INFOB3DAR: Lab Exercise 1

Auke Roest (6922554)
Mikey Lam (1672320)

May 26, 2023

1 Introduction

For the course Data-analyse en retrieval (INFOB3DAR) and in the topic of Information Retrieval (IR) we were tasked to create a solution for two existing problems within a SQL database: a *many answers* problem and a *zero answers* problem. A way to solve these problems is to apply IR functions to the Boolean query model. In this case, a metadatabase containing information of the original database will be created so that it can be used to calculate a top-k result given a query within the model.

In this report, we will show how we designed our metadatabase, explain how the programs for the metadatabase functions, display our results from the programs and discuss our experiences of our approach.

2 Design Metadatabase

For the design of the metadatabase for autompg, we decided to split the metadatabase into two parts: the categorical metadata and the numerical metadata. The categorical metadata holds information of categorical attributes such as values, inverse document frequency (IDF), query frequency (QF) and a set of workload IDs. The numerical metadata holds the values, query frequency and bandwidth of numerical attributes. (2)

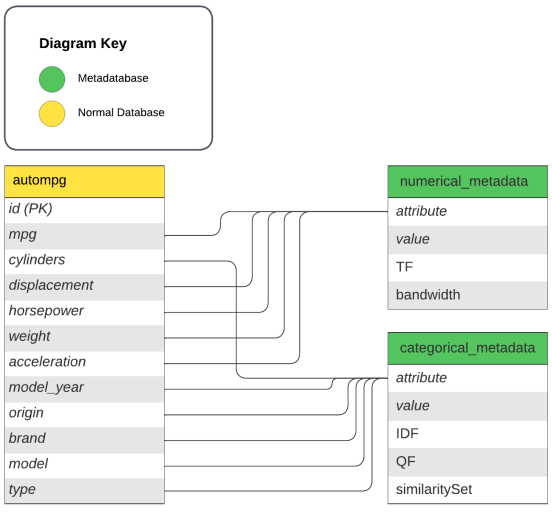

Figure 1: Proposed design for the metadatabase which consists of a numerical and a categorical metadata table

The decision to include these metabase attributes is to hasten the work of the top-k program, which needs these values to calculates scores of autompg values given a certain query. The formulas for these scores are derived from the article by Agrawal, Chaudhuri et al. [AJ13]

3 Programs

For the assignment, we created two C# programs: MetaDatabaseCreator, which will read the original database and create the metadatabase based around the information it will read, and QueryProcessor, which will read the metadatabase and give a top-k selection of results given a certain query.

3.1 MetaDatabaseCreator

The program MetaDatabaseCreator reads the data from autompg.sql and workload.txt and processes that data into metaload.txt for the creation of the metadatabase. In order to process this information, a class TableTuple has been created to store the term frequency, raw query frequency and set similarity of a value. For storing specific attribute values, dictionaries with ⟨**key, value**⟩ tuples have been used where the ⟨**key**⟩ is a value of an attribute and the ⟨**value**⟩ is a TableTuple object that stores the information. For the calculations of certain

First, the program creates a database with autompg.sql and reads all the information stored there, which in turn creates ⟨**key, value**⟩ tuples for the attribute values and counts the term frequency of said values. After reading the database information, the program reads the workload.txt file where it checks conjunctive equality queries, counts how many times a certain value has been queried and if that query occurs in an IN-clause. This information will be stored again in the TableTuple object of the attribute values.

When the program is done reading with reading the data and workload, it will start creating INSERT statements for metaload.txt. For each tuple in each dictionary, the program will calculate the corresponding values for the attribute value, such as IDF for categorical attributes (1) and QF (2). These INSERT statements will be then injected into metaload.txt.

$$IDF_k(t) = \log(n/F_k(t)) \tag{1}$$

$$QF(q) = RQF(q)/RQFMax \tag{2}$$

3.2 QueryProcessor

We used three classes for the program QuerProcessor Processor, Predicate and a static class Interface. Console I/O and predicate parsing is handled by the Interface class. You have to input queries in the format of attribute = value, separated by commas and ending with a semicolon. Values which are strings need to start and end with double quotation marks. White space between words is ignored. Few examples would be: "brand = "ford";" or "cylinders=4 , k = 10;". If the query is parsed to a list of predicates, then the Interface class will then call FindTopk() from Processor and print the values returned.

Predicate contains an enum queryType, which represents an attribute and a value as string. We could have chosen to make Predicate a generic type class, however we found that storing int's and floats as strings less cumbersome to work with. The class also contains methods to turn queryTypes to strings and back, since SQL queries are given as strings.

The processor class will create a list of all database entries and calculate the score for each entry. It will then return the top-k elements with the highest score. The score is calculated the same as we described in our planning::

For every attribute which is in the list of predicates the similarity coefficient (defined later) will be calculated. This value will be multiplied by 10.000, because it is more important than the attributes not in the query. Values which do not have a QF will be ignored.

For every attribute of the entry which is not in the list predicates we will calculate the root of the query frequency. This is mainly used to break ties if the similarity coefficients are the same. The total score will be the sum of the values mentioned.

For predicates of categorical attributes we will use QFIDF similarity. If the value of the entry matches the predicate it will have the QFIDF value, otherwise it will be 0. The QFIDF is calculated as the attribute's IDF (1) times QF (2) of the value.

For numerical attributes we calculate the similarity score using the formula seen in (4), where the numerical IDF is calculated as (3).

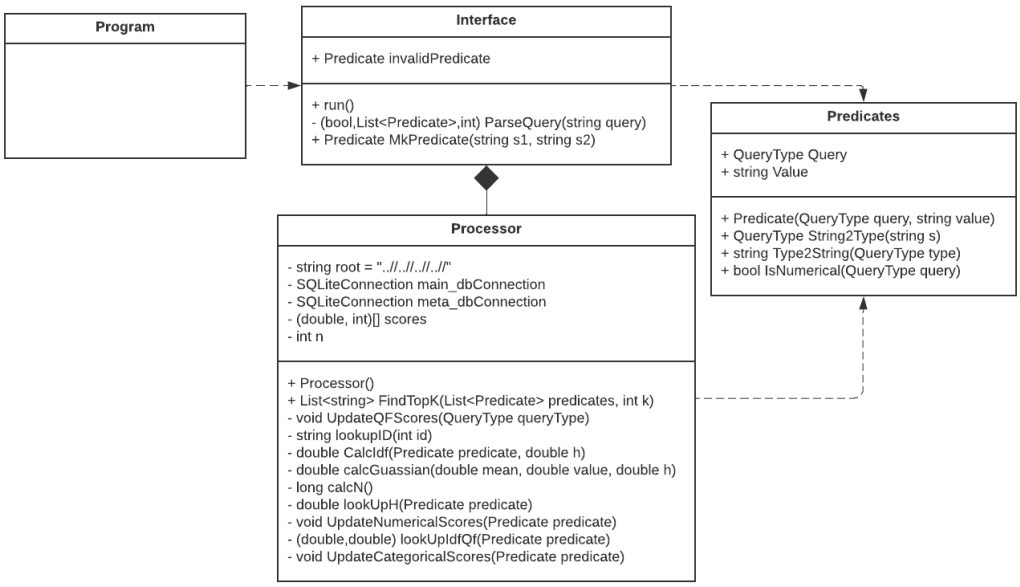

Figure 2: Class diagram of QueryProcessor

$$IDF(t) = \log(n/\sum_{i}^{n} e^{-\frac{1}{2}(\frac{t_i - t}{h})}) \tag{3}$$

$$S(t,q) = e^{-\frac{1}{2}(\frac{t-q}{h})} IDF(q) \tag{4}$$

4 Results

4.1 Many answers

With regard to the *many answers* problem, for every entry in the database we added the log of the query frequency to the score. Since this value is small in comparison to the score calculated from attributes, this means that in general a database entry matching more predicates will be ranked higher then a entry matching less attributes. However database entries matching the same predicates will be ranked based on other attributes they have. Thus if all entries match the predicates, only the most popular entries will get to the top-k.

4.2 Zero answers

In regard to the *zero answers* problem, each database entry will be given a score based on which and how many predicates it matched. This means that in order to get to the top-k, an entry does not need to match all predicates and in the case that none of the entries match any of the predicates, then the most popular ones will be put in the top-k as described in the previous section.

4.3 Value similarity

In order to solve the problem of value similarities for numeric attributes we have decided to make score given to a database entry for a given numeric predicate inversely proportional to the difference between the value of the predicate and the value of the entry, which means that database entries with values close to the value given in the attribute will be given more score and ranked higher. This is done via a Gaussian distribution, so score will be minimal for big differences.
Note that in our database we see integer values as categorical instead of numeric.

5 Discussion

With the creation of the metadatabase, we think our approach of implementing it seems sufficient enough, as it stores the relevant information that is needed to calculate the scores for the top-k results. A possible revision could be to split the structure of the numerical metadata table, as the bandwidth value repeats in the table.

Some improvements could have been made into finding the top-k results for a certain query, however we quickly discovered we that we were running out of time and unable to implement the proper improvements on the program. One such example is the use of the Jaccard coefficient where similar categorical values have a certain similarity score with each other. We wanted to include this implementation in QueryProcessor, however we ran out of time and we could not think of a proper way to program this function in.

Another example is the possible use of Threshold Algorithm (TA) and No Random Access algorithm (NRA) for the QueryProcesser. These algorithms could have provided a more sophisticated way of calculating the top-k results within the program, yet we also could not implement these due to time constraints . There is also the case that due to little amount of time, there might be a chance of bugs being present in the program which we have not uncovered yet.

In hindsight, certain aspects of this assignments could have been performed better if the project was made in time.

References

[AJ13] P. Ayyadurai and S. Jayanthi. Article: Automated ranking for web databases using k-means algorithm and uqdr approach. *IJCA Proceedings on International Conference on Research Trends in Computer Technologies 2013*, ICRTCT(4):9–12, February 2013. Full text available.