Program and solution description

In this document I show an overview of my program and the steps taken in order to solve the given problem. It is partitioned into three sections:

-A general overview: an overview of steps ive taken to solve the problem

-The preprocessing phase: An explanation of program 1, the tables created and the steps of the program

-The query phase: An explanation of program 2, the general steps needed for this phase, a general overview of the classes and their functional purposes and in more detail: the QueryProcessor class.

**General overview**

The general idea was to have tables filled with QFIDF scores for each attribute that is needed for the query. In the preprocessing phase we try to generate as much as possible QFIDF data. Then at querytime we join these tables with autompg ON a1 = q1, …, an = qn. Where a1,…,an are the attributes specified in the query and q1,…,qn are the values for these attributes also specified in this query. Then we can pick the tuples which have the highest the sum of QFIDF score over all specified attributes. This will be the first attempt to generate a top-k. From here on, when I mention the “QFIDF score of a tuple” I mean the sum of QFIDF scores over all attributes specified in the query for that specific tuple.

If there are many tuples we simply pick the top k tuples with the highest QFIDF score. And rank these by an additional ranking function. If there are zero or less than k tuples then we try to take several consecutive steps in which we change/weaken the searching constraints until we find enough tuples. The general iterations will be: change constraints -> check if there are enough tuples -> if yes then we return, if no then we change the constraints -> check if there are enough tuples etc. The idea of having these two programs is that the preprocessing program can be executed beforehand at any time to index the data, so that at querytime(program 2) this data is ready to be used for information retrieval. Program1.exe can be run from the commandline. Or from visual studio. This wil create the metadb file in the db folder and fill it with the necessary data. Then once its finished we can run the exe of program2 at any time, which also can be run from the commandline or visual studio. This will open a commandline from which we can do the queries. It is also to run a demo from this program. You just need to uncomment runDemo() and comment runProgram() in the main function and then run it.

**Preprocessing phase**

**The QFIDF formula used**

In order to determine the QFIDF similarity between a tuple and the query we use the function: SIM(T, Q) = Sumk=1 to m Sk(tk, qk). Where Sk(tk,qk) = Scoreqf(tk, qk)\*Scoreidf(tk, qk).

Where the sum k = 1 to m is simply the sum over all attributes specified in the query. For both categorical and numerical attributes we use Scoreqf(t, q) = (RQF(q) + 1)/(RQFmax(q) + 1) if t = q, else 0, where RQF(q) is the raw query frequency of q in the workload. Notice the + 1 in both the numerator and the denominator. We do this in order to get a close to zero value even if q is not mentioned in the workload. If we’re looking at a categorical attribute then Scoreidf(t, q) = log(n/F(q)) if t = q otherwise 0, where F(q) is the amount of tuples in autompg we for which a = q holds. If the attribute is numerical then we use formula (3) on page 4 of the paper.

**Which tables have been created**

Each categorical attribute has its own qfidf score table. To calculate the qfidf scores we used intermediate tables for qf score and for idf score. These are multiplied in order to get the qfidf score. The intermediate tables will be deleted at the end of the preprocessing phase. For the numerical attributes we create only the qf score. This will be explained later. Also we’ve created tables for each numerical attribute that holds the h/bandwidth value which will be used for calculating the idf(t,q) score at query time. Lastly we also have a table holding the amount of tuples in autompg, so we don’t have to constantly count during query time.

**Which tables haven’t been created**

Since we don’t know the query at preprocessing time we need to create idf data for numerical attributes at query time. I know there are options for creating data for numerical idf to be used later through histograms or interpolation. However because of limited time and the fact ive been working alone, ive opted for creating numerical idf data at query time. This is the reason why you don’t see any idf or qfidf tables for numerical attributes in the metadatabase yet at preprocessing time. So the idea is that at query time we create the numerical idf tables and multiply them with the corresponding qf tables (which will already be there) in order to get the qfidf data for that numerical attribute.

**Query Phase**

**Steps taken to solve the query problem**

First we parse the input and extract the attributes in question. For all numerical attributes: lets assume the attribute is called a and the corresponding query value is q. We first iterate over all distinct values for a in order to calculate IDF(q), then we use this value to create and fill the idf table. This table will contain a tuple for each distinct value of a. Then we use this table together with the qf table to create the qfidf table. Each tuple in this table contains the score qfidf(t, q). All qfidf data for categorical attributes is already created. So after doing this we only need to join all the qfidf tables with autompg and sort it, with the tuples who have the highest sum of qfidf scores on top and the lowest on the bottom.

**General overview of classes and their purposes**

Weve structed program 2 in to 5 classes, of which the class diagram is shown in the png file. Program class contains the main function and from there we invoke the needed functions/methods, do the necessary parsing and instantiations.

We have a class for a NumericalAttribute. We create an object for each numerical attribute specified in the query. This class is responsible for creating the numerical QFIDF data. We also have a class for categorical attributes which just contains the query value and its attributename. Next there is the query class which should represent the current query. It contains a list of numerical attributes and a list of categorical attributes. Each of which are specified in the query. This class is responsible for making sure every table that is needed is available in the metadatabase. It invokes the createQFIDFTable() function for each numerical attribute. Then the biggest class is the QueryProcessor class. An object of this class will be instantiated in the program class and we will pass a Query object to it for it to process.

**The QueryProcessor class and the control flow of the algorithm**

This is the class where the main part of the query processing happens. In this class we make sure the query object creates all the qfidf tables. Then the queryprocessor joins all these tables together with the autompg table. We will call this table allqfidf. From this big joined table we need to extract a top-k. This happens in the QueryProcessor.createTuplesWithQFIDFScoreTable() function. After invoking this function, the QueryProcessor.findTopK() function should be called. The following steps are taken in the findTopK() function. First we try to retrieve all tuples from the table allqfidf where a1 = q1, …, an = qn for all specified attributes in the query. Then for each tuple we take the sum QFIDF(t1, q1) + …. + QFIDF(tn, q1) and order the table by these values. The tuple with the highest sum will be on top. At first, If we find many tuples then that means we’ve found >= k tuples for which the whole ceq holds. All these tuples have the same score, so to me it made sense to do an additional ranking between these. My ranking weighs horsepower, acceleration, mpg and model\_year. I want to favour cars with good performance but also looks at how new and economical it is. The function is simply a sum horsepower\*a + acceleration\*b + mpg\*c + model\_year\*d. Ive first added factors for each term in order to weigh them all evenly and then I added extra weight to mpg and model\_year in order to favour environmental friendliness a bit more than performance. The reason for favouring newer cars is for lower maintenance costs and in general newer cars are more environmentally friendly. This function takes a sort of middle road between environment and performance. However its very easy to change this function because we have only one method in our class that encapsulates everything that is needed for this.

That was the case of many tuples. But the biggest part of the program is how to handle zero/too little tuples. Every time there are too little tuples we change/weaken the searching constraints and check if we got enough tuples. Then if needed we add additional ranking, return to print the tuples, or again weaken the constraints. This is the general control flow where each step is taken if there are still too little tuples:

-Instead of a = q for each numerical attribute, we add margins, so a = q becomes q – 4\*h <= a <= q + 4\*h, so this means we will retrieve all tuples for which the attribute value is 4 times the bandwidth value away from the query value.

-Next we try to find enough tuples by widening the margins even more to 8\*h.

-Next if brand is specified we try to search just by brand and numerical constraints with margins of 4\*h.

-If brand is not specified or if the previous step didn’t yield enough tuples we try to search by just type(if specified) and numerical constraints with margins of 4\*h.

-If type wasn’t specified or the previous step didn’t yield enough tuples we try to search just by brand, so this means the numerical constrains were probably not feasible and we need to search just by brand.

-Then we try to search just by type

-All previous steps have failed so now we just search from all cars in autompg the cars with the highest score of our personal function. This will always lead to enough tuples.

Note that a step may fail because k is too high. Lowering k might succeed earlier in finding a top-k. In all steps which involve numerical attributes we order the result by qfidf similarity.

Some notes:

-The preprocessing takes about 30 seconds to finish. Its not hanging, its calculating.  
-Program 2 sometimes hangs before even the first Console.WriteLine statement. I don’t know why this happens, I think maybe this happens when you close the console and the database connection is not properly closed. But if this happens just restart the program and everything should be fine.