Course: Complex Data Management

Project 3: Transaction Queries

Name: Konstantinos Vardakas

Student ID: 522

Email: pcs0522@uoi.gr

Part 1: Containment Queries

In the first part of this project, various methodologies for evaluating containment queries were implemented and evaluated. The relevant code has been organized into two files, the containment_functions.py file, which contains the helper functions, and the containment_main.py file, which contains the main program. Initially, the files transactions.txt and queries.txt are read using the function read_file(filename), which returns a list of sets. Each set contains integer values corresponding to the IDs of objects found either in a given transaction or in a given query.

Naive method

The first containment query method implemented was a simple reference method, naive_method(), to compare performance with the others. In this function, for each transaction, it is checked whether the query is a subset of the transaction objects. If it is, the transaction ID is added to the list of results.

Exact signature file

The second method implemented is based on the use of bitmaps (signature files). For each transaction, a bitvector (integer) is created, which is initialized with the value 0. Then, for each object contained in the transaction, the corresponding bit of the vector is activated using bitwise operators. The bits are numbered from right to left (i.e., from the least significant bit to the most significant bit) by applying the shift operator (1 <<

item). At the same time, the OR operator is used to retain all bits that have been activated in previous iterations of the loop.

Next, the to_sigfile(transactions) function uses the bitmap function for all transactions, creating the signature file, which is saved using the save_file(list_to_save, save_name) function. The containment query is performed with the signature_file(query_bin, transactions_bin) function, where for each transaction, it is checked whether the bits that are activated in the query are also active in the corresponding bitvector of the transaction. This check is performed with the bitwise AND between the query and the transaction, and comparison of the AND result with the original query.

Exact bitslice signature file

The third method implemented is the bitslice signature file, where instead of creating a bitvector for each transaction, one is created for each object. Each such bitmap has positions equal to the number of transactions and has a value of 1 in the positions corresponding to transactions where this object exists. The creation of the bitslice structure is implemented with the bitslice(transactions) function, which initializes a list of zeros, which are the initial bitvectors of each object. For each object, the bit of the transactions that contain it is activated, in the same way as previously in the bitmap function, using the shift operator (1 << item) to insert the unit from the right, and the OR operator to keep the bits that were inserted in the previous iteration of the loop. This creates a list where each position corresponds to an object and contains an integer representing the bitvector (bitslice) of the object. This list is saved to the bitslice.txt file using the save_enu_file function, with each line containing the object ID and the corresponding integer bitvector that describes it.

Next, to evaluate containment queries, the bitsliced_signature_file(query, bitslice_trans) function was used, which accepts the output of the previous function, i.e., the bitmaps of the objects and the query, and performs the logical AND between the objects of the query. This is done by comparing the bitvector of the first object with the rest of the query objects, and the final result has a unit at the positions of the

transactions that contain all the query objects. Finally, these transaction positions are exported to a list which is returned by the function.

Inverted file

The fourth and final method is based on the creation of an inverted file. For each object that appears in transactions, a list is created containing the IDs of the transactions in which that object participates. These lists are sorted and stored in a dictionary, where the key is the object ID and the value is the corresponding list of transactions. This structure is created with the build_inverted_file() function.

To evaluate a containment query, the inverted_file(query, inverted_index) function is used, which adds the transaction lists from the inverted index dictionary created earlier for each query object. These lists are collected and then their intersection is calculated using the merge algorithm for sorted lists with the merge_intersection function. In the merge_intersection function, the first two lists corresponding to the transactions involving the first two objects are initially compared linearly (since they are sorted), similar to the sorted intersection algorithm. Then, the common transactions remain in the result list that was initially the first object and are compared with the third object. This process is repeated for each object and, similar to the previous method, the transactions that are common to each object are retained. Finally, the results are saved using the save_inverted_file function, with the dictionary keys sorted.

Main Program

The main program uses the time_method function to format and time the results of each methodology. Timing is performed using the perf_counter() function instead of time() due to greater accuracy, and in each case, timing refers to the time required to extract the query results, not the preprocessing that needs to be performed on the data (e.g. exporting sigfile, bitslice, and invfile files). In any case, these files are first created and stored, and the query is performed using the methodology selected based on the value of the method argument. The query is also performed using the map function, as

each query function implemented refers to a single query. Thus, if there is only one query, it is converted into a list so that the results can be extracted using map.

Part 2: Relevance Queries

In the second part of the thesis, methods for evaluating relevance queries were implemented, taking into account the frequency of occurrence of items in each transaction (bag semantics) and the rarity of items in the set of transactions. The code was organized similarly, in the containment_functions.py file, which includes the auxiliary functions, and in the containment_main.py file, which includes the main program. The transactions.txt file is read using the read_file(filename) function, which returns a list of lists, preserving the multiple occurrences of items.

Creating an Inverted File with Occurrences

In order to implement relevance queries, a basic requirement is the construction of a modified inverted file structure that takes into account the number of occurrences of each object in each transaction, as well as the construction of a structure that maintains the relative rarity of each object in the set of transactions. The build_inverted_file_occ function is responsible for creating these two structures in dictionary form, where they are initialized as empty dictionaries.

The process begins with each transaction being processed separately. For each transaction, a temporary frequency dictionary, occ_dict, is initially maintained, which records how many times each object appears in that transaction. Once occ_dict is calculated, its contents are added to the inverted_index dictionary. Specifically, for each item in occ_dict, a pair of the form $(\tau, occ(i,\tau))$ is added to inverted_index, where τ is the transaction ID and $occ(i,\tau)$ is the number of occurrences of item i in that transaction.

At the same time, for each object that appears in this transaction, only one occurrence is added to the trf dictionary, so that the total number of different transactions in which each object participates can be counted. It is important to note

that the trf is updated based on the occ_dict, i.e. each object is counted only once per transaction, regardless of how many times it appears within it.

Once all transactions have been processed and the two dictionaries (inverted_index and trf) have been updated, the final value of the relative scarcity of each item is calculated using the idf dictionary. For each item, the formula |T| / trf(i,T) is applied, where |T| is the number of transactions. The final result of the function is the inverted_index, which contains the relevant transactions and occurrences for each item, and the idf, which expresses how rare each item is based on its participation in the total number of transactions.

Inverted file

It is now possible to implement the relevance query evaluation process using the two dictionaries that were created. For a given query, the corresponding lists are searched for each object in the inverted_index dictionary. Each such list, as mentioned, contains pairs of the form [transaction_id, occ(i, τ)]. To calculate the union of transactions that include at least one of the query objects, the merge_union function is used. This function applies a merge algorithm to sorted lists of pairs [tid, occ]. During processing, a pointer is maintained for each list and the current transaction_id pointed to by each pointer is compared. At each step, the smallest tid among all pointers is identified and all occurrences (occ) of the items for that tid are collected from the lists containing that tid. If an item does not appear in the tid, 0 is added in its place. This produces a list of pairs of the form [tid, occ_list], where occ_list is a list of the frequency of occurrence of each object in tid.

Next, in the inverted_file_method function, for each tid resulting from the union, the relevance function rel(τ, q) is calculated, which is defined as:

$$rel(\tau, q) = \sum_{i \in q} \left(occ(i, \tau) \cdot \frac{|T|}{\text{trf}(i, T)} \right)$$

That is, for each object i of the query that appears in transaction t, the product of its number of occurrences in the transaction (occ(i, t)) and its rarity in the set of transactions (which has been calculated in advance as idf[i] = |T| / trf(i,T)). The total

rel is added to the results only if there is some relevance (rel > 0), and is recorded together with the transaction_id in the results. Finally, the results are sorted in descending order of relevance and the k most relevant transaction_ids are returned, or all of them if there are fewer than k.

Naive method

The naive_relevance_query function implements a simple approach to evaluating the relevance of a query to the set of transactions, without using the inverted_index dictionary, but by examining one by one all the transactions contained in the transactions list.

At the beginning of the function, an empty dictionary relevance_scores is created, which is used to store the relevance of each transaction containing at least one object from the query. For each transaction in the transactions list, a dictionary occ_dict is constructed, which records how many times each item appears in that transaction, similar to the occ_dict of the build_inverted_file_occ function. Once the number of occurrences of the items in the transaction has been calculated, the calculation of the relevance function rel(τ, q) for the specific query begins. The variable rel is initialized to zero, and for each item in the query, it is checked whether it exists in occ_dict and also whether it exists in the idf dictionary. If both conditions are true, then the product occ_dict[item] * idf[item] is added to the score.

If the relevance for the transaction is not zero, it is entered in the relevance_scores dictionary with the transaction tid as the key and the relevance as the value. Once the relevance for all transactions has been calculated, the results are converted into a list of [score, tid] pairs and sorted in descending order of relevance. Finally, the function returns the k transactions with the highest relevance or all of them if there are fewer than k.

Main Program

Similar to the first part, the time_method method is used, the query is performed with the map function, so if the query is specific, it is first converted to a list. In any case, the invfileocc.txt file is first created and saved, and the query is performed using the methodology selected based on the value of the argument method.

Instructions for execution

Execution of the first part:

python containment_main.py <transactions file> <queries file> <qnum> <method>

Execution of the second part::

python relevance_main.py <transactions file> <queries file> <qnum> <method> <k>

Transactions and queries files can be given either with or without the .txt extension.