PROJECT REPORT:
SOFTWARE DEVELOPMENT FOR INFORMATION SYSTEMS

Team:

- Panagiotis Kokkinakos
- Giorgos Rouvalis
- Thodoris Stefou

The whole work was done using C language, and for compilation gcc with -O2 optimization flag.
The following are also used: gitlab for version control, Unity for test drive and valgrind for memory leakage checks. All results have been checked with the diff command and haven’t showed any differences.

Part 1

Our first approach was for each node to have a table of pointers on child nodes and when inserting or deleting a table element we made copies with assignment. Also, for insertion and deletion of data from tables we used linear search.

At this stage, the program ran for about 1 minute and 30 seconds. From the very first part we changed the above with a table of nodes-children instead of pointers as well as with the use of the memmove function instead of assignment to take advantage of the locality of memory. These two changes saved from our program 30 seconds setting the execution now just below 1 minute. Right afterwards, we compiled with flag -O2, saving another 25 seconds. Finally, we changed the linear binary search where we spotted a big difference: the program was running now in just over 1 second.

In general, on the first part we focused mainly on getting right results, so we did not implement additional structures to improve the run time.

Part 2

On the second part BloomFilter and Linear Hash Table were added for speed improvements as well as the concept of static trie to reduce the space it occupies. Top K has also been added to find the K-most popular n-grams in each burst.

In particular, BloomFilter has a table of cells bytes (char) instead of int so we don’t waste 4x space if the values are boolean type. During the "Cleaning" of the tree, instead of assignment we used the function memset.

For BloomFilter hashing the MurmurHash2 function was used and we took advantage of the fact that it hashes in different positions according to the seed which takes as an argument, so we had as many hash functions as we needed. Depending on the size of the tree initialization file, we found different values for table size and the number of functions, so that the results with the specific input do not make any mistake.

Specifically:
For the small dataset: 9280 cells and 4 functions
For the medium dataset: 57500 cells and 5 functions

We preferred to have more cells instead of functions so we keep the program speed low. Originally, because we had a lot of functions instead of many cells, the structure of BloomFilter slowed down the program. When we changed it, we saw an average difference in the medium dataset minus 4 seconds and for the small dataset minus 0.3 seconds. Each trie used 1 Bloom Filter.

We also added the new type of static tree for which we used different types of nodes and a different Q query function. Due to the fact that the type of tree was not known during the compilation, we used void * for the pointer to the tree and function pointers so we achieve a small model of object-oriented programming. While reading the first line of the file, we assign to our function pointers the functions that we will use during the remainder program. To achieve this, the functions of the trees now get void * to be the same and to declare the corresponding function pointers correctly and inside the body are casting the tree into what it will eventually use. For the static part, we also added some dummy functions for “add” and “delete”, so if there is a mistake in a file, for example, if a query on a static tree does not use NULL function pointer and breaks down the program, then nothing will happen.

In addition to the above, a Linear Hashtable was added to the first level of both trees which improved the runtime of the small dataset by half second. A design choice we made to avoid the consecutive splits when a bucket overflowed, was that the new bucket

which was created in each overflow, would have the same size as the bucket that would be split. In the remaining dataset we saw an improvement of 12 seconds on average.

Finally, Top k was added, which although it slowed a bit the program, something which is perfectly normal as it added an extra function. So change was not particularly important. For medium and large datasets the time increased from around 1.5 to 2 seconds, while at the small no change was observed.

For the static tree, we gathered the following statistics before and after compression:

Small dataset:
Before compression: 997,230 bytes 2,358 nodes
After compression: 841,529 bytes 1,981 nodes
Compression time: 0.013s

Medium dataset:
Before compressing: 18,950,786 bytes 44,376 nodes
After compressing: 15,828,919 bytes 36,817 nodes
Compression time: 0.059s

Large dataset:
Before compressing: 1,292,915,312 bytes 3,007,804 nodes
After compressing: 511,564,285 bytes 1,122,525 nodes
Compression time: 0.539s

Reading the above we see that compression has an impact as the size of input grows. Specifically:

Dataset/%Improvement	Bytes	Nodes
Small	15.6%	15.9%
Medium	16.4%	17%
Large	60.4%	62.6%

It is also noted that the time of compression is minimal in relation to saving space.

At the end of the second part, we gathered the following time statistics on Intel Core i5 CPU 750 2.67GHz × 4.

To have greater objectivity in measurements, static trees were measured compressed as well as not compressed.

Small static compressed: 0.041 sec
Small static not compressed: 0.062 sec
Small dynamic: 0 sec

Medium input compressed: 23.192 sec
Medium input not compressed: 19.746 sec
Medium dynamic: 27.756 sec

We observe that compression causes a slight increase in time, not only due to compression but also because in compressed trees the comparisons and conditions calculated for question queries increase.

Dynamic trees are slower in execution because they perform addition and deletion.

The following measurements show the time difference between 1st and the 2nd part for the small and medium datasets (with dynamic trie since there was no static in part 1).

1st part small: 1.059 sec

2nd part small: 0.582 sec

1st part medium: 63.042 sec

2nd part medium: 27.756 sec

It seems that the addition of BloomFilter and Linear Hashtable structures has improved noticeably the execution time.

Part 3

In the third part, we initially converted the table which consisted of bytes of BloomFilter to bit vector. Now each position is a bit that we can access through special functions. Doing this we reduced the space occupied by BloomFilter to the 1/8 of the previous one.

We also changed the way the results of each burst are collected. Now the Queues with the results returned from each question are not stored in a Queue, but in a table for a reason that is documented below.

We have set up a general purpose job scheduler that creates a number of threads that wait in a condition variable until the scheduler can mark the beginning of their operation. For every question, the job is added to the queue of the scheduler and when F is found and the burst is finished he sends a signal to the threads so they start working. Threads wake one another until there are no other jobs in the queue. The main thread wait for all jobs to finish before it prints results and find Top K n-grams. When we find an import or a deletion of n-grams (in dynamic tris), these are done by the main thread on-the-spot, as there is no paralleling for them. Every question in each burst has a unique id, which starts at 0 and increases linearly, and writes its result in the corresponding position of the result table. In this way, we are not interested in which sequence they have finished the questions, since during the printing of the results we print linearly the contents of the table with them.

Static trees were paralleled much more easily than dynamics because
there are no insertions and deletions in their work file.

For dynamic trees, we implemented versioning, in which in every query we assigned an id to know when a n-gram was added and when was deleted, so we do not print out any wrong results because all the questions of each burst are executed after all the import and delete commands. Each node of the tree has two new variables that express when this node was added and deleted. If the variable that shows when it was deleted exists, then it gets deleted too. We are only interested in these variables in the final nodes. Every Question includes in its result any n-grams added earlier than when the question was performed, that is, whichever has an id of input less than the id of the question, and at the same time when they have an id of -1 or greater than the id of the question.

Now, when a deletion command is displayed, n-gram is not actually deleted directly, but we change the variable that expresses the time to which was deleted. This happens like that so that versioning works smoothly. When we find an F command, then we clean and prepare the trie for the next burst, that is, we delete as many n-grams that have to be deleted, and initialize again the insertion and deletion variables to 0 and -1 respectively. Also, when we find F, numbering of ids of queries but also those of questions begins again from 0 for the next burst.

Now, due to the paralleling of questions, we can not only have 1 Bloom Filter for the entire trie, since it would be used at the same time from n to number of questions, where n is the number of threads. To deal with this, each thread has its own Bloom Filter, and if the job that has been assigned is a question then it passes it as an argument.

A bug we encountered was caused by the use of strtok in the questions when running two or more threads at the same time. With a little research we understood that strtok is not thread safe and so we have replaced it with strtok_r.

Summary of runtime measurements for the 3rd part

Intel Core i5 CPU 750 2.67GHz × 4

Dynamic Tries:

Dataset/# threads	1 thread	2 threads	3 threads	4 threads	5 threads
Small	0.684s	0.430s	0.314s	0.248s	0.256s
Medium	22.992s	14.048s	11.403s	9.579s	9.741s
Large	69.259s	52.022s	48.626s	46.210s	46.777s

Static Tries:

Dataset/# threads	1 thread	2 threads	3 threads	4 threads	5 threads
Small	0.575s	0.371s	0.276s	0.215s	0.234s
Medium	25.965s	12.829s	10.031s	9.975s	10.037s
Large	30.463s	18.054s	14.414s	12.613s	12.990s

AMD Phenom 965 CPU 3.4GHz × 4

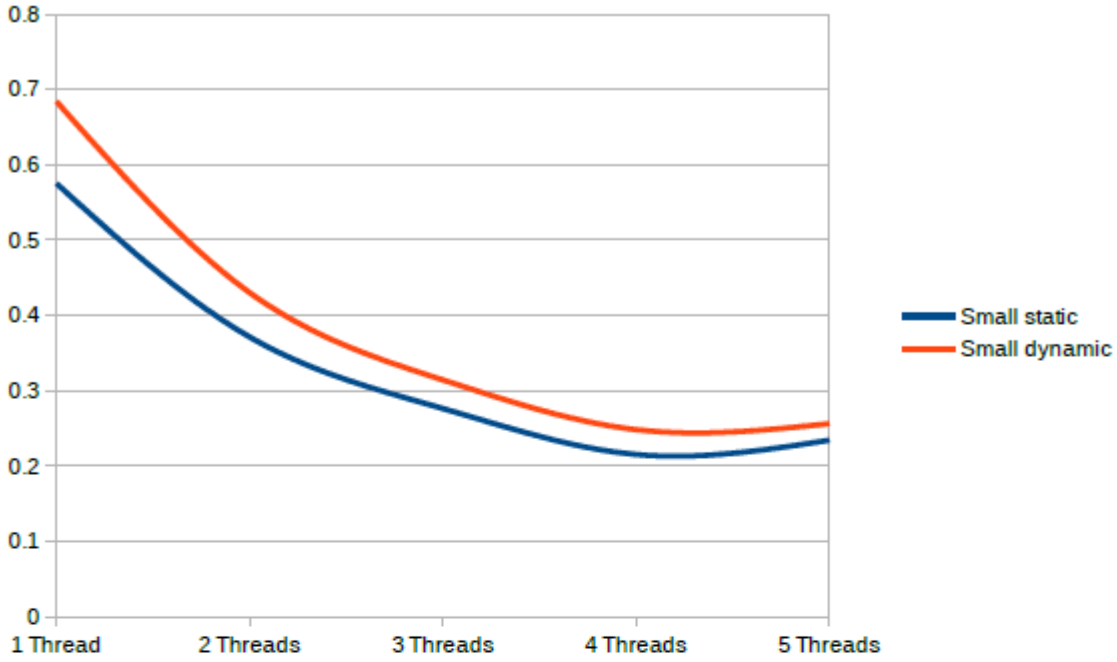
Dynamic Tries:

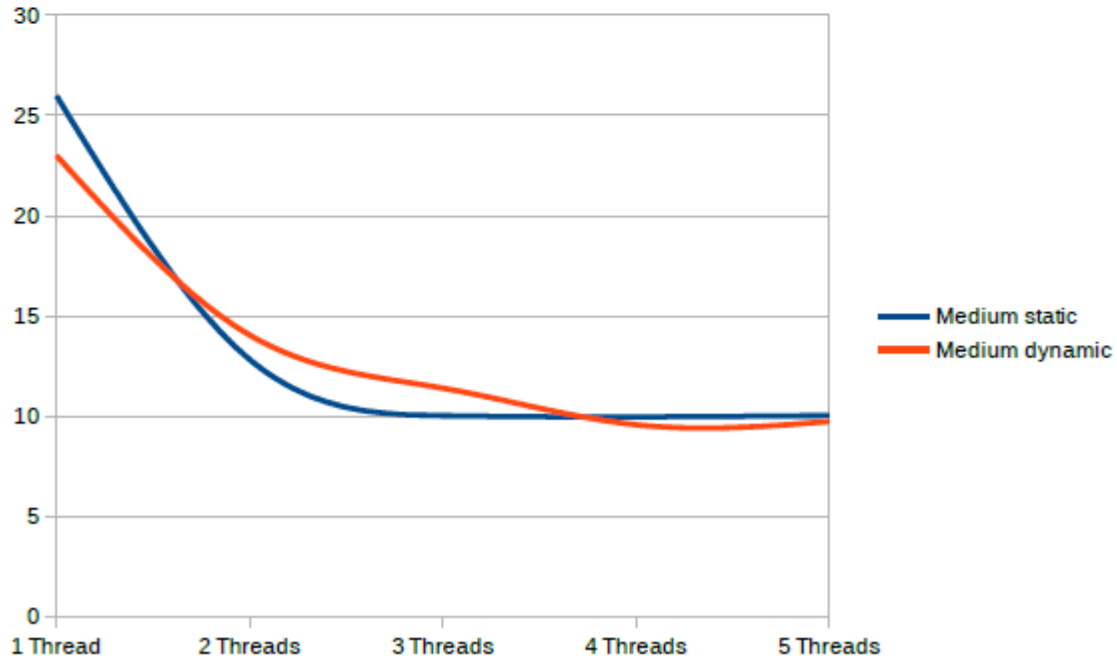
Dataset/# threads	1 thread	2 threads	3 threads	4 threads	5 threads
Small	0.666s	0.388s	0.316s	0.243s	0.246s
Medium	24.972s	15.306s	11.833s	10.180s	10.226s
Large	80.794s	67.290s	59.810s	57.200s	58.270s

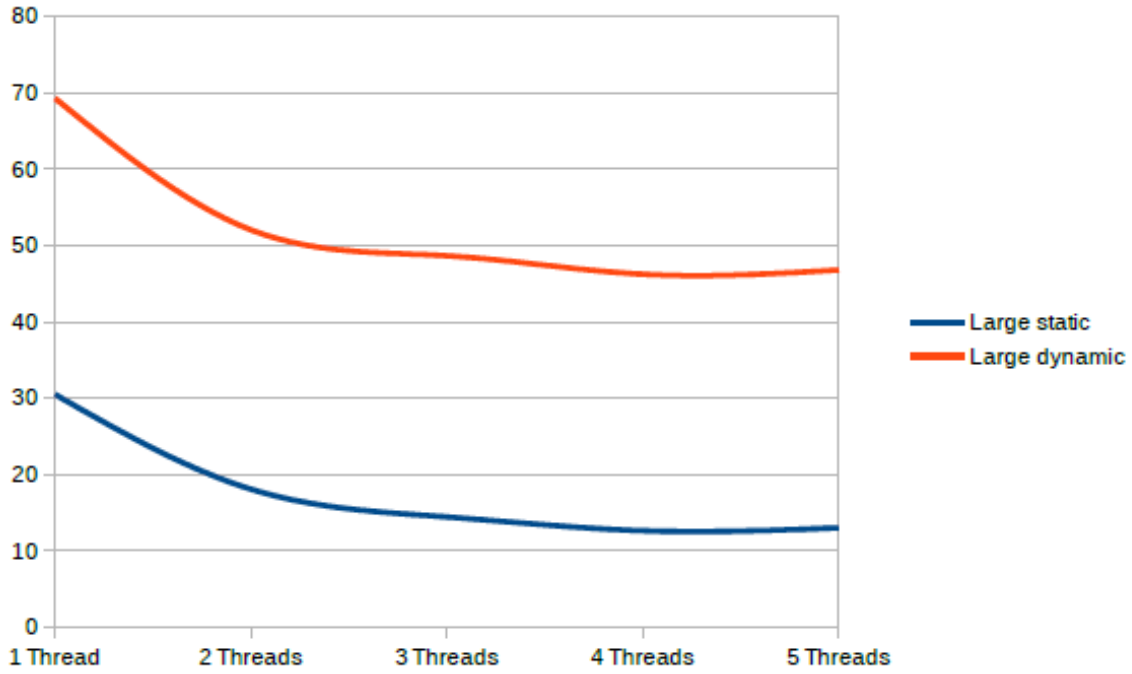
Static Tries:

Dataset/# threads	1 thread	2 threads	3 threads	4 threads	5 threads
Small	0.647s	0.374s	0.293s	0.216s	0.217s
Medium	26.854s	13.604s	10.241s	8.408s	8.453s
Large	41.610s	22.236s	17.521s	14.823s	14.899s

Below are some graphs of time in relation to the number of threads. The measurements come from the intel processor that was used.







General Conclusions:

We noticed that for the same tree if we compress it the execution time increases, but the number of nodes and bytes used is significantly reduced something which is desirable.

With the introduction of threads, execution times are significantly reduced. As for the number of threads, the 4-core processors that we ran our program, time is reduced for every extra thread up to 4. From there and beyond it remains constant and may be slightly increased due to cost of communication, as shown in the charts.

On average, dynamic trees have a longer run time than static trees for the corresponding files, which is, however, expected, since in work files of dynamics trees, there are insert and delete commands which are time-consuming.