Made by:
Olivia Lohmann: (8742779)
Wifried Snijders(8657078)

Our approach to the problems:

The main formula we wanted to implement was to process queries was Similarity(searchValue, resultValue) * QF(searchvalue) for each attribute. To do this we split the attributes into categorical and numerical, for categorical data we used the definition for similarity of (W(t) ∩ W(q)) / (W(t) ∪ W(q)). To speed up the queries we have preprocessed this data in our metaDb, to do this we first translate the workload.txt document into a more computer readable json document, then for every pair of categorical values we calculate their similarity and for every value pair and save that into our metaDB, in the tables named {attr}IDF. Then to weight the categorical values we also calculate the QF for every value of every attribute, this is saved into our {attr}QF tables in the metaDB, during querying we multiply these values to get a score for that attribute. For numerical data we use the definition for similarity described in the paper, as this cannot be preprocessed since the value of the query is not known at the time. However there is one thing we can preprocess, the standard deviation of every attribute, which we save into the stDev table. Then we didn't find a formula to calculate a QF for numerical data we use a constant, we choose 2 as this is around the average QFscore of our categorical data and we wanted numerical and categorical data to be of similar importance, with all this data and these formulas we use a topK algorithm to get the highest scoring values for a given query, after we get these we sort them and then take the first k values, this is to break ties randomly, as the tiebreaker of QF was already included in the score. We tried to match the algorithms described in the paper given in the literature, this informed our choses in terms of formulas used,

Due to our own mishaps in filling in basic queries we made ourselves a tiny saveguard for the input, while it will not filter out malicious attacks. It will wake you up if you made some small spelling mistake. We chose to leave these things as we wanted to spend most of our time on the algorithms and the requirements in the assignment didn't seem to indicate them as important.

Experiences:

We tried a lot of things to try and speed up query processing that didn't end up working, most notable we were trying to only use the top X values of each sorted attribute list, where X was defined as chosen number of different queries * k * 1.5 + 5, this ended up leading to bugs we couldn't figure out how to fix, so we ended up dropping this optimization. We started this project in week 2, but a lot of time was spend trying to figure out what we were expected to do as the assignment was very unclear, we hope we ended up getting everything we were supposed to into the project, but we had to make a lot of last minute changes to try and get everything we think we needed in. Also one of us had one week of experienced with python (We picked it as we read that for the second project we could also get help in python and R, and we wanted to use the same language for both projects) which lead to some coding style conflicts which made trying to understand each other's code more hassle than we expected.

Made by:
Olivia Lohmann: (8742779)
Wifried Snijders(8657078)

FunctionDescriptions:

- **HelperFunctions**
  - readFile(fileName)
    gets all lines that were in a txt file
  - GetSqliteInsertCode(fileName)
    reads a txt file and removes whitelines
  - createSQLITEDB(fileName)
    deletes the last sqliteDB and replaces it with a new db and gives you a connection to the new db
  - openSQLITEDB(fileName)
    opens a sqliteDB and returns a connection to execute queries
  - getResultsOfQuery(fileName)
    returns the result from a given query on a given DB
  - insertDataIntoQuery(fileName)
    modifies the db that was given with a query
- **FillMainDB**
  - main
    Executes sql queries from autompg.txt
- **fillMetaDB**
  - executeQuery
    Executes a query while logging it if it's a create or table, used to creare metadb.txt and metaload.txt
  - createWorkloadJson
    Transforms the workload.txt document into a more computer readable workload.json file
  - getAllUniqueVals
    Use to get all unique values an attribute can have
  - createIDFTables
    Calls createIDFTable for every categorical data attribute
  - createIDFTable
    Given an attribute creates a table with every pair of values and their IDF score
  - getIDFScore
    Calculated $(W(t) \cap W(q)) / (W(t) \cup W(q)$ for a given pair of values for an attribute
  - createQFTables
    Calls createQFTable for every categorical data attribute
  - createQFTable
    Given an attribute looks up how many cars in the autompg database match each attributes and stores those values in a table
  - createStDevTable
    For each numerical attribute stores the standard deviation of values of that attribute in the autompg database and stores this in a table

Made by:
Olivia Lohmann: (8742779)
Wifried Snijders(8657078)

- **processQuery**
  - transfromCEQ(line)
    Uses the input line to determine if k was given, splits the other query up to be used by dependentiesToTopK.

  - dependentiesToTopK
    Splits the dependenties up and returns them inside a 2dList.
  - checkIfValidQuery
    Checks if the input of the query was correct, if it can still solve it with lesser constraints it returns the values.
  - calcWeightedScore
    Calculates the weighted score of the current maxvalue possible new value inside topK.
  - topK
    does way too much for the function name:
    - It starts by requesting the topX of all the data it was given, numerical or categorical if it is categorical it also gets the QF weights required
    - It sets the topK up by having a result buffer and max values
    - It starts the TopK
    - It requests the values for the topK
    - It sorts the list of topK's and removes the lowest ones
  - getTopXCatagoricalData
    While it was first intended to get only a couple dataPoints we got unsuccessful query results from that(not getting k values), now we sort the entire list and check in topK what values we use.
    It gets a list of (ID's, scores) that have the best IDF with the asked query.
  - getTopXNumericalData
    While it was first intended to get only a couple dataPoints we got unsuccessful query results from that(not getting k values), it now returns a sorted list of distance to requested query of the data , it slightly favors higher numbers due to people not minding a free upgrade.
  - getIDValue
    Calculates the score of each ID that is assigned by topK to be next, and puts it inside the buffer (ID, score).
  - calculateQFNumerical
    Calculates the similarity score of two values using the formula e^(-0.5*(t-q)/h)^2) * IDF(q)
  - getNandH
    Calculates n and h for a given attribute, using h = 1.06 * stDiv(attr) * n^ -⅕, This function is cached as it will always return the same value with the same arguments so there's no need to call the db too much.

- calculateIDFNumerical

  Calculates the IDF of a query numerical data, using

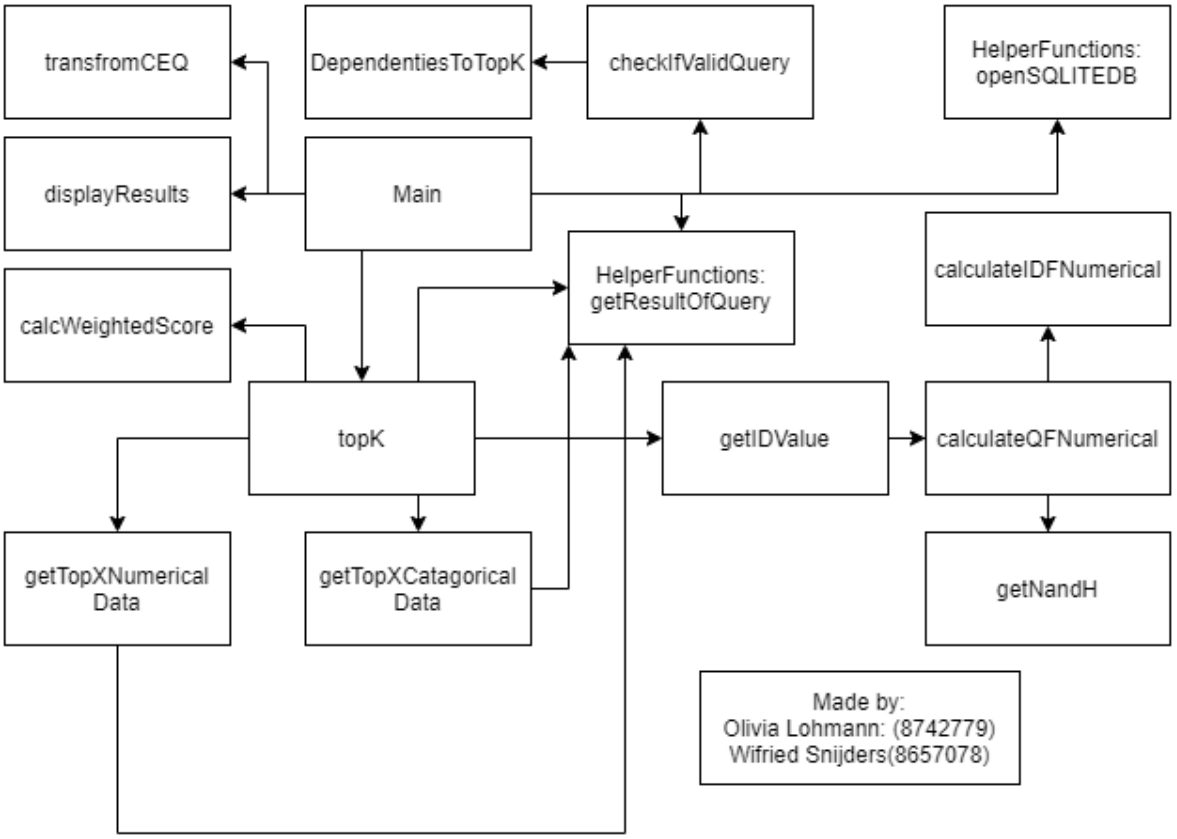
$$IDF\ (t) = \log\left(\frac{n}{\sum_{i}^{n} e^{-\frac{1}{2}\left(\frac{t_i - t}{h}\right)^2}}\right)$$

  This is also cached since it's the same value for the same q every time.

- displayResult(result)

  Displays the result in a nice table, and a * in front of the answer if it was a exact match uses the ID's given by topK, to get all the final values

We did not use classes so instead of a class diagram we added a function diagram to still give a overview of the flow of code used



Usage Guide

To get started:

1. Run fillMainDb.py
2. Run fillMetaDb.py

To run a query once that's done:

1. Run processQuery.py