalDateTime today = LocalDateTime.now();
        LocalDateTime past = LocalDateTime.now().minusYears(1);
        Bson initialMatch = match(and(gt( fieldName: "num_rating",  value: 0), ne( fieldName: "style",  value: "--"),
                lt( fieldName: "reviews.date", today), gt( fieldName: "reviews.date", past)));
        Bson groupStyle = new Document("$group", new Document("_id", "$style")
                .append("total_style_score", new Document("$sum",
                        new Document("$multiply", Arrays.asList("$rating", "$num_rating"))))
                .append("total_num_score", new Document("$sum", "$num_rating")));
        Bson projectRound = project(new Document("style_score", new Document("$round",
                Arrays.asList(new Document("$divide",
                        Arrays.asList("$total_style_score", "$total_num_score")), 2L))));
        Bson sortScore = sort(descending( ...fieldNames: "style_score"));
        Bson limitResult = limit(3);

        list = beersCollection.aggregate(
                Arrays.asList(initialMatch, groupStyle, projectRound, sortScore, limitResult));

    } catch (Exception e) {
        e.printStackTrace();
    }
    return list;
}
```

- MongoDB Indexes Analysis

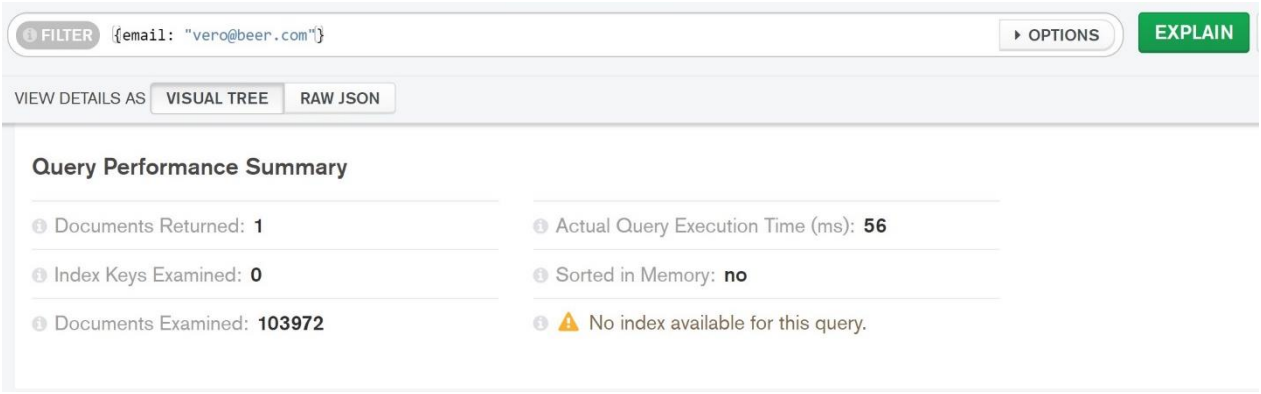
Some indexes have been created to improve the performance of queries. Using Mongo Compass and the explain() function provided by MongoDB, we performed several tests to measure the speed of query execution with and without index.

Users Collection Index

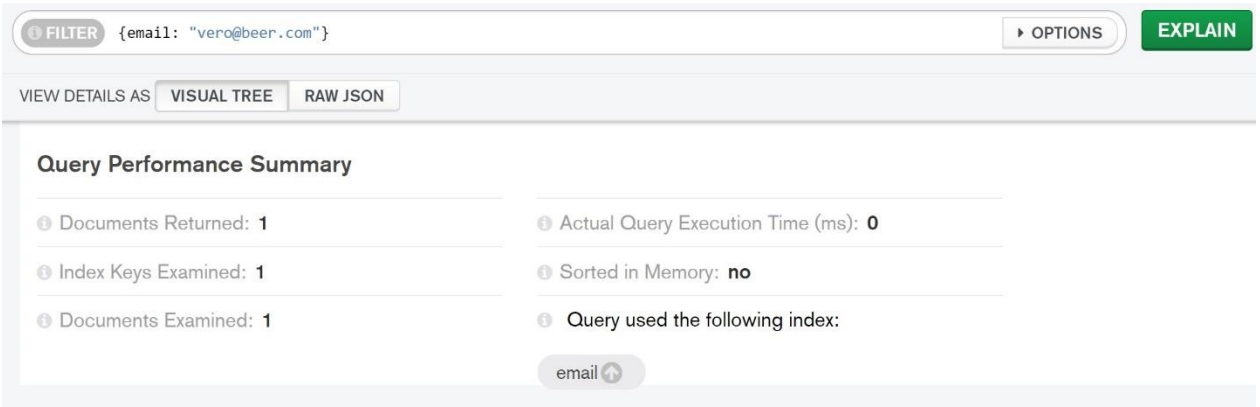
In the *Users* collection we introduced an index on the `email` field and a partial index on `username` field when the user has `type:0`, as shown in the next snapshot.

Name and Definition	Type	Size	Usage	Properties
_id_ _id	REGULAR	3.0 MB	9 since Wed Jan 19 2022	UNIQUE
email email	REGULAR	2.9 MB	2 since Wed Jan 19 2022	
username username	REGULAR	847.9 KB	4 since Wed Jan 19 2022	UNIQUE PARTIAL

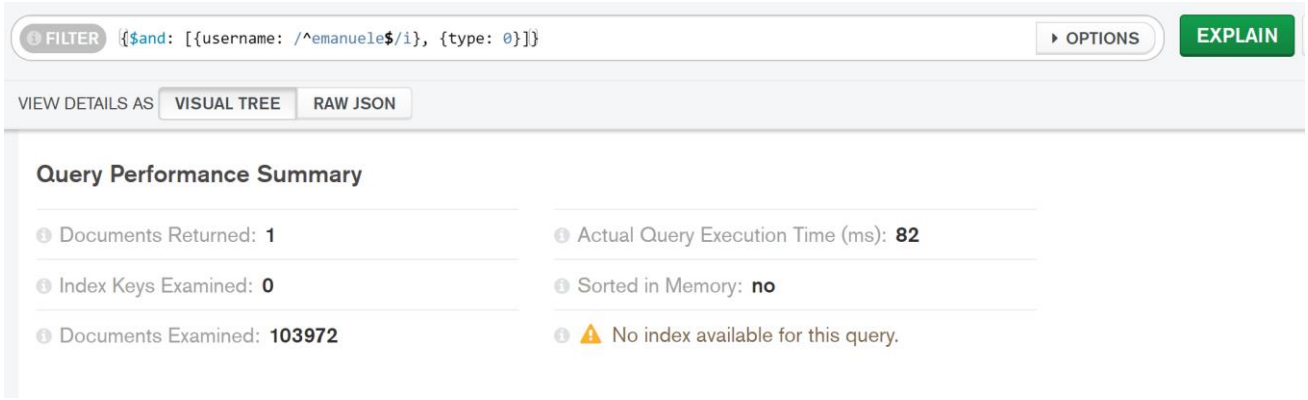
- The first index "`email`" is used to find the matching document of a user, when a login operation is performed, or to check if an email is already associated to an account, when a new user wants to register for the application and we obtain the following results
  - Without index



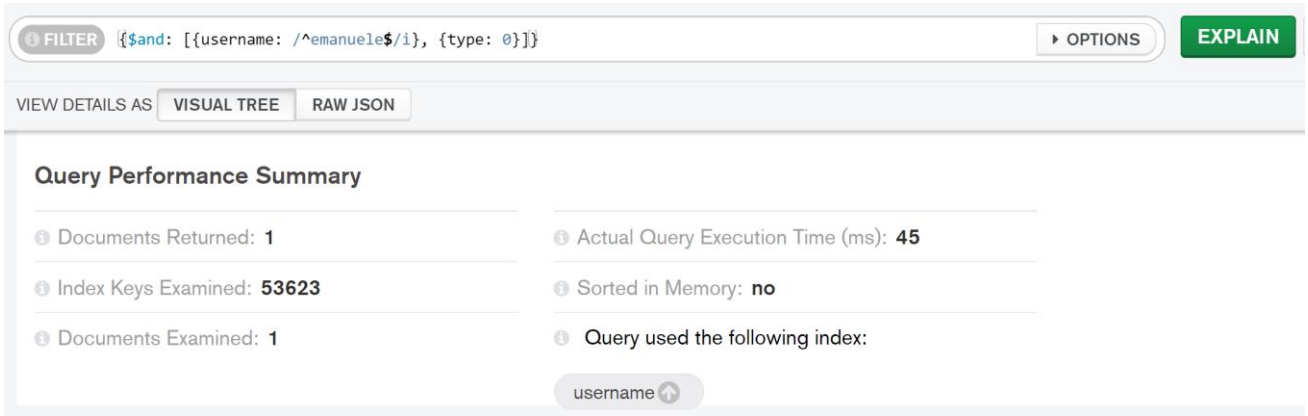
  - With index



- The second index "`username`" is a unique index and is used to find the matching document of a user (when a login operation is performed), or to check if an email is already associated to an account (when a new user want to register for the application) and we obtain the following results:
  - Without index



  - With index

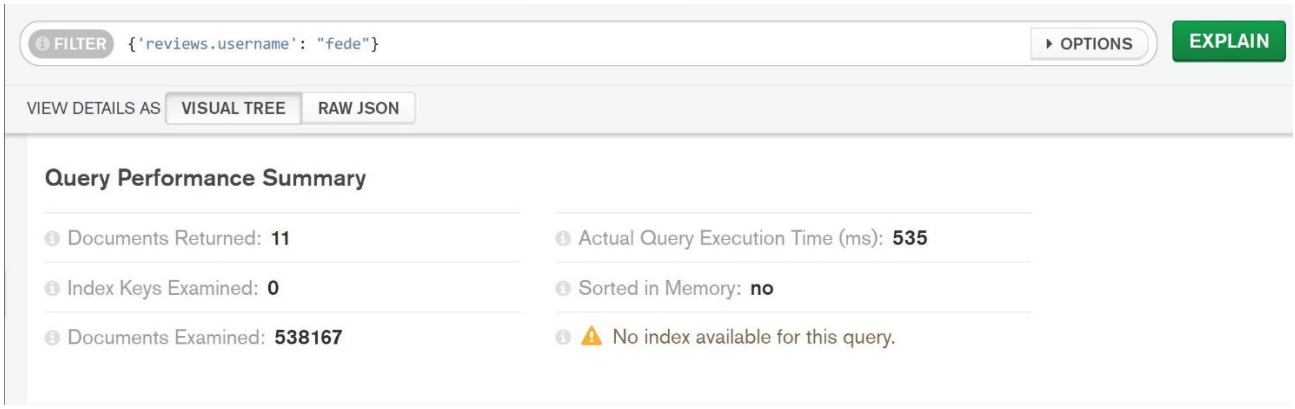


Beers Collection Index

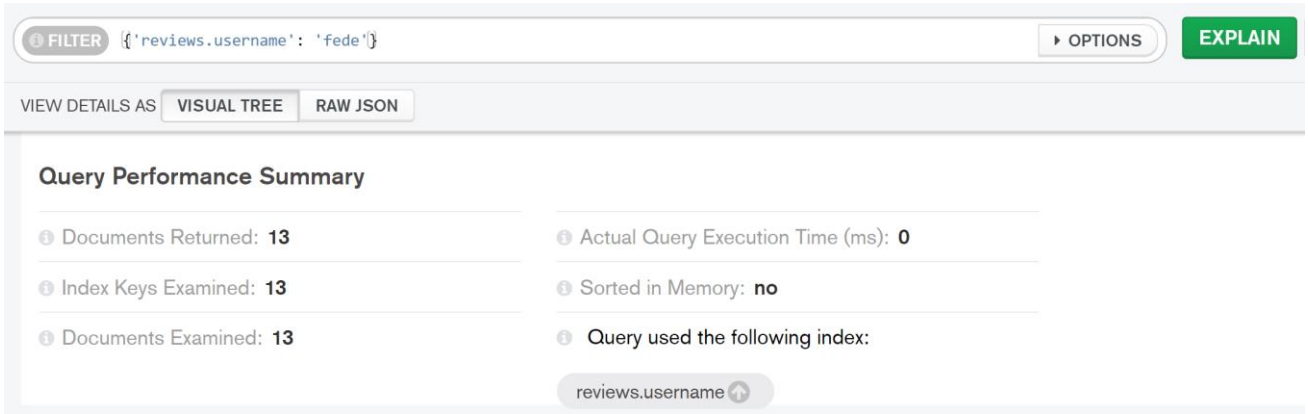
In the *Beers* collection we introduced an index on the embedded fields `reviews.username` and `reviews.date`, as shown in the next snapshot.

Name and Definition	Type	Size	Usage	Properties
_id_ _id	REGULAR	15.6 MB	33 since Wed Jan 19 2022	UNIQUE
date reviews.date	REGULAR	7.0 MB	35 since Wed Jan 19 2022	
username reviews.username	REGULAR	7.7 MB	6 since Wed Jan 19 2022	

- The index "`reviews.username`" is used when an user account is deleted, to find all reviews he wrote and update the username in "deleted_user", obtaining the following results:
  - Without index

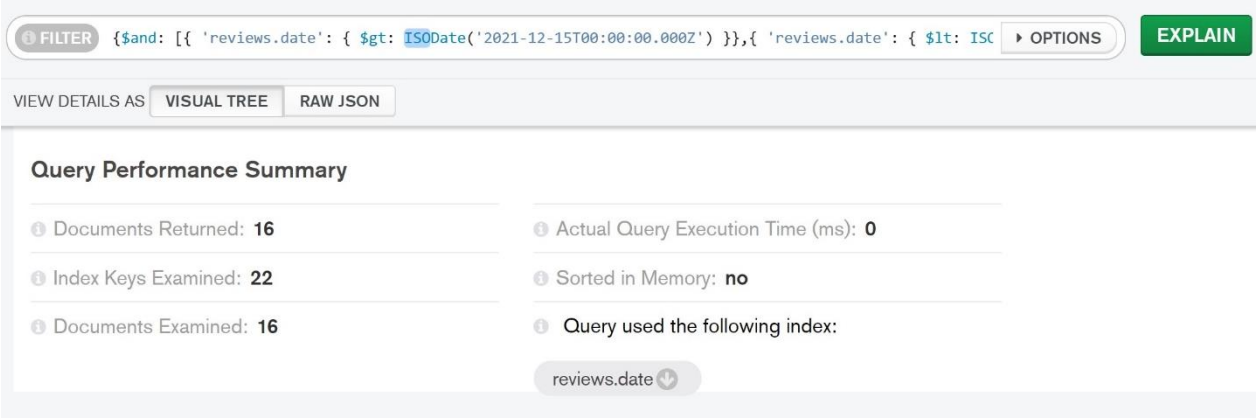


  - With index



- The index "`reviews.date`" is used mostly in the first match stage of aggregation pipelines, obtaining the following:
  - Without index

  - With index



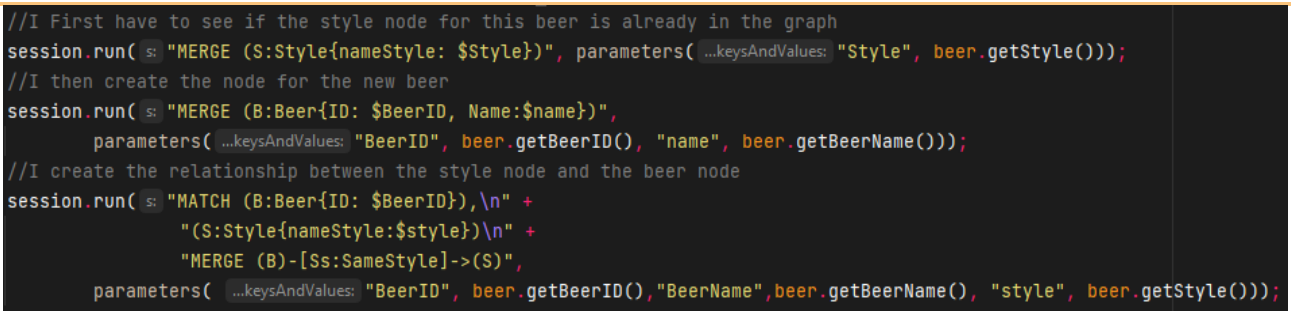
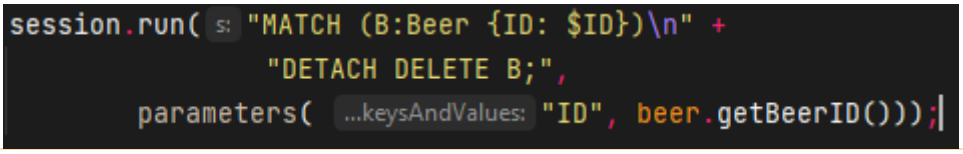
- Neo4J Queries Analysis

Name of the operation	Frequency	Computational cost
Add a new beer	Low	Average (It must create two nodes: Beer and Style, and a relationship if they don't exist already)
Get Suggested	Average	High (multiple reads)
Get Most favorited beers of this month	Average	High (multiple reads and check on date condition)

Remove a beer	Low	Low (Single indexed write)
Add a Review	Average	Average (Single relationship creation and check on existence of two nodes)
Delete a Review	Low	Low (Single relationship deletion)
Get most reviewed beers of this month	Average	High (multiple unindexed reads)
Add to favorites	High	Average (Single relationship creation and check on existence of two nodes)
Remove from favorites	High	Low (single relationship deletion)
Get favorites list	Average	Average (multiple unindexed read)

- Neo4J Queries Implementation

Below we will show the general-purpose queries that implement CRUD operations

Operation	Cypher Implementation
Add a beer	
Remove a beer	

Add a review

```
//Check if user exists
UserManager.getInstance().addStandardUser(review.getUsername());
//Check if beer exists
this.addBeer(beer);
//Put the date in the right format
SimpleDateFormat formatter = new SimpleDateFormat( pattern: "yyyy-MM-dd");
String str = formatter.format(review.getReviewDateNeo());
//Create the relationship
session.run( s: "MATCH (U:User{Username:$Username})" +
               "MATCH (B:Beer{ID:$BeerID,Name:$BeerName})\n" +
               "MERGE (U)-[R:Reviewed]-(B)\n" +
               "ON CREATE\n" +
               "SET R.date=date($Date)",
       parameters( ...keysAndValues: "Username", review.getUsername(), "BeerID", beer.getBeerID(),"BeerName",beer.getBeerName(),"Date", str));
```

Remove a Review

```
session.run( s: "MATCH (U:User {Username: $Username})-[R:Reviewed]-(B:Beer {ID: $BeerID}) \n" +
                "DELETE R",
        parameters( ...keysAndValues: "Username", Username, "BeerID", BeerID));
```

Add a User

```
session.run( s: "MERGE (U:User{Username: $Username}) \n" +
        "ON CREATE \n" +
        "SET U.Username=$Username",parameters( ...keysAndValues: "Username",Username));
```

Add a Favorite

```
//Check if user exists
UserManager.getInstance().addStandardUser(Username);
//Check if beer exists
BeerManager.getInstance().addBeer(BeerManager.getInstance().getBeer(fv.getBeerID()));
//Run the query
session.run( s: "MATCH\n" +
               "  (B:Beer{ID:$BeerID}),\n" +
               "  (U:User{Username:$Username})\n" +
               "MERGE (U)-[F:Favorite]->(B)\n" +
               " ON CREATE SET F.date=date($date)",
       parameters( ...keysAndValues: "Username",Username, "BeerID", fv.getBeerID(), "date", fv.getFavoriteDate()));
```

Remove a User

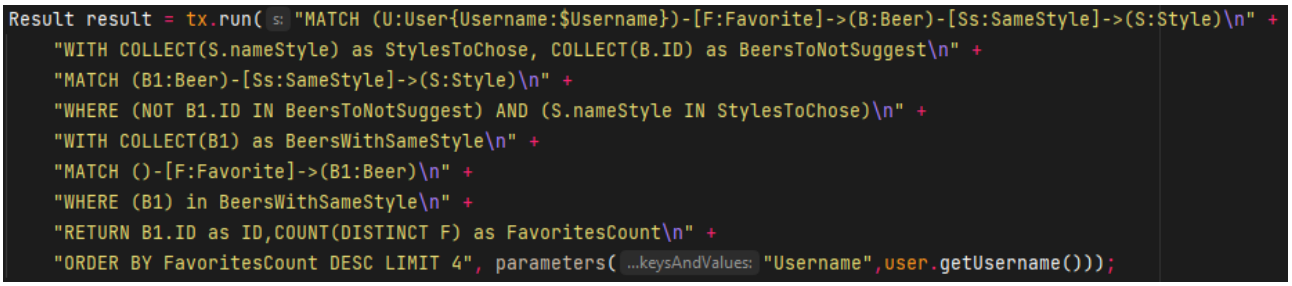
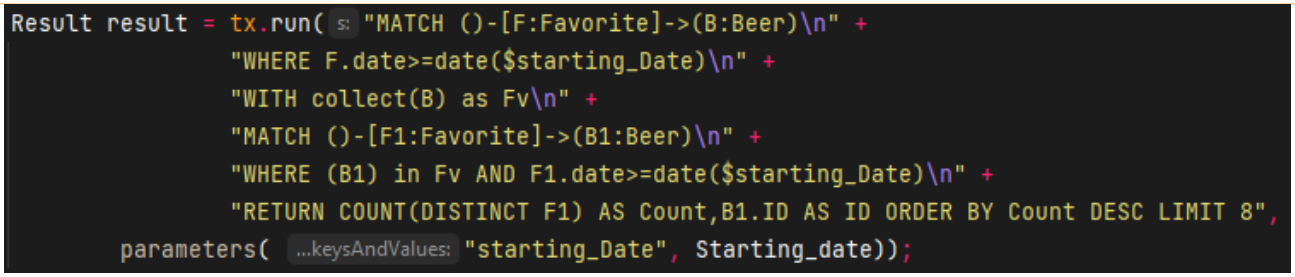
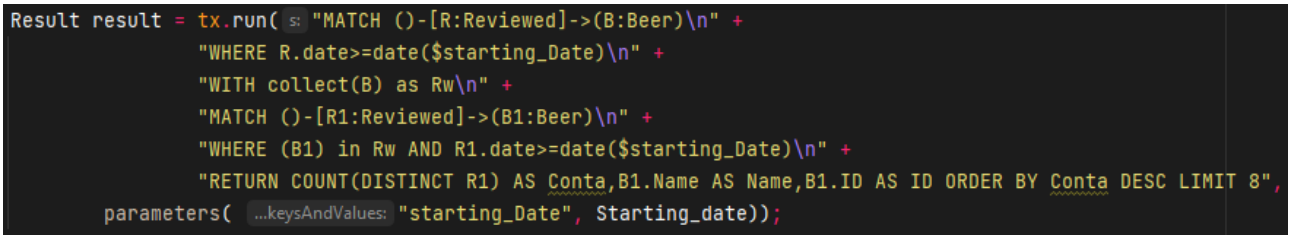
```
session.run( s: "MATCH (U {Username: $username})\n" +
                "DETACH DELETE U",
        parameters( ...keysAndValues: "username", username));
```

Remove a Favorite

```
session.run( s: "MATCH (U:User {Username: $Username})-[F:Favorites]-(B:Beer {ID: $BeerID}) \n" +
                "DELETE F",
        parameters( ...keysAndValues: "Username", Username, "BeerID", BeerID));
```

- Neo4J Analytics Implementation

Below we will show how the analytics are implemented

Operation	Cypher Implementation
Get User Suggestions	
Most Favorited beers of this month	
Most Reviewed beers of this month	

- Neo4J Indexes

To enhance the read operations the following constraints were introduced

Index Name	Type	Uniqueness	EntityType	LabelsOrTypes	Properties	State
BeerConst	BTREE	UNIQUE	NODE	["Beer"]	["ID"]	ONLINE
UserConst	BTREE	UNIQUE	NODE	["User"]	["Username"]	ONLINE

Constraints

ON (beer:Beer) ASSERT (beer.ID) IS UNIQUE

ON (user:User) ASSERT (user.Username) IS UNIQUE

Those constraint were introduced to guarantee the data consistency since MERGE Cypher operation sometimes duplicate nodes or relationships that are already present in the graph.

They are also used as indexes that enhance the data read operation that increase final performances.

Below we will show examples of how the performances changes with or without those constraints on generic queries:

Query:

```
PROFILE MATCH (U:User{Username:"fede"})-[r:Favorite]-() RETURN COUNT(r);
```

Without Constraint:

Operator	Details	Estimated Rows	Rows	DB Hits
NodeByLabelScan	U:User	53.615	53.615	53.616
Filter	U.Username = “fede”	5.362	1	53.615

Expand	(U)-[r:Favorite]-(anon_46)	3.002	10	11
EagerAggregation	Count(r)	1	1	0
ProduceResults	Count(r)	1	1	0
Results: Total DB Hits: 107.242 DBHits				Time: 105ms

With Constraint:

Operator	Details	Estimated Rows	Rows	DB Hits
NodeUniqueIndexSeek	UNIQUE(U:User) WHERE U.Username="fede"	1	1	2
Expand	(U)-[r:Favorite]-(anon_46)	1	10	11
EagerAggregation	Count(r)	1	1	0
ProduceResults	Count(r)	1	1	0
Results: Total DB Hits: 13 DBHits				Time: 13ms

Similar results can be found performing similar read-queries on Neo4J Browser,

Without B:Beer{ID} Constraint:

```
Cypher version: CYPHER 4.2, planner: COST, runtime: INTERPRETED. 117884 total db hits in 61 ms.
```

With B:Beer{ID} Constraint:

```
Cypher version: CYPHER 4.2, planner: COST, runtime: INTERPRETED. 5 total db hits in 7 ms.
```

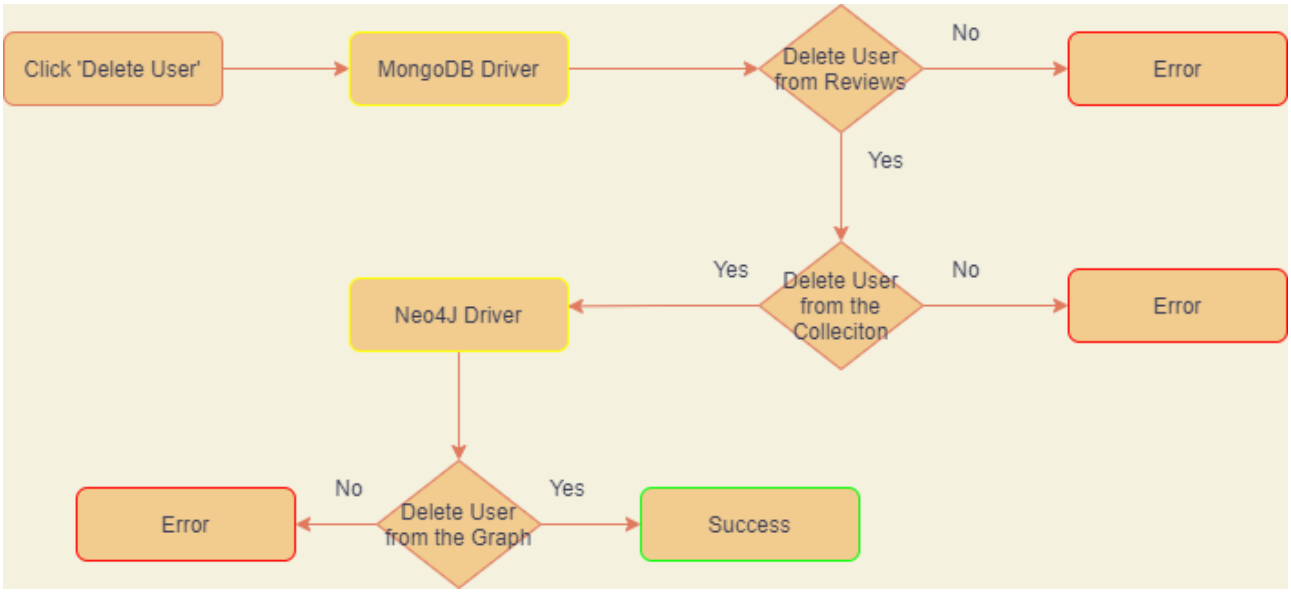
In the end, considering that we are dealing with a social-network, a large volume of queries will be sent to the Neo4J DBMS, making it necessary to manage them in a way that minimizes the user wait-times and makes the experience more fluid and interactive.

Data Consistency

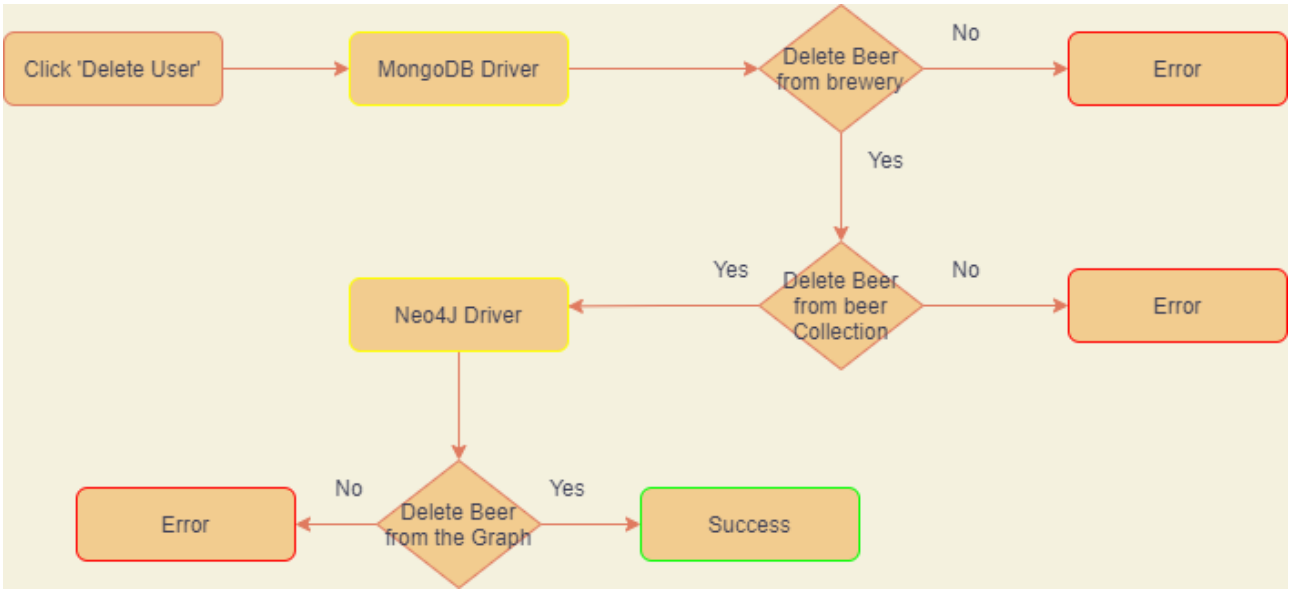
Since we chose to use two different database architectures, Neo4J and MongoDB, the data consistency, between MongoDB documents and Neo4j entities, had to be managed.

An update can, indeed, create inconsistencies between the two databases. This problem occurs only in few cases where we handle data that are present in both the databases, like the reviews and where a beer or a user deletion is performed.

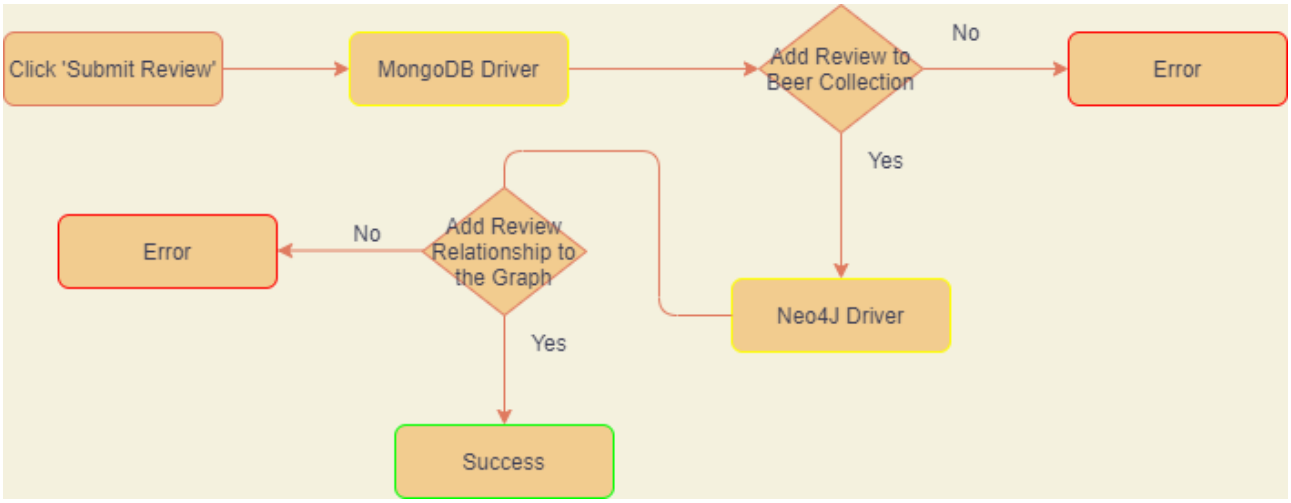
- **User Deletion**



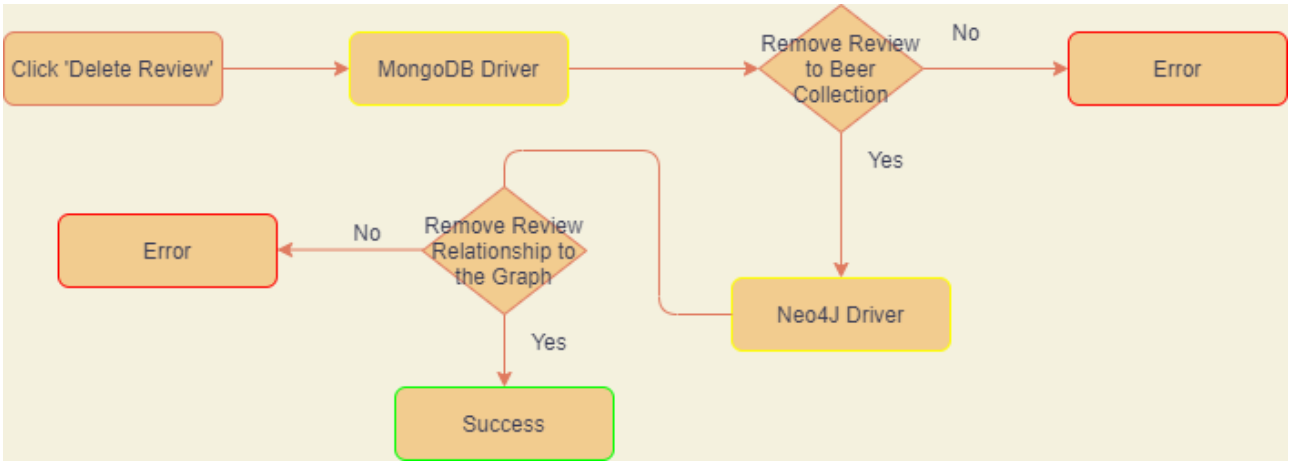
- **Beer Deletion**



- **Adding a Review**



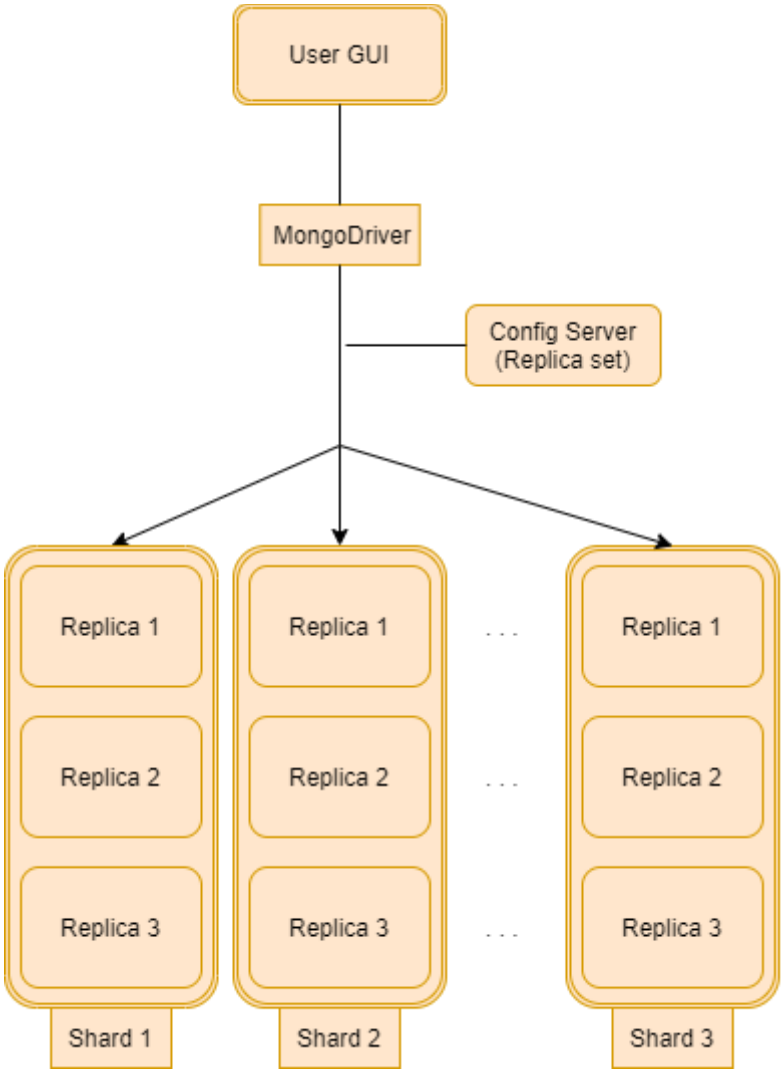
- **Removing a Review**



When an error occurs on the MongoDB side, we prevent the application to upload the data on Neo4J but when the error occurs on the Neo side, we keep the data saved in MongoDB. We chose not to delete data on MongoDB because we didn't want to burden the server with write operations that don't affect the final user experience. Absent data ('Reviewed' relationships in this case) are used to perform an aggregation, the only thing that will be different is the result of that aggregation, not making crucial the management of inconsistency compared to the increase in performance that we may obtain.

Sharding Proposals

Here is shown the structure of how we could implement the sharding mechanism:



Given that we have two different collections we need to define two different sharding keys, one for each collection.

Dealing with the fact that we are handling a social-network we must choose sharding keys that improve or, at least, do not degrade our query performances to guarantee the availability.

To do that, we choose the following sharding keys:

- For the User collection we chose to shard the data by username since are commonly used in our queries, and we can optimize queries within each shard. The other options are to divide the Users by type: "Brewery" or "Standard User"

which allow to divide the data in a more evenly distributed manner or by email, another unique attribute but that isn't much used inside the queries.

- For the Beer Collection we chose to divide the data by the Beer ID ("_id") since it is commonly used as unique parameter to perform queries in the DB. Another option was to divide by beer style to get a more evenly distributed data inside the shards.

Finally, as a strategy we chose to map the sharding key with hashing technique. The field chosen will be mapped through a hash function.