INSTITUTO SUPERIOR TÉCNICO

INFORMATION SYSTEMS AND DATABASES

Database Project, Part 3

AUTHORS

Name: Ana Carolina Lima	**Number:** 83993	**Participation:** 33.3% (24h)
Name: Carlos Cardoso	**Number:** 87161	**Participation:** 33.4% (24h)
Name: Luís Coelho	**Number:** 90127	**Participation:** 33.3% (24h)

Group 38 - Shift a_SIBD77L05
Prof. Inês Filipe

2020/2021 - 1ST SEMESTER
16th of December of 2020

1 Integrity Constraints

In this part of the project we had to implement many different Integrity Constraints which we did by creating the needed triggers & stored procedures.

For the first integrity constraint (IC1) we had to make it so that for each transformer-busbar connection the primary voltage of the transformer matched the voltage of the busbar connected to the primary side of the transformer.

For the second integrity constraint (IC2) we had something similar which was to make it so that for each transformer-busbar connection the secondary voltage of the transformer matched the voltage of the busbar connected to the secondary side of the transformer.

This was implemented by creating a BEFORE INSERT or UPDATE type of trigger called "check_Voltages" along with a stored procedure "check_V()" that selected the voltage from the busbar table whose id's matched the primary and secondary busbar id's (pbbid and sbbid) which we want to insert/update and compared them to the transformer's primary and secondary voltage (pv and sv). If these values matched the conditions where fulfilled and the insert/update in the "transformer" table would be made, otherwise an exception would be raised, warning the user that the operation was invalid, informing him/her of the reason and the insert/update operation would be cancelled.

For the fifth integrity constraint (IC5) we had to make sure that for each analysis concerning a transformer the supervisor name and supervisor address (sname and saddress) of the supervisor, that supervises the substation where the transformer is located, didn't match the analyst's (that analyses that transformer's incident) name and address (name and address from the "analysis table"). This basically means that a person cant analyse an incident with a transformer that occurred in a substation which they supervise.

This was implemented again using a BEFORE INSERT or UPDATE type of trigger called "check_Analyses" along with a stored procedure "check_A()" which at first checks if the analysis the user is trying to insert/update is concerning a transformer. If so, the procedure ends and the insert/update is made. Otherwise that means the analysis is concerning a transformer so we still have to check (IC5). For that and still inside the same procedure we check if the supervisor's name and address (sname and saddress) correspondent to the substation where the analysed incident's transformer is located matches the analyst's name and address we are trying to insert/update.

2 View

For the third task we created a view named supervisors which returns a table with the identification of the supervisors (name,address) and the number of substations each of them supervises. All of this while not including the supervisors that do not supervise any substation. This was done by creating a query. For this query we created a table using a LEFT OUTER JOIN between the supervisor's table and the substation's table on name = sname and address = saddress and used a WHERE condition such that in that table the we didn't include rows where the gpslat values where null (not including the supervisors that do not supervise any substation). From that table we selected the identification of the supervisor (name, address) and count(*) grouping it by name,address in order to also get the number of substations each supervisor supervises.

3 SQL Queries

For the first query given that each person is identified by the primary key (name,address) in order to return the analysts that have analyzed the incidents regarding element 'B-789' we need to query the analyses table and from there select the distinct name-address combinations whose corresponding analyses id are equal 'B-789'.

The second query was implemented by creating a table with the name and address of the supervisors that supervise substations south of Rio Maior (Portugal) (substations where gpslat values are lower than 39.34). From there getting the supervisors that do not supervise substations south of Rio Maior is the equivalent of selecting every name,address combination from the supervisor's table which didn't match any combination in the table we created before.

In the third query where we wanted to get the elements with the smallest amount of incidents the implementation was easily made by creating a table with a LEFT OUTER JOIN between the element's table and the incident's table on element.id = incident.id. From that table counted the number of incidents of each element in order to know the minimum value of number of incidents of that table. Then we created the same table but with the having condition which made it so that we only selected the elements whose number of incidents matched the minimun number of incidents. Giving us the elements with the smallest amount of incidents even if those elements had zero incidents.

In the forth query we wanted to know how many substations does each superviser supervise. For that we created a table with a LEFT OUTER JOIN between the supervisor's table and the substation's table on name = sname and address = saddress. Then, from that table we selected the name, address and count(distinct gpslat) this last selection was formed in this way so that we counted the rows where gpslat values are null (rows where the supervisor doesn't supervise any substation) as 0.

4 Application

The working version of the web application can be found at: `http://web2.tecnico.ulisboa.pt/ist425293/powergrid.cgi`.

A few assumptions are made to simplify the program, such as that a new supervisor has to exist in the database to begin with before they can be assigned to supervise a substation, or analyse an incident.

Overall, the application is not as extensible as it could have been, but it still supports new elements with no adjustments made to the files, as long as they also have `id` as their primary key, and with minor adjustments in other cases.

4.1 Program Flow

A flowchart that explains how every file is connected can be seen in 1.

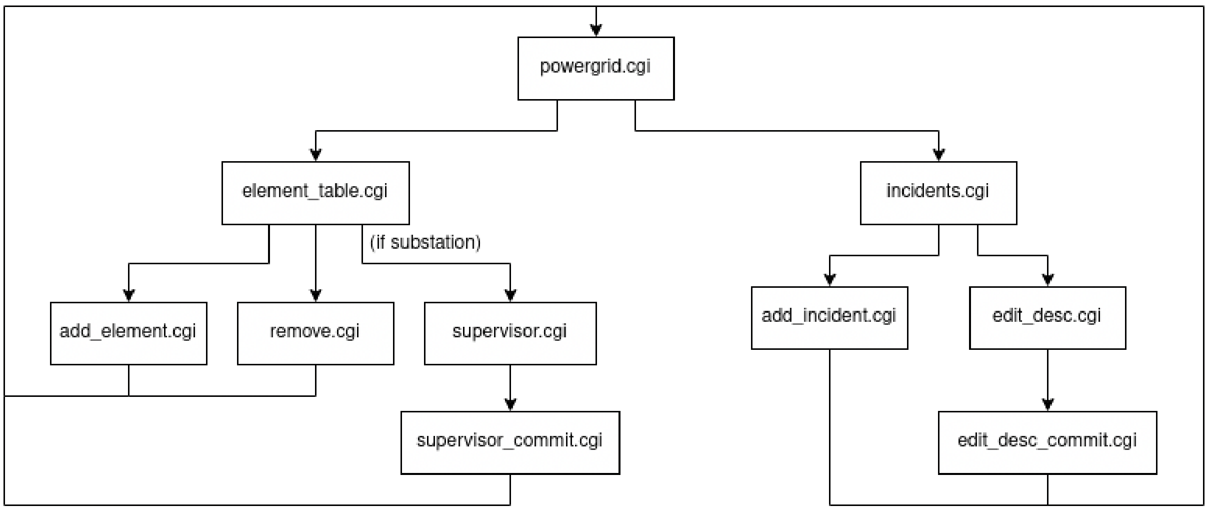

Figure 1: Flowchart representing the flow of the application.

4.2 Program Files

- **powergrid.cgi** The main landing page of the application. The dropdown menu allows the user to choose the elements whose table they would like to see. There is also a link to the incident table.
- **element_table.cgi** Shows the table that matches the element that the user picked in the previous page (powergrid.cgi). There's very few things that are hardcoded, making it easily extensible if a new element is added to the schema. There are fields that allow the user to insert a new element and to delete a element that is already present in the table. If the element is a substation specifically, it also allows the user to change its respective supervisor.
- **add_element.cgi** Adds the new element to the correct table if the user has filled out the forms to add a new element.
- **add_supervisor.cgi** Shows the appropriate form to add a new supervisor to a substation. It's separate from the substation table because it would make the page cluttered and hard to read.
- **remove.cgi** Deletes the selected element from the table and returns to the main page.
- **supervisor_commit.cgi** Adds the new supervisor to the database and returns to the main page.
- **incidents.cgi** Shows the incident table and lets the user add a new incident.
- **add_incident.cgi** Adds the new incident to the database.
- **edit_desc.cgi** Opens up a new page for the user to write in a new description for the incident.
- **edit_desc_commit.cgi** Updates the new description to the database.

5 Indexes

5.1

In this query we decided to create two different indexes: one for the GROUP BY statement and another for the WHERE statement. For the GROUP BY statement, in order to improve the performance of the query, we created a binary tree index on the record's locality of the table Substation. This way, when the query begins and the group by is initiated, the search for each different locality is done using the ordered b-tree.

For the WHERE statement, where we are dealing with a "Point Query", we decided the use a hash index to increase the query's speed/performance. In this statement we are just looking for a specific value named "some_value" in the column pv of the table Transformer and, because of that, by creating a hash index on the column pv in the Transformer's table we go from a previously existing table scan to just an index scan, where only the structure accessed is the bucket corresponding to our hash function where the desired value is located. If by chance we were to use a b-tree index in this case, we would have to scan, in the worst case possible, a number of records equal to the height of the tree. This means having a complexity of $O(|K|log\frac{n}{2})$, where K is the number of records and n is the b-tree fan out, compared to the much simpler complexity of O(1) searching a given value with a hash index, where we just apply the hash function to the pv value we are searching. If multiple values of pv have the same result when passing through the hash function, we would have to scan the records present in the same bucket as our desired data. Nonetheless, the complexity of the search would be lower than the one of the binary tree.

```
CREATE INDEX index_pv ON transformer USING hash(pv);
CREATE INDEX index_locality ON substation(locality);
```

5.2

In this query, in order to improve its performance, we decided to use one composite b-tree index on the columns (description,instant). Besides being faster than having two separate b-tree indexes, we decided to the use this type of index because the query in question is a range query, meaning that a hash index wouldn't suffice.

As for the order chosen for the columns in the index, we decided to consider the most general case possible, where the variation of the instants, which covers a timestamp up to its seconds, was much bigger than the variation of the descriptions. That being said, by making the index filter the records by their descriptions first, which is a more selective attribute, we would eliminate more records in the first scan, making the second search, on the instants of the records, much faster.

However, in the WHERE statement of the query, although we have a LIKE clause in the second condition, that condition covers a range of descriptions too, as we have a wildcard '%' in the end of the description we are searching for. This wildcard means that every record whose description starts with a given pattern can be selected by this query. That being said, if by chance we were to give a very generic pattern to the error description, like the letter "E" for error, we would be covering the entire table, or most of it at least, meaning that the previously declared index would no longer be able to optimize the query to its fullest potential. In this case we would have to swap the order of the arguments in the binary tree index, making the composite index on the columns (instant, description) of the table incident the best in order to increase the performance of the given query.

```
CREATE INDEX index_incident ON incident(description, instant);
```

6 Multidimensional Model

When creating the desired Multidimensional Model, we started by creating each dimension table and the fact table using the same data types used in the given ".sql" file. As for the new attributes, not present in the file, we chose to make the element type a VARCHAR and to declare each attribute of the time dimension as a NUMERIC value, to make the comparison with the timestamp easier. In each dimension table we declared its ID as the table's primary key and, for the incident fact table we declared the combination of all the ID's present in the table as its primary key. Each ID present in the fact table, as it is referring to a certain dimension, was also declared as a Foreign Key pointing to its respective dimension table. The created star schema can be seen in the following Figure:

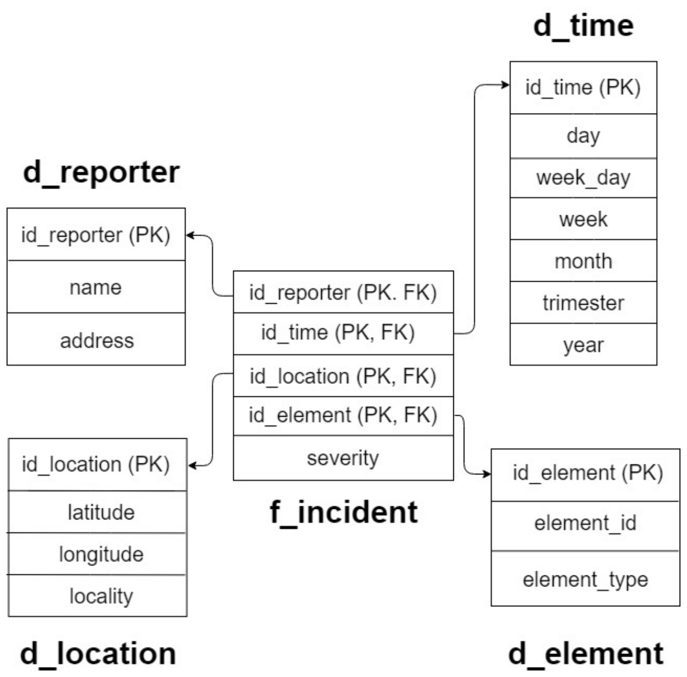

Figure 2: Schema of the Multidimensional Model created.

Then, we began to load the newly created star schema by starting with the dimensions table. To load every dimension table, we queried the corresponding table joining it with the Incident table, as it can be seen in the etl.sql file. For example, when considering the Location dimensional table we decided to insert each unique location from the Substation's table, also present in the Incident table joined with the Transformer table, into its corresponding dimension table. As for the Time dimension table, in order to extract the desired information from each unique instant, which was in the timestamp format, we used the function DATE_PART(), giving it the argument corresponding to the time format we were trying to obtain. For the day of the week we chose the argument 'isodow', which extracts from a given timestamp its day of the week in the numeric format, considering Monday as the number 1 and Sunday as the number 7. As for the month, we considered 1 as being January as 12 as being December. One last important decision we made, regarding the loading of the dimension table was the creation an unknown location type, which was inserted in the first row of the locality dimension table. This assures that incidents with elements that do not have an established and unique location, like bus bars or lines, can still have an id_location in the incident's fact table, instead of having a NULL value, which wouldn't be possible considering the fact that the id_location is also a primary key in that table.

Finally, we load the incident fact table, as show in the etl.sql file, by querying the Analyses table, joined by a few other tables, row by row and selecting the corresponding id for each dimension registered. When a NULL value of a location is registered, we give the attribute id_location the value 1, in order to point to the unknown entry in the location dimension table.

7 Data Analytics Queries

In this final section, we were asked to write an SQL query that would allow us to analyze the total number of anomalies, or incidents, reported by their severity, locality and week_day. To do this, as no grouping order was specified for the number of incidents, we decided to use the group by cube statement on the attributes severity, locality and week_day, as it is the most flexible grouping tool available, allowing us to obtain information for every combination of attribute values. If, for example we were to use a GROUP BY ROLLUP statement, we would have a hierarchy of attributes, meaning that grouping sets such as (NULL, locality, week_day) would not have been possible to obtain. We could also use the union of multiple GROUP BY statements to get the same result, but it would greatly increase the complexity of the query in question. The query can be accessed in the olap.sql file.

As it can be seen in the query, we selected the number of anomalies in each group of attributes by using the COUNT(*) function, which counts the number of rows (incidents) registered for each different combination of severity, locality and week_day. Thanks to the star schema, the complexity of this query was greatly reduced as we now only need two NATURAL JOINS (one for each accessed dimension table).

This type of queries facilitates a lot the analysis of the data present in a given data set because, by only using WHERE statements and attributing different values to each dimension, we can easily filter the set in order to obtain the desired information. We could also insert the result of the query into another table, in the order to simplify the querying process.