Information Systems and Databases

SIBD Project - Part 3

Report

Shift PB07
Laboratory Teacher: Daniel Mateus Gonçalves

Group 17

Alexandra Gonçalves, 89785	Contribution: 33%	Hours: 30h
Ana Matoso, 89787	Contribution: 33%	Hours: 30h
Inês Arana, 89805	Contribution: 33%	Hours: 30h

In this report, some of the decisions that were made whilst doing the project will be explained. Namely, we will explain some additional triggers that were necessary to implement, the interpretation of the queries, the indexes and the architecture of the app.

1 Integrity Constraints

1.1 IC-1

To implement this integrity constraint, a cursor was used to ensure that, before inserting or updating a record to the table reservation, the dates of the new reservation were compared to the dates of all reservations already on the table. The statement `IF reserva IS NULL THEN`, where reserva corresponds to each record in the table reservation, is used to break the loop when all records have been searched. An exception is raised when the dates of the new reservation overlap with an existing reservation.

1.2 IC-2

This trigger was implemented to respond after the insertion of a new record to the table location. First, the tables from the 3 specialized locations are united using `UNION ALL` (unites with duplicates). Then, the condition `WHERE latitude=new.latitude AND longitude=new.longitude` is used on that table to only compare the records from the new location added. By counting the records left, we can determine if the specialization is correct:

- if `count(*)=0`, the record is on table location, but not on a specialized location, meaning there was no mandatory specialization.
- if `count(*)>1`, the record is on more than one specialized location, breaking the disjoint specialization constraint.
- if `count(*)=1`, the record is on only one of the specialized locations, meeting the mandatory disjoint specialization constraint.

In the first two cases, the integrity constraint is not met, and an exception is raised.

Additionally, this trigger can only be activated if a record is already on the location table, and also the specialized location tables. Thus, when populating the database, since we add a location first to the location table, and only after to the specialized location tables, the activation of this trigger would also raise an exception. As such, the trigger was made `DEFERRABLE`, to only activate after the transaction (i.e. inserting in the table location and in one of the specializations) has been committed.

1.3 IC-3

Before inserting or updating a record on the boat table, we count all distinct locations from the country the new boat is registered to, to ensure there is at least one in the database. If not, an exception is raised.

2 Queries

Some of the assignment queries were somewhat subjective, having potentially more than one interpretation. As such, we will explain below what each of our queries is meant to do.

Query 1: Returns a list of the owners with the most boats registered in each country (meaning, we considered the countries in which boats are registered and not the countries of the owners).

Query 2: Returns a list of owners who have at least two boats registered in different countries, as well as the total number of countries in which each owner has at least one boat.

Query 3: Returns a list of all the sailors who have sailed to every location in Portugal.

Query 4: Returns a list of all the sailors and their reservations by descending order of the number of trips carried out by a sailor in a single reservation.

Query 5: Returns a list of all the sailors who have had reservations by descending order of the sum of the trip durations for a single reservation.

3 Additional Triggers

In the file `triggers.sql` there are some additional triggers that were added in order to give consistency to the app. We will explain them by order of appearance in the file.

Firstly, a trigger was created to prevent a person being added to the table person if they are already there. This comes in handy when adding sailors that are already owners (thus being already in the table person) and vice versa. Then, two triggers were created so that when deleting sailors/owners, if the person is not in the owner/sailor table respectively, then it deletes them also from the table person. These three triggers are essential in order to maintain the mandatory specialization in sailor or owner.

Additionally, a trigger was created so that when adding a schedule (needed before adding a reservation), if that schedule is already in the table, then the action of adding it to the table is aborted. Similarly, a trigger was created so that when deleting a reservation, it checks if there are more reservations with that same schedule and if so, aborts the deletion from the schedule table.

4 Web Application

The application can be acessed using a web browser and the link: http://web2.tecnico.ulisboa.pt/ist189805/homepage.cgi.

The architecture of the web application can be found in figure 1. Firstly, the page that is initially called is the one corresponding to the `homepage.cgi` file where one can chose what operation is to be done (one hyper-reference per question) which are shown in the second layer of figure 1.

In each of these pages (e.g. `owner.cgi`), you can choose if you wish to add or remove the respective object and in the case of sailors if you wish to list them (hyper-reference to `list_sailors.cgi`).

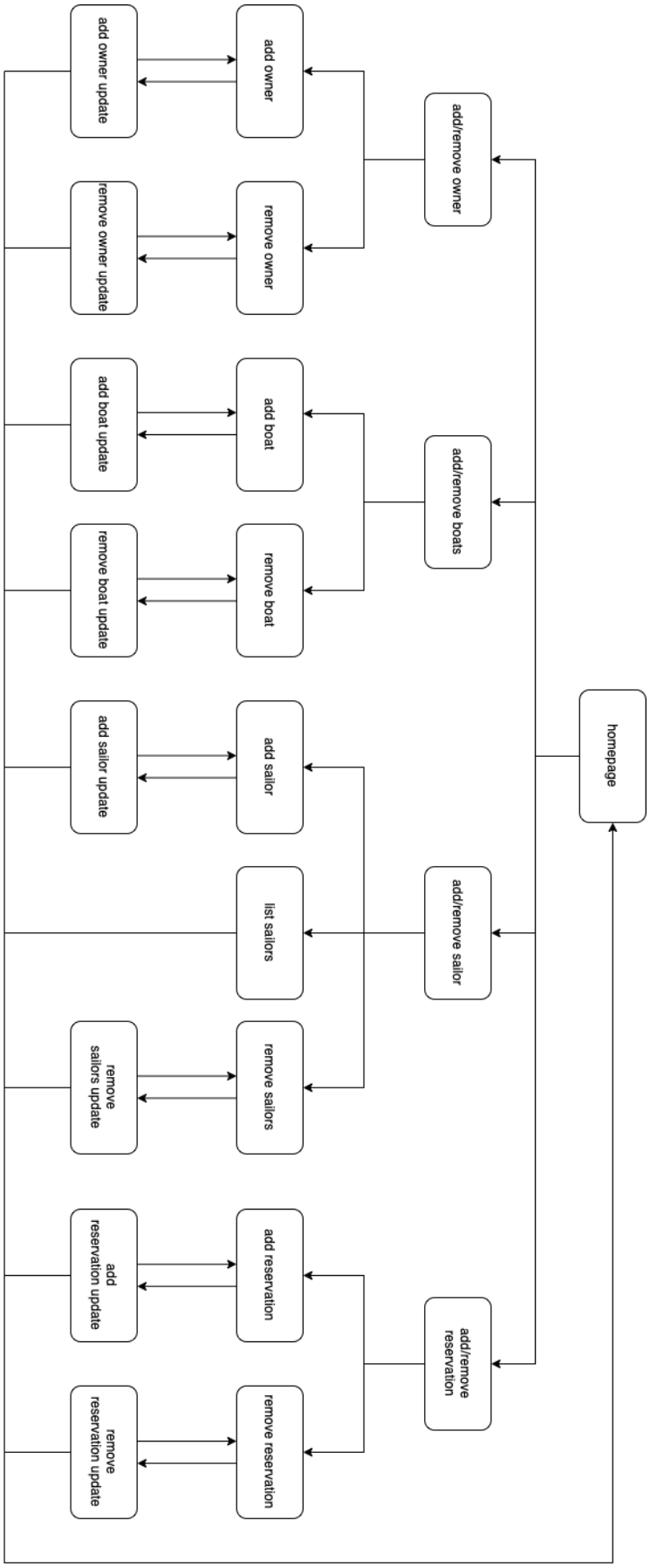

Figure 1: Architecture of the web application. Every box represents a `.cgi` file in the web directory and every arrow a hypertext reference or a submit button.

Then in the add (e.g. `add_owner.cgi`) and delete (e.g. `del_owner.cgi`) pages of all objects there are forms to be filled with the information needed to perform the action. It is important to note that the parameters need to be valid (e.g. the ISO code needs to be in the country table) or else an error message is shown (figure 2).

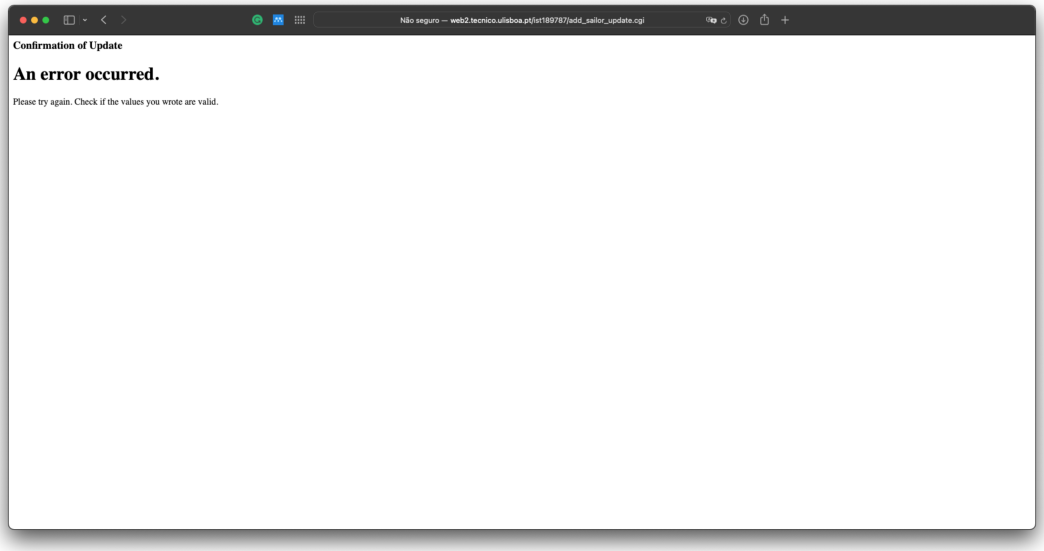

Figure 2: Error message shown if the parameters for adding/removing a sailor/owner/boat/reservation are invalid.

Additionally, when adding owners, if the owner is not in the person table, then it adds them automatically and when deleting owners, and if the person is not a sailor, then it automatically deletes the owner from the table person as well. This mechanism of automatically adding and deleting from the person table also happens when sailors are added/deleted and is due to the triggers already explained.

Moreover, when adding boats, the owner needs to be already in the owner table. Plus, in the forms for adding and deleting boats, the MMSI is an optional parameter that if filled, the boat is also added/removed in the boat_vhf table (see `add_boat_update.cgi` file).

In addition, when adding reservations, due to the triggers explained above, if the reservation is in a schedule that is not present in the schedule table, then it adds it, and similarly when deleting, if there are no other reservations for that schedule, then it deletes the schedule from the schedule table as well.

The pages with the names ending in `_update.cgi` receive the data from the respective forms, execute the requested queries, and if they are able to perform the action successfully, a confirmation page is shown (e.g. figure 3) but otherwise shows the error message mentioned above. Additionally, there are hyper-references to perform the same action or to go back to the homepage if the action is successfully completed.

Finally, it is also important to note that each operation a person executes in the app is performed as a single transaction which means that it is performed as a whole and if there is any error in the middle of the transaction, it is rolled back thus reverting the transaction. This is done by using the command `connection.commit()`. Since autocommit is by default false, the commit is only performed when explicitly written instead of every time a `cursor.execute()` is called. Transactions are also important to guarantee atomicity of related operations so that each transaction is done sequentially.

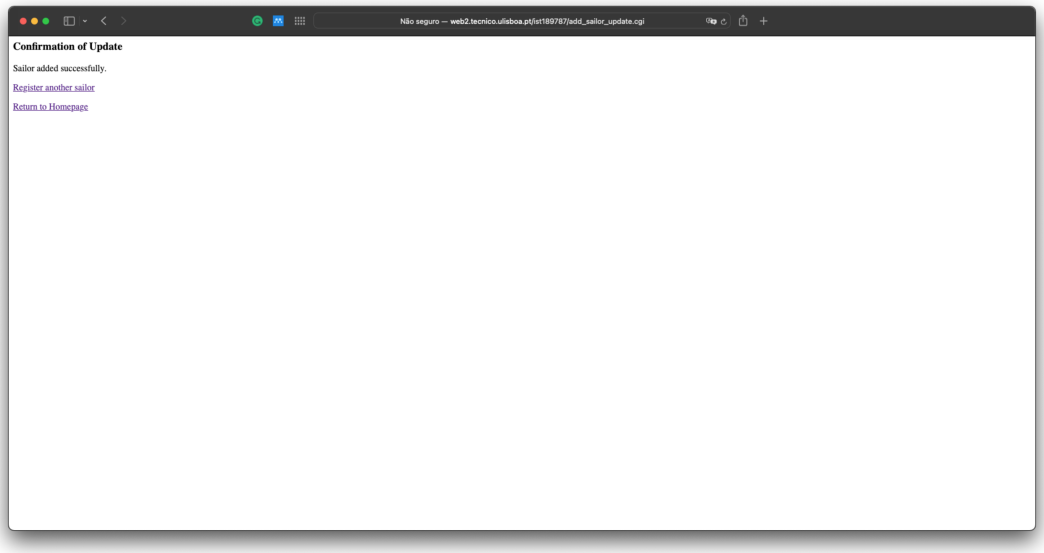

Figure 3: Confirmation message shown if the action is successful.

5 Indexes

5.1

For this query, there are two attributes being searched: year and country name. For the attribute year, it is a range query (all boats registered from <some year> on). Thus, a B-tree index is the most appropriate, since it can navigate through the records according to whether a certain attribute is higher or smaller than the attribute of another record. For the country name attribute, we are searching for a specific value (point query), thus a hash index can be used to help the process. A hash index divides the records in buckets, searching only the bucket containing the record selected by the query. The smaller the number of records in each bucket, the faster the query. For this case, the number of country names is naturally limited, and as such the number of records per bucket will always be small enough to ensure the efficiency of using a hash index. Additionally, since the country name is unique, the search is faster, since there are no duplicates. Therefore, the following instructions would be used to create the indexes:

```
CREATE INDEX year_idx ON boat(year);
CREATE INDEX name_idx ON country USING HASH (name);
```

5.2

For this query, there are two attributes being searched: start date and location name. For the start date attribute, we have a range query, which indicates that the most adequate index would be a B-tree index, since a Hash index does not support range queries. The index will navigate through the records according to whether the dates are higher or smaller than a certain record. For the location attribute, a pattern is being searched. Since the pattern can be in the beginning, middle or end of the location name, no index can make the search more efficient, since all records need to be searched anyways. Hence, the following command would be used to create the index:

```
CREATE INDEX start_date_idx ON trip(start_date);
```