Times Series Indexing Project Report

Federico Umani

2017-31-01

Contents

1 Synopsis

The goal of this project is to provide an efficient data structure to answer queries on Wikipedia page view statistics. The program take in input a preprocessed version of the Wikipedia page view dataset, and builds an index which is used to perform two types of queries on the dataset: a range query and a top k query.

The project consists in two programs, one reading the dataset, building the index and serializing the data structure, which is called Builder; another one deserializes the data structure, reads the query set and answers the queries, printing the results. This last one is called Interrogator.

There is also another executable, called Queryset_generator, capable of generating a random set of range and top k queries to test the project. Different versions of these three programs have been developed as different classes called by the same main program, which is called Builder_Main for the Builder program and Interrogator_Main for the Interrogator program. The two Main programs take a version number among the other parameters and instantiate the correct class accordingly.

The first five versions, from 1 to 5, try to index the data by date. We will show that even the most optimized version, the fifth, is not even comparable with the versions 6 to 8, that index the data by page.

Every new version improves some aspect of one of the two solutions, either in space occupation or performances, making use of succint data structure or other type of optimitation; the most important ones are done in correspondance of the hot spots of the program.

2 Specification Details

The Wikipedia statistics are organized (after the preprocessing phase) in a list of triples like:

```
<Date, Page, Counter>
```

The triple tells that a certain Page has ben visualized Counter times in a certain Date. The Date is expressed like a string in the format yyyymmdd-hh. The triples are unordered and not all the pages have been viewed in every date. The two queries that the program must be able to answer are a range and a top k query. We could define these two operation like that:

```
Range (Date1, Date2, Page): For every date between Date1 and Date2
    returns the number of times that the page Page has been accessed in
    that specific date, ordered by date. If a page has never been
    viewed on a certain date, the counter for that date is zero.
```

```
TopK (Date1, Date2, Page, K): Returns the K dates (in the interval of
    time between Date1 and Date2) where the page Page has been viewed
    the highest number of times, along with their counter.
```

3 Overview

The Main programs don't change with a new version of the implementation so their code is useful to understand the general structure of the Builder and the Interrogator. The following is the code for the Builder_Main: his task is to instantiate the correct version of the Builder class, based on the number given in input, contained in a different Builder.h file.

Builder_Main Using a macro code block to avoid using virtual classes. The operations are the same with every class version, so the code is the same.

```
#include <Builder1.h>
#include <Builder2.h>
#include <Builder3.h>
#include <Builder4.h>
#include <Builder5.h>
#include <cstdint>
#include <chrono>
#include <ctime>

//-- readDataset(): the class reads the "time_series.txt" dataset file,
//parse the triples and stores them in a temporary data structure.
//-- the time measuring starts
//-- compressStructure(): the temporary data structure is reorganized in
//some compressed structure, usually a vector with some kind of indexing
//-- serialize(): the compressed structure is stored in a index file
    along
//with all the information needed to retrieve information from it
//-- the time measuring stops
#define BUILD \
  do { \
      b.readDataset(); \
      std::chrono::time_point<std::chrono::system_clock> start, end; \
      start = std::chrono::system_clock::now(); \
      b.compressStructure(); \
      b.serialize(); \
       end = std::chrono::system_clock::now(); \
       std::chrono::duration<double> elapsed_seconds = end-start; \
       cout << "elapsed time: " << elapsed_seconds.count() << "s\n"; \
} while (false)

int main(int argv, char** argc)
{

   int8_t versionID;

   //command line: Builder_Main <VersionID>
   if (argv != 2) return -1;
   versionID = stoi(argc[1]);
```

```
  //each class contains a different implementation of the same
  //sequence of functions (see the macro BUILD)
  switch (versionID) {
  //"by date" versions
  case 1 : {Builder_V1 b (versionID); BUILD; break;}
  case 2 : {Builder_V2 b (versionID); BUILD; break;}
  case 3 : {Builder_V3 b (versionID); BUILD; break;}
  case 4 : {Builder_V4 b (versionID); BUILD; break;}
  case 5 : {Builder_V5 b (versionID); BUILD; break;}

  //"by page" versions
  case 6 : {Builder_V6 b (versionID); BUILD; break;}
  case 7 : {Builder_V7 b (versionID); BUILD; break;}
  case 8 : {Builder_V8 b (versionID); BUILD; break;}
  default : {Builder_V8 b (versionID); BUILD; break;}
  }

}
```

Interrogator_Main Similar to the Builder_Main, but with a different sequence of functions.

```
#include <Interrogator1.h>
#include <Interrogator2.h>
#include <Interrogator3.h>
#include <Interrogator4.h>
#include <Interrogator5.h>
#include <cstdint>
#include <chrono>
#include <ctime>

//-- deserialize(): reads the compressed data structure from the index
//-- readyQueryset(queryset) scanning the queryset file, instantiate a
//RangeQuery or a TopkQuery object for every query and stores them
//in two vectors containing a list of all the Query objects for each type
//-- time measuring starts
//-- performQueries(): scans the Query vectors produced by the
//readQueryset functions and calls the performQuery method on each
//query object
//-- the time measuring stops

#define INTERROGATE \
  do { \
   i.deserialize(); \
   i.readQueryset(queryset); \
   std::chrono::time_point<std::chrono::system_clock> start, end; \
   start = std::chrono::system_clock::now(); \
```

```
   i.performQueries(); \
   end = std::chrono::system_clock::now(); \
    std::chrono::duration<double> elapsed_seconds = end-start; \
    cout << "elapsed time: " << elapsed_seconds.count() << "s\n"; \
} while (false)

int main(int argv, char** argc)
{

   int8_t versionID;


   //command line: Interrogator_Main <versionID> <queryset file>
   if (argv != 3) return -1;
   versionID = stoi(argc[1]);
   string queryset = argc[2];

   //each class contains a different implementation of the same
   //sequence of functions (see the macro INTERROGATE)
   switch (versionID) {
   //"by date" versions
   case 1 : {Interrogator_V1 i (versionID); INTERROGATE; break;}
   case 2 : {Interrogator_V2 i (versionID); INTERROGATE; break;}
   case 3 : {Interrogator_V3 i (versionID); INTERROGATE; break;}
   case 4 : {Interrogator_V4 i (versionID); INTERROGATE; break;}
   case 5 : {Interrogator_V5 i (versionID); INTERROGATE; break;}

   //"by page" versions
   case 6 : {Interrogator_V6 i (versionID); INTERROGATE; break;}
   case 7 : {Interrogator_V7 i (versionID); INTERROGATE; break;}
   case 8 : {Interrogator_V8 i (versionID); INTERROGATE; break;}
   default : {Interrogator_V8 i (versionID); INTERROGATE; break;}
   }


}
```

The difference between the various versions is how these functions have been implemented, so we'll describe, for every function of the Builder and Interrogator programs, the implementation across the versions and we'll discute the performance improvements (or loss).

Notes on the Measurements See section 6 for details on the environment, the modalities and the motivations behind the tests. We also report some experiments with completion time.

4 The Builder

4.1 Function: readDataset()

The first part is the same in every version of the software: the preprocessed dataset file is opened and scanned line per line, extracting a Date, a Page and a Counter from each. Then a *memo* function is called, passing such triple as an argument. The memo function changes. In the version from 1 to 5, they are stored by date. From the version 6 onwards, they're stored by page.

Version 1 In the baseline (Version 1) we use a std::map with integers as keys and a vector of pair of strings as values. The dates are immediately converted in integers calculating the difference from a reference date in an hour grain:

```
int32_t ID = 0;

ID += stoi(hour) - refHour;
ID += (stoi(day) - refDay) * 24;
ID += (stoi(month) - refMonth) * 744;
ID += (stoi(year) - refYear) * 8928;

return ID;
```

Using the integer date as a key, the pair of strings containing the Page and the Counter is pushed back in the vector that matches the integer key, so to create a list of pair $< Page, Counter >$ for every possible date (and some dates that don't exist, actually, which obviously will have no pairs related).

```
    if (structure.find(dateAsNumber) != structure.end())
    {
       vector<pair<string, string>> &sameDateListOfPair =
           ref(structure[dateAsNumber]);
       sameDateListOfPair.push_back(values);
       structure[dateAsNumber] = sameDateListOfPair;
    }
    else
    {
       vector<pair<string, string>> sameDateListOfPair;
       sameDateListOfPair.push_back(values);
       structure[dateAsNumber] = sameDateListOfPair;
    }
}
```

Version 2 In the first update of the memo function, the vectors of strings have been replaced with a concatenation of strings, which is better for a number of reasons, with being more memory-friendly among them (and has also proven dramatically more efficient in the measures, see section 6). As the strings are

concatenated in a unique, very long string, a vector of integer telling where every string starts has been added.

```
    string values = pagina + numero;
    if (structure.find(dateAsNumber) != structure.end())
    {
       string &sameDateListOfPair = ref(structure[dateAsNumber]);
       int32_t sameDateListOfPair_length = sameDateListOfPair.length();
       positions[dateAsNumber].push_back(sameDateListOfPair_length);
       positions[dateAsNumber].push_back(sameDateListOfPair_length +
           pagina.length());
       sameDateListOfPair += values;
       structure[dateAsNumber] = sameDateListOfPair;
    }
    else
    {
       positions[dateAsNumber].push_back(0);
       positions[dateAsNumbser].push_back(pagina.length());
       structure[dateAsNumber] = values;
    }
 }
```

Version 3 The third version is a first failed experiment with succint data structures. The idea was to use a rrr_vector, that is a binary characteristic vector of the string delimiter integer array created in the Version 2. While the space usage has improved, the performace have worsened. If m is the total length of the concatenated strings and n is the number of such strings, for a classic vector rappresentation we pay $O(nlogm)$ in space, while the rrr_vector occupies $O(m)$ space. If $\frac{m}{n}$ (the average length of a string) is not so high, the latter can be less space consuming than the former.

$$n * log(m) > m?$$
$$n * log(m) > n * \frac{m}{n}?$$
$$log(m) > \frac{m}{n}?$$

Even if the gain in space is not huge, the index was about 20 mega bytes smaller than the Version 2 index (450 Mb against 470 Mb), resulting in a saving of about 4,25% of the total space. Also Elias-Fano has been tested here, but for the same reasons the rrr_vector has proven useful, the Elias-Fano solution was not functional.

Version 4, 5 This is the best solution we could find for the "by date" readDataset function. Here we use a more compact way of traslating dates in integers: a minimal perfect hash function. In order to build the function we first need a vector containing all the possibile keys. We do that with the class

bit_vector_alloc, which uses the old way of translating the dates to remember if the date passed as an argument to the memo function has already been saved in the keys vector. After the reading phase, the keys vector will be used to build the minimal perfect hash function. The precondition needed by the minimal perfect hash function to work is respected, because we know that all the queries will ask for existent dates (assuming that the dates that not feature in the dataset don't exist).

Furthermore, the triples $< page, date, number >$ are not stored in a complex data structure here, as we'll build a compressed data structure later (this two-phase storing is necessary, no way has been found to store immediately the triples in a compressed structure). We simply concatenate all the triple in a very long string. Actually we had to divide the string in a few pieces, because the request of an enormous quantity of contiguus memory made the builder cause a bad_alloc() exception.

The advantage of using a unique string (or actually, only a few strings), albeit disordered, instead of using a vector of strings, it's not only on the fact that we have a smaller number of memory faults, but also on the fact that the string is much, much more long than before. Actually now the string is so long that we had to cut it in typically four or five pieces to avoid bad_alloc problems. Now that the string is so much long, if m is the length of the string and n is the number of different words in the string, the value of $\frac{m}{n}$ is far higher than before, so that we can think to use the Elias-Fano representation to store the vector of integers that delimit the words in the string. Since the sequence of delimiters is not-decreasing, this is possibile.

A class called sd_vector_alloc was used to store integer until is no more needed, and then close the vector and build an sd_vector, containing the Elias-Fano support, over it. So here is the final "by date" version of the memo function, which is the core of the readDataset function.

```
void memo(const string &data, const string &pagina, const string
    &numero) {
   //initialization of the bit_vector_alloc structure (here named
       bitKeys)
   initBitKeys(data);

   //the keys vector must contain all the different dates as CStrings
   if (bitKeys.check(data) == 0) {
      bitKeys.insert(data);
      keys.push_back(data.c_str());
   }
      int32_t l = pairs[pidx].length();
      int32_t d = data.length();
      int32_t p = pagina.length();
```

```
        //pairDelim is a object of the sd_vector_alloc class, with
            Elias-Fano support
        pairDelim.push_back(l);
        pairDelim.push_back(l+d);
        pairDelim.push_back(l+d+p);

        //pairs is indeed a vector, but with very few entries.
        pairs[pidx] += data+pagina+numero;

        //cut the string to reduce contiguous alloc problems
        if (pairs[pidx].length() >
            pairs[pidx].max_size()/MAX_SIZE_FRACTION)
        {
           pidx++;
           pairs.push_back("");

        }

    }
```

Version 6 Starting from version 6 we totally changed approach by storing the triples by page instead of by date. The readDataset function obviously still exists, but is used in a slightly different way. Here the file is accessed two times: in the first time we read all the triples in order to build the vector of all the different pages, that is the vector of the minimal perfect hash keys according to the new "by page" approach, and the vector of all the different dates, useful to translate them in integers using their position in the array after sorting.

In the second visit of the file we actually stored the triples in a data structure which varies between version 6 and version 7. Since we read the dataset two times, the readDataset() function is called two times, but is used as an higher order function, receiving as an argument another function which "do something" with the read triples.

The *read* function is the one used to build the pages vector and the dates vector, and so it is passed as an argument to the readDataset() function in the first time is called.

```
    inline void read(const string &data, const string &pagina, const
        string &numero) {

      if (pageCheck[pagina] != 1)
        {
           pageCheck[pagina] = 1;
           pages.push_back(pagina.c_str());
        }
```

```
    if (dateCheck[data] != 1)
       {
          dateCheck[data] = 1;
          dates.push_back(data);
       }


}
```

The *memo* function is passed to the readDataset function the second time and is used to store the triples in a structure. In this version (version 6) the structure is a $vector < vector < int >>$ type which stores, for every page identified by an integer, a vector of all the counters ordered by date.

```
inline void memo(const string &data, const string &pagina, const
    string &numero) {


    uint32_t Pid = cmph_search(hash, pagina.c_str(),
        (cmph_uint32)strlen(pagina.c_str()));
    uint32_t Did = dateCheck[data];

    (Structure[Pid])[Did] = stoi(numero);

}
```

Version 7 In the version 7 we sacrifice performance in order to obtain a gain in space occupancy. All the vectors in the second level of the $vector < vector < int >>$ structure used before have been concatenated, resulting in a unique vector rather than a matrix. The purpose of using only one level of vectors is maximize Elias-Fano gain. The counters are not in non-decreasing order but using some sort of prefix sums we can build a non-decreasing sequence.

As stated before the larger is m with respect to n (with m maximum value and n number of entries), the larger is the advantage of using Elias-Fano for non-decreasing sequences. We tried to use a separate Elias-Fano vector for every second level vector, but there are many pages whose vector of counter was very sparse, resulting in a very small distance between the values. Since all the counters are summed up, the value of m raise quickly and with it the efficiency of the Elias-Fano solution.

4.2 Function: compressStructure()

For different reasons it was impossibile to store the $< Date, Page, Counter >$ triples in a compressed data structure immediately upon reading them from the file: either because we didn't know the least recent date or because we needed to know all the dates to build the hash function. So this function, the compressStructure function, must read the temporary data structure used for the readDataset function and store the triples in a more convenient way to use them in the Interrogator program. As stated before, the versions from 1 to 5 try to do their best to optimize a data structure organized by date, while from the version 6 onwards we switch to the "by page" organization.

Version 1 In the baseline we had a std::map mapping the dates expressed as integers to a vector of pair of strings with all the pages accessed in that date and relative counters. To compress the data structure we simply used a vector instead of a map, so a vector of vectors, using the integers to index them.

Since the integer could also be negative (we used a random date as a reference for translating the other dates in integers) we had to memorize the least recent date as an integer, here called *min*.

```
inline void compressStructure()
{
      if (structure.empty()) return;
      shift = -(min);
      max = max + 1;
      compactStructure.reserve((max +
          shift)*sizeof(vector<pair<string, string>>));
      for (int i = 0; i < (max+shift); i++)
      {

         vector<pair<string, string>> init2;
         compactStructure.push_back(init2);
      }


      for (auto const ent1 : structure)
      {
         compactStructure[ent1.first + shift] = ref(ent1.second);
      }


}
```

Version 2 The Version 2 is not so different, apart from the changes that follow the use of concatenated strings instead of vector of strings, as stated in the previous subsection (see subsection 4.1).

```
inline void compressStructure()
{
      if (structure.empty()) return;
      shift = -(min);
      max = max + 1;
      compactStructure.reserve((max + shift)*sizeof(string));

      for (int i = 0; i < (max+shift); i++)
      {
         string init2;
         vector<int32_t> init1;
         compactStructure.push_back(init2);
         compactPositions.push_back(init1);
      }


      for (auto const ent1 : structure)
      {
         compactStructure[ent1.first + shift] = ref(ent1.second);
         compactPositions[ent1.first + shift] =
             ref(positions[ent1.first]);
      }


}
```

Version 3 The compressStructure function is where the rrr_vector is really used in Version 3 (see subsection 4.1). The code is very similar to the Version 2 code, except that we call a compress function before passing the position from the temporary structure to the compressed one.

```
rrr_vector<> compress(vector<int32_t> & large) {

   bit_vector small (large.size()*(1<<5),0);
   int32_t i = 0;

   for (const int32_t number : large)
   {
      small.set_int(i,number,32);
      i+=32;
   }
   rrr_vector<> rrr_small(small);

   // m/n too low: characteristic vector better than Elias-Fano (534
       mb vs 450 mb)
   //cout << "small: " << size_in_mega_bytes(rrr_small) << endl;
```

```
    return rrr_small;
}
```

Version 4, 5 In the last "by date" solution we had a few strings containing all the triples in sequence and a sd_vector used to store the position of each word in the string. Is here built a vector which uses as indexes the result of a minimal perfect hash function, so to create a vector of strings organized by date.

```
        unsigned int id = cmph_search(hash, date.c_str(),
            (cmph_uint32)strlen(date.c_str()));

        delimPerDate[id].push_back(listPerDate[id].length());
        delimPerDate[id].push_back(listPerDate[id].length() +
            page.length());

        listPerDate[id]+= page + count;
```

The dimension of the index would be the smallest among all the "by date" versions, with a total of 394Mb in the tests, if weren't for the fact that we have to serialize the keys vector too (otherwise we don't know how to order the answers by date, see later).

Version 6, 7 Here is where, in the "by page" approach, we call the readDataset function a second time, passing the memo function as an argument to the other function to build the compressed structure. In version 6, the compressStructure function does nothing more. In the version 7, where we use Elias-Fano, the compressStructure function is responsible for summing the counters to make them a non-decreasing sequence and for applying the Elias-Fano construction.

4.3 Function: serialize()

Apart from the kind of library used to serialize the compact structure, the serialize function is always more or less the same. In the final version of the "by date" approach we had to serialize the keys vector containing all the possibile dates, because a better solution to sort the answers by date has not been found. Even if the hash function itself can translate every existent date in a correct index to access the vector, it's not sufficient to determine the order of the dates in an interval of time (requested by the two types of query).

In the "by page" approach is mandatory to serialize the dates vector, because it's the only way to translate the dates in integers.

5 The Interrogator

5.1 Function: deserialize()

Not much to say about the deserialize() function. From the fifth version onwards the keys vector is here sorted in order to answer correctly to the queries requiring the results to be sorted in the answers. The keys vector dimension is much less important than the structure's dimension, but in any case if the results has to be sorted, sooner or later is mandatory to sort the dates.

5.2 Function: readQueryset(queryset)

The queryset file given in input is opened, scanned line by line and each query is memorized in a RangeQuery or TopkQuery class, then added to a list. The queries are performed later: this is done in two different moment not to include the file reading time in the measures. Here is the saving subroutine:

```
inline void saveOp(const string & operation,
                   const string & time1,
                    const string & time2,
                    const string & page,
                    const string & K)
{

   if (operation == "range")
   {
      RangeQuery_V1 query(compressDate(time1) + normalize,
          compressDate(time2) + normalize, page);
      range_set.push_back(query);
   }

   if (operation == "top_k")
   {
      int64_t k = stoi(K);
      if (k <= 0) return;
      TopkQuery_V1 query(compressDate(time1) + normalize,
          compressDate(time2) + normalize, page, k);
      topk_set.push_back(query);
   }

}
```

What is showed here is the "by date" first version, where we transform the dates in integers using a reference date and then we translate all the negative nates when we know the least recent date. In the "by page" approach both the time interval and the pages are passed as integers, the former using the dates vector and the latter using the minimal perfect hash function.

5.3 Function: performQueries()

The outer code of this function is very simple: the queries are executed. (From the version 6 onwards we measure the performance by query, so here we can find also some measuring. The Structure decompression in Version 8 in order to use rmq is also done here).

```
inline void performQueries()
{
   for (RangeQuery_V1 query : range_set)
   {
      query.performQuery(compactStructure);
   }

   for (TopkQuery_V1 query : topk_set)
   {
      query.performQuery(compactStructure);
   }
}
```

What is different is how the queries are performed.

Version 1 Here we have a vector of lists with the $< page, counter >$ couples relatives to each date. Each date is expressed like an integer so the dates are implicit in the outer vector indexing. This structure is passed as an argument to the query class which simply translates the time interval requested in the query in an interval of two integers, then the outer vector is scanned in the subportion asked by the query and answers with a range or a top k result. The main drawback is that the vector has many empty spaces in corrispondence with unexistent dates.

In the TopKQuery class we use a heap to store the K pairs with the highest counters. The following code is in the TopkQuery class:

```
//begin, end are the extremes of the time interval
for (int i = begin; i < end; i++)
{
   //sameDateListOfPair is the inner vector, as the name suggests
   sameDateListOfPair = ref(compactStructure[i]);
   uint32_t j = 0;
   uint32_t dim = sameDateListOfPair.size();
   pair<int32_t, uint32_t> Kpair;

   //Scanning the <Page, Counter> couples with same date
   while (j < dim)
   {
      //If the page is the one requested by the query...
      if (sameDateListOfPair[j].first == page)
```

```
        {
           Kpair.first = i;
           Kpair.second = stoi(sameDateListOfPair[j].second);

           //...in the RangeQuery class, here we print
           //...but here, we must first check if the heap is full
           if (topK.size() >= K)
           {
              //the heap minimum is compared with actual counter
              if (topK.top().second < Kpair.second)
              {
                 //if it's the case, the minimum is removed
                 topK.pop();
                 topK.push(Kpair);
              }
           }
           else topK.push(Kpair);

           break;
        }
        else j++;
     }
  }
```

Version 2 Here there are some complications due to the fact that we use concatenated strings instead of vector of strings, but nothing different in principle.

Version 3 Even more complications here, due to the fact that the rrr_vector used in this version (see subsection4.1) is a bit vector. This version of the interrogator is dramatically inefficient, simply a wrong choice, so nothing more to say.

Version 4 In the Version 4 we no more have the lists of couples sorted by date, as they are stored in the vector in a order decided by the minimal perfect hash function. So even if is very simple to find the query extremes in the vector by translating the extremes of the time interval with the hash function, it is not so simple to find all the dates in chronological order from the first to the second.

There were two alternatives: wasting time, using the find function to retrieve the inner lists, or wasting space, using two vector to redirect the indexes in each iteration. We opted for the second choice, so we have a Chrono_ID vector which contains as a value the hash ID of the dates and a ID_Chrono vector which is ordered by the ID values, and in the cells are stored the unsigned integers corresponding to the position of the ID in the other vector. With this double access, we could get the extremes and move from one extreme to the other in two access per iteration.

```
//time1 and time2 are the result of the mph function applied to
    the string dates
for (int i = time1; i != time2; i =
    keys_Chrono_ID[keys_ID_Chrono[i]+1])
{
   list = ref(listPerDate[i]);
   delims = ref(delimPerDate[i]);
...
```

Version 5 The performQueries function is where the final "by date" version really differs from the fourth version. Two more optimization has been done, one regarding the substring function and one regarding the compare function used to manage the heap. These two optimizations have proven very effective, almost halving the execution time in the tests (see section 6).

The following test was an hotspot in the *perf* analysis:

```
if (list.substr(pagStart, pagLength) == page)  ...
```

This is because a new string is created only to be compared with the page string. Since this is done in every iteration and for almost every subportion of the long string, it is almost equivalent to a copy of the entire list of couples. We decided, in Version 5, to replace it with a low level string comparison, and this was successfull.

```
inline bool substr_cmp (const char * str1, const char * str2, size_t
    n) const
{
   bool equal = true;
   size_t i = 0;

   while (equal && i<n)
   {
      if (str1[i] != str2[i]) equal = false;
      i++;
   }

   return equal;
}
```

The other hotspot was the heap management. In the worst case (for instance if the requested page is in every date's list with increasing counters) we had to insert a new value in the heap for every date, paying $O(dlogk)$ time with d number of dates.

The fifth version of the Interrogator software uses RMQ to reduce the overhead due to Topk operations. Using the cartesian trees implementation, the rmq can retrive the index of the maximum value in constant time. We do k iterations using RMQ two times per iteration, in the left subarray and in the right subarray. This returns two values which are stored along with the extremes of their intervals in a heap. After k iterations we have the Top k values in the heap, paying $O(klogk)$, that doesn't depend on the number of keys.

```
for (size_t i = 0; i < K - 1; i++) {

      if (l < idx) {
         size_t m_left = rmq(l, idx-1);
         heap.push(make_tuple(m_left,topK[m_left], l, idx-1));
      }

      if (idx < r) {

      size_t m_right = rmq(idx+1, r);
      heap.push(make_tuple(m_right,topK[m_right], idx+1, r));
      }

      if (heap.empty()) break;
      tuple<int32_t, uint32_t, uint32_t, uint32_t> actual = heap.top();
          heap.pop();
      out << "Date: " << get<0>(actual) << " Counter: " <<
          get<1>(actual) << endl;
      l = get<2>(actual);
      r = get<3>(actual);
      idx = get<0>(actual);
}
```

Version 6 In the "by page" world things get very easy here. All we must do is to apply minimal perfect hash function to the page requested by the query, find the vector of counters relative to that page and scan every counter from the position corresponding to the left extreme of the interval of time to the position of the right extreme, and it's done. (The writing of the results are done outside the performQuery function in versions 6 onwards for measuring reasons). Here is the code of the topk query, which uses a queue to store the K highest counters along with their date found during the scan.

```
      for (int i = time1; i <= time2; i++)
      {
            pair<int32_t, uint32_t> Kpair;
            Kpair.first = i;
            Kpair.second = (Structure[page])[i];
            if (topK.size() >= K)
            {
```

```
            if (topK.top().second < Kpair.second)
            {
               topK.pop();
               topK.push(Kpair);
            }
         }
         else topK.push(Kpair);

   }
```

Version 7 Only minor changes here, mainly due to the fact that now the Structure is organized with Elias-Fano. So cannot simply access the Structure but we must use the select support for Elias-Fano non-decreasing sequences. By subtracting an entry to its predecessor we find the original counter.

Version 8 This version is only different from the previous in the fact that we no more use only a queue but we use also RMQ to find the Top K elements. The rmq support is applied to an uncompressed vector unfolded outside the class, in the performQueries function. See Version 5 for the analysis and details, as RMQ is used in the same way.

6 Measurements

The measurements are relative to an hardware accelerated 32-bit Ubuntu virtual machine, single core and with 1587 MB RAM free to use. The host machine is a Windows machine featuring an Intel Core^{TM}i7 with 2.2 Ghz CPU. They are performed using the Chrono library and the $Perf$ profiling tool.

We used two different tecniques in order to measure performances within the versions. In the Version 1 to 5 ("by date" world) we only measured the overall completion time. The test set is a one thousand queries random generated query set. Time intervals and K values (when needed) are random generated among allowed values across the test set.

Our idea here is the following: the Version from 1 to 5 are there only to show how dramatic a single choice can be (organizing "by page" instead of "by date") so would be a waste of time to do an accurate per-query measure. Once we have showed how much more performant is the Version 6 with respect the Version 5 in the very same test set, we switch to a more accurate test set. These measurements are done both with and without the file-reading time and using or not using the -O3 optimization flag.

Builder Here are the results of the Builder measurements in the test set.

Table 1: Builder: "by date" vs "by page"

	-O0 No File	**-O3 No File**	**-O3 All**	**Page Faults**	**Structure dim.**
Version 1	700.05 s	618.61 s	1117.32 s	1'353'887	620.8 Mb
Version 2	43.69 s	8.64 s	25.47 s	251'875	470.0 Mb
Version 3	184.33 s	136.72 s	164.46	334'372	450.7 Mb
Version 4	627.93 s	324.11 s	480.35 s	1'089'550	394.0 Mb
Version 5	627.93 s	324.11 s	480.35 s	1'089'550	394.0 Mb
Version 6	92.53 s	83.31 s	113.17 s	476'865	119.6 Mb

Interrogator These are the result for the 1 to 5 versions of the Interrogator program. The only purpose is to show how drastic is the performance gain by simply doing the correct choice (ordering by page instead of by date as stated previously). The version 6 is at least ten times faster than the fastest "by date" version, the version 5.

Table 2: Interrogator: "by date" vs "by page"

	-O0 No File	**-O3 No File**	**-O3 All**	**Page Faults**
Version 1	346.14 s	236.59 s	239.62 s	527'729
Version 2	243.40 s	159.12 s	166.33 s	19'101'011
Version 3	10'583 s	not meas.	6'113.69 s	279'921
Version 4	196.25	113.14 s	127.02 s	285'381
Version 5	101.96	38.23 s	55.23 s	285'377
Version 6	4.63 s	4.14 s	5.57 s	31'681

We can now switch to more accurate measurements between the Version 6, 7, 8, which are more interesting to measure accurately.

Table 3: Index space occupation

	Version 6	**Version 7, 8**
Space (Mb)	119.4	31.8

Range Queries The time intervals are given in milliseconds, varying K (for topk queries) and the dimension of the range. The range dimensions are expressed in hours. They are 5 hours and about a day (25 hours), about 3 days (75 hours), about 1 week (175 hours), about 2 weeks (350 hours), about 1 month (700 hours) and about 2 month (1400 hours). These measures are an average of a one thousand large sample.

Table 4: Range queries performances (ms)

	5 h	**25 h**	**75 h**	**175 h**	**350 h**	**700 h**	**1400 h**
Version 6	0.00057	0.00065	0.00511	0.00108	0.00329	0.00223	0.01155
Version 7, 8	0.00164	0.00836	0.02384	0.04804	0.10410	0.16835	0.28692

Top K Queries Computation measured for three possible ranges (25, 175 and 700 hours) and three possible K (5, 10 and 20).

Table 5: Version 6 (Double Vector)

	25 h	**175 h**	**700 h**
Top 5	0.00149	0.00250	0.00804
Top 10	0.00179	0.00331	0.01015
Top 20	0.00482	0.00837	0.02181

Table 6: Version 7 (Elias-Fano)

	25 h	**175 h**	**700 h**
Top 5	0.00467	0.03715	0.07400
Top 10	0.00483	0.04676	0.06929
Top 20	0.01393	0.04585	0.07545

Table 7: Version 8 (Elias-Fano, RMQ)

	25 h	**175 h**	**700 h**
Top 5	0.00604	0.01107	0.01761
Top 10	0.00990	0.01736	0.02657
Top 20	0.01128	0.02539	0.04079

7 Further Development

There are many aspect to improve, among the others the RangeQuery performances, which cannot benefit from the rmq. The use of Elias-Fano provokes a performance loss that, albeit theoretically admissible, is too high, so further investigation could be done. It's natural to assume that in order to build a smaller data structure we have to pay something, but it's certain that further investigations would go in that way.

8 Readme

In the conclusion we report the istructions to download, compile and execute the program. You can get the code here: `https://github.com/Feum/timeseries` Install git and do the following in a folder of your choice:

```
git clone https://github.com/Feum/timeseries
cd timeseries
make
```

In order to execute the program, you must do the following:

```
./Builder_Main <VersionN> <Source_File>
```

This will create the index, reading the formatted $< Source_File >$ and using the implementation of the version specified in the $< VersionN >$ parameter (from 1 to 8).

```
./Interrogator_Main <VersionN> <queryset>
```

The main program will pick an index according to the $< VersionN >$ parameter specified here (the index created by the Builder of that version, if exists) and will use that in order to answer the queries specified in the $< queryset >$ file. You can use the default query_set_1.txt file given or use Queryset_generator to create one (see below). As an example:

```
./Builder_Main 2 time_series.txt
./Interrogator_Main 2 query_set_1.txt
```

The results will be written in a file along with the version number, and a time measure (without file reading time) will be printed out. We also provide a standard, already formatted, source file, called time_series.txt (Wikipedia page view statistics preprocessed by Rossano Venturini). You can optionally download the file to test the program via git lfs extension. In the project folder type:

```
git lfs pull
```

And the download will start. Warning: the file has a dimension of 583Mb. To create a queryset with the generator, do the following:

```
./Queryset_generator <Seed> <ID> <Source_File>
```

The program will use the $< Seed >$ parameter to create a random query-set named $query\_set\_ < ID > .txt$ containing one thousand queries. The $< Source\_File >$ must be the same dataset file that the Builder program will use in the tests. Queryset_generator2 can create a more accurate query set. You can type:

```
./Queryset_generator2 <Seed> <ID> <Source_File> <Range> <K>
```

To build a queryset with one thousand queries of the type Top K (if $K > 0$) or Range (if $K <= 0$), using $< Range >$ as the range dimension (expressed in hours).