Project: Time series indexing
Laboratory on Algorithms for Big Data

Italo Guerrieri

January 28, 2017

About the project. The goal of the project is to design and to implement a data structure to efficiently answer queries on page view statistics files for Wikipedia. These statistics are (hourly per page aggregates of) the number of requests for the pages of Wikipedia.

The goal is to build a data structure that indexes a file of triples to efficiently answer the following two queries.

- $Range(page, time_1, time_2)$ that returns all the counters of page in the time interval $[time_1, time_2]$. The counters have to be reported sorted by time.

- $TopKRange(page, time_1, time_2, K)$ that returns the K highest counters for page in the time interval $[time_1, time_2]$. The counters have to be reported together with their times.

Baseline (baseline.hpp). All the implementations use a map structure that, given a date, return an identifier of that date. This map is built before building the index. The index is a map that given the name of a page, returns a vector of pairs. The pairs are composed by the identifier of the date and the counter of that date.

The function *Range* is done getting the vector of pairs and using a binary search to find the needed date in the vector. From this date the function returns all the counters it encounters, until it arrives to the date that is specified as the last one of the range.

The function $TopKRange$ do the same thing but instead of returning all the values it encounters, it uses a min heap to get the top k elements.

This implementation builds and index that has a size of 130MB and this operation is done in 68,4 seconds. Profiling this baseline it can be noticed that the most of the time, in building the index, is spent sorting the vector of pairs.

Implementation1 (baseline1.hpp). This version has a map that given a name of a page returns a vector of counters. All these vectors are initialized with 0 values and they are set with a length equal to number of unique date presents in the dataset. Thanks to that, this version does not need to sort anything.

At query time it is no longer needed to do a binary search because the first date has the identifier equals to the index of counter needed; And this, causes a very little increments of the performances.

This version builds and index of size 81MB and in only 42,2 seconds. It is a huge improvement in comparison to the baseline in terms of memory occupancy, because the identifiers of the dates are no longer stored in the index. Furthermore, the building time is decreased because, by removing the sort operation, we have decreased the number of operation that has to be done, as we can see in Figures 1 and 2. Another important information that we can get from these figures, is that there are a lot of L1 cache misses in the baseline implementation, about one order of magnitude in plus. For all these reasons the implementation 1 has been preferred in comparison to the baseline.

458.022.968.264	instructions:u	
132.295.274.806	L1-dcache-loads	
6.036.229.100	L1-dcache-loads-misses	4,56%
370.115.487	cache-references	
74.172.515	cache-misses	20,040 %

Figure 1: Profiling's results of the building of the index for the Baseline

266.129.827.931	instructions:u	
83.892.982.410	L1-dcache-loads	
565.036.249	L1-dcache-loads-misses	0,67%
208.519.239	cache-references	
72.629.047	cache-misses	35,790 %

Figure 2: Profiling's results of the building of the index for the Implementation1

Implementation2 (baseline2.hpp). This version, instead of using a single vector, as the implementation 1, uses a bit vector to understand if a date has a counter different from 0, referred to that page. All the counter that differs from zero are stored in a separated vector sorted by their date identifier.

To answer the queries, this version uses the rank function on the bit vector to know which counter to access.

This version produces an index that has a size of 59MB, which is an improvement of 27% of space occupancy, in respect of the implementation 1. This index is build in 56,4 seconds.

In comparison to the implementation 1, the time spent on answering the queries is greater, about four times. This is due to the use of the rank function on the bit vector. Indeed, profiling this version on the query phase, it can be noticed than the rank function is in the 15% of the execution of the queries.

Despite these results, it is better to choose the implementation 2 because it uses less space (27%). In big data terms, less space implies faster and cheaper algorithms due to the memory hierarchy.

Implementation3 (baseline3.hpp). This version is equal to the implementation 2, but it compresses the vector of counter with Elias Fano. With this compression we got an index of size 25MB instead of 59MB as for the last version, 57% less. This compression is paid in the query time in average is increased from 7e-05 seconds to 3e-04 second, that is more than one order of magnitudes. Indeed, a lot of the query time is spent using the select function needed to decompress the Elias Fano compression. Both of these implementations load the entire index on the principal memory. Although, with this compression we have an index that occupies less than half than the index of the implementation 2, and as it is said in the last section, less space is very important when there are big data. Indeed, in figures 3 and 4 it is showed that there are less cache misses and l1 cache misses when the queries are made with the implementation 3.

12.456.325.739	instructions:u	
3.607.082.540	L1-dcache-loads	
51.428.074	L1-dcache-loads-misses	1,43%
39.755.749	cache-references	
7.929.629	cache-misses	19,946 %

Figure 3: Profiling's results of answering the queries for the Implementation 2

18.808.608.035	instructions:u	
3.843.783.369	L1-dcache-loads	
41.406.544	L1-dcache-loads-misses	1,08%
56.481.737	cache-references	
3.627.661	cache-misses	6,423 %

Figure 4: Profiling's results of answering the queries for the Implementation 3

Final version (baseline4.hpp). In this version it has been added to the index a succinct RMQ for each page in the dataset. This has grown the size of the index to 31MB but has decreased in average the time of answering to the $TopKRange$ range queries to 3e-05, an order of magnitude less.

This implementation is preferred if it is not needed to have a smallest index but it is needed to have a good time to respond to the $TopKRange$ query. In figure 5 it is showed that this solution, in comparison to the implementation 3 (Figure 4, has similar number of cache misses and l1 cache misses, when doing

the query phase, but the number of instruction are less because of the succinct RMQ.

The only part of the index that can be improved is the one relative to the map table. Although, profiling the application it is highlighted that the map table has a very low impact in the computation of the application. For this reason this implementation is also the final one of this project.

10.522.814.559	instructions:u	
2.201.647.101	L1-dcache-loads	
43.366.071	L1-dcache-loads-misses	1,08%
48.696.027	cache-references	
3.522.237	cache-misses	6,423 %

Figure 5: Profiling's results of answering the queries for the Final version

Query Randomization. The set of queries used to make the test on the various implementations are 10000, for each type of query (*Range*, *TopKRange*). These queries have been randomized following these rules:

- The name of the page to be queried is chosen at random.
- Three type of ranges can be randomized: small, medium and big ranges. Where the small ranges have a range of 500 dates, the medium 2500 and the big 5000.
- The k value for the *TopKRange* is randomized between these three value: 5, 10 or 20.

Final comparisons. In this section it will be reported, in tables and plots, the scores of all the implementation. In the following table it is showed the performance of each index in terms of space occupancy and in terms of time of building these indexes. These table is also reported as two plots in figures 6 and 7

Baseline	**Index Memory**	**Index Time**
0	130MB	68,4s
1	81MB	42,2s
2	59MB	56,4s
3	25MB	59,0s
4	31MB	60,4s

The plot in Figure 6 shows that from one implementation to another one we have decreased the size of the index, exept for the last implementation where it has been added the succinct RMQ structure.

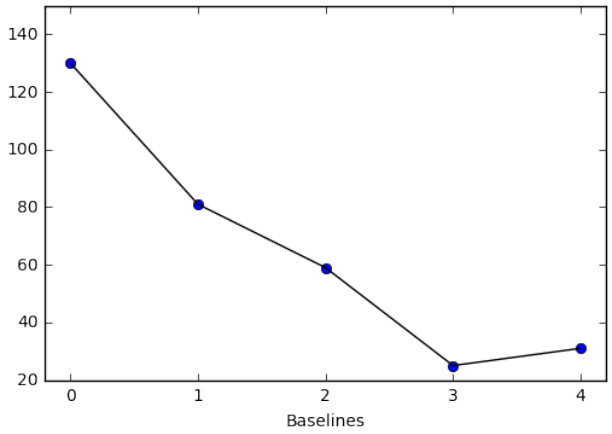

Figure 6: Size of the index for each implementation

In Figure 7 it is showed that the time to build the various indexes is decreased from the baseline to the implementation 1. Instead, from the implementation 1 until the last one the time to build the indexes is increased.

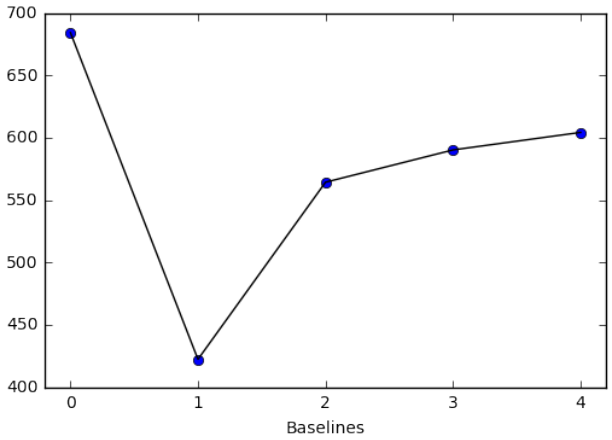

Figure 7: Time to build the index for each implementation

The following table shows the time of execution of the queries in the average of the various implementation. It showed that until the version 3 we have a similar time between the *Range* query and the *TopKRange* query. In the last version, the average time of execution of the *TopKRange* queries decreases because we have used the RMQ structure. All these values are represented in Figure 8.

From these plots, without considering the baseline, it is highlighted that each time it has

been compressed the index, the query time has grown.

Baseline	Range Time	TopK Time
0	2,17e-05	2,38e-05
1	1,93e-05	1,70e-05
2	7,72e-05	7,95e-05
3	3,10e-04	3,08e-04
4	3,07e-04	3,39e-05

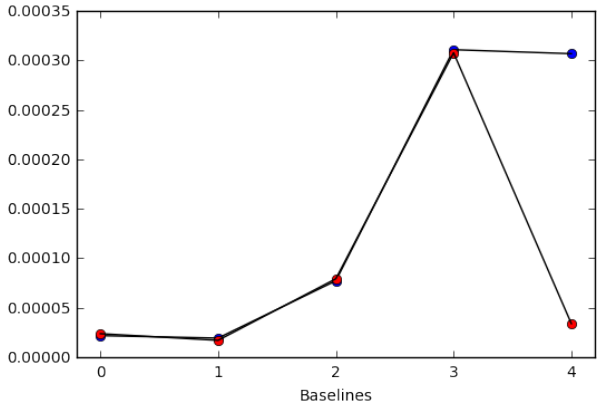

Figure 8: Time to answer the queries for each implementation. In blu the *Range* queries and in red the $TopKRange$ queries