# Time Series Indexing
Laboratory On Algorithms for Big Data

Alessandro Romano
alerom90@gmail.com

## Abstract

This document reports the phases of design and implementation of a data structure that answers queries on page view statistics files for Wikipedia. A base solution is proposed in order to compare it with a more efficient one that exploits a succinct representation. The final results are presented by comparing time and memory complexity of the data structures implemented. This project is a mandatory of the course *Laboratory on Algorithm for Big Data* [1] - University of Pisa.

## Dataset description

The dataset is a unique csv file that collects page visits from January 2016 until October 2016. This file is preprocessed by starting from Wikimedia Dump [2]. Each row of this file is a triple <time,page,counter>, whose components are separated by a `tab`. The triple says the number of requests (counter) of a certain Wikipedia page (page) in a certain hour (time). The time is expressed with the format `YYYYMMDD-HH`, where `YYYY` is the year, `MM` is the month, `DD` is the day, and `HH` is the hour represented as a number from `00` to `23`. Triples where counter equals 0 were removed from the file. The following lines are a small sample of the file:

```
20160713-06 Italy 183
20160626-23 Galileo_Galilei 65
20160626-23 Game_of_Thrones 2894
```

After data preprocessing we have 5998 pages and for each page 5221 timestamps.

## Goals and tools

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.

For data structure construction two strategies are followed, the first one is to build it in a simply way that we call Baseline, the second exploits a succinct rappresentation and it's called Index1. Those two implementations are compared by analyzing their complexity in term of time and space. Figure 1 shows two execution phases, the data structure is built by reading data from file then it will be serialized in order to answer the two queries.

The entire project is written in C++11 with Cereal [3] as serialization library for Baseline. Index1 is based on SDSL [4] that implements succinct data structures. At last building and testing by using CMake [6] and G-Test [7].

Performances are analyzed with perf [8] and results are explored with Python3 and Matplotlib [5] as plotting library.

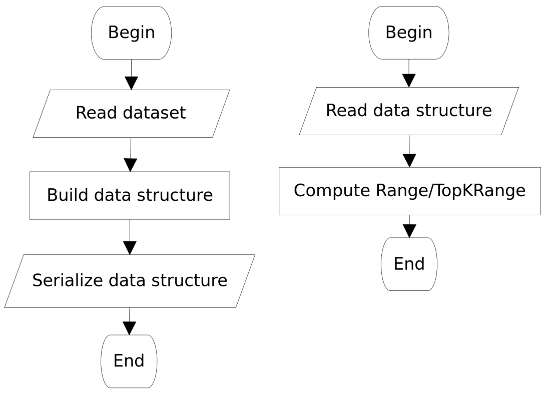


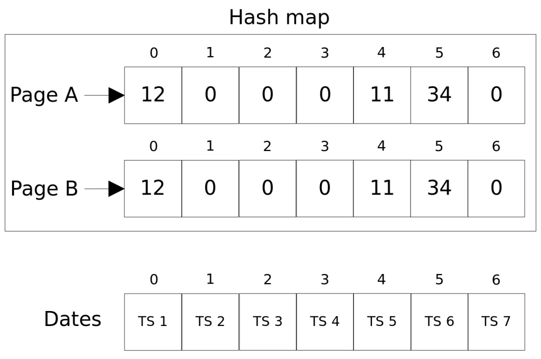
Figure 1: Pipeline execution

## Solutions

For each solution is presented the concept behind the implementation with focus on how the data structures are built.

### Baseline

The main data structure is a `std::unordered_map` with name page as key and a vector of visits as value. This vector is built by starting from all dates available on the dataset that are saved in a second vector in order to mapping each indexes with a time-stamp as shown in Figure 2. A second map with time-stamps as key and indexes as value, is also used in order to avoid call of `find()` to retrieve an index given a time-stamp.



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Page A | 12 | 0 | 0 | 0 | 11 | 34 | 0 |
| Page B | 12 | 0 | 0 | 0 | 11 | 34 | 0 |
| Dates | TS 1 | TS 2 | TS 3 | TS 4 | TS 5 | TS 6 | TS 7 |

Figure 2: Baseline data structure

Function `range()` is quite simple. Given a page and a time-stamps range, the two maps introduced above allow constant time access to a slice of vector of visits the we need.

Function `rangeTopK()` retrieve the slice of visits in the same way as `range()` does. A priority queue it's used in order to select top K visits by visiting the array just once.

### Index1

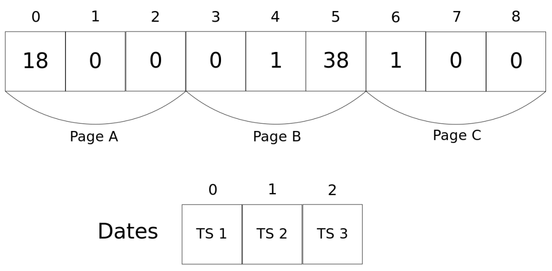
In computer science, a succinct data structure is a data structure which uses an amount of space that is "close" to the information-theoretic lower bound, but (unlike other compressed representations) still allows for efficient query operations[1]. As literature says this kind of data representation fit our problem very well and SDSL provides a documented implementation of Elias-Fano succinct data structures.

The following SDSL classes do what we need:

- `sdsl::sd_vector` is a bit vector representation and with `sdsl::select_support_sd` compresses and retrieves data by starting from an increasing sequence of integer

- `sdsl::rmq_succinct_sct` it's used to determine the position of the minimum value in an arbitrary subrange of a vector.

Since `sdsl::sd_vector` an increasing sequence of integer we accomplish it by saving the visits of all pages in a single vector on which will be made prefix sum[2]. For each page the staring index it's saved (Figure 3).



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 18 | 0 | 0 | 0 | 1 | 38 | 1 | 0 | 0 |

| | 0 | 1 | 2 |
|---|---|---|---|
| Dates | TS 1 | TS 2 | TS 3 |

Figure 3: Vector of visits for Index1

[1] `https://en.wikipedia.org/wiki/Succinct_data_structure`
[2] `https://en.wikipedia.org/wiki/Prefix_sum`

Data structures construction follows the scheme in Figure 4.

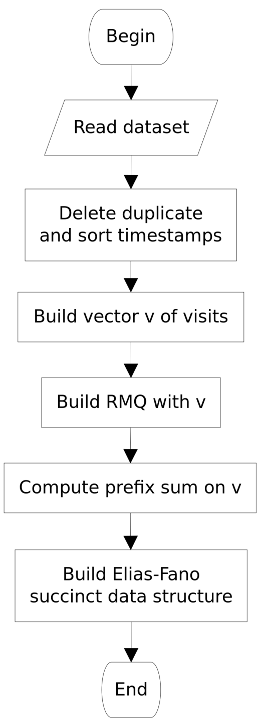


Figure 4: Data structures construction on Index1

Function `range()` uses bit vector and retrieves the visits of a specific page for a specific time-stamp by calculating the value before prefix sum.

Function `rangeTopK()` is based on RMQ that returns the index of max visits in range $[i, j]$. The idea beyhind implementation of this approach comes from a tutorial presented at SIGIR 2016 [9].

## Results

### Construction time

In order to understand the improvements that Index1 should bring, it's compared with Baseline by measuring the amount of space that data structures use and the time needed during construction and query time.

The results in term of time and space are shown in Table 1. Baseline is a little bit more faster than Index1 because of RMQ construction but the improvement in term of memory it's very high, almost 90MB less.

| | Time (ns) | Space (byte) |
|---|---|---|
| Baseline | 21, 964, 580, 358 | 125, 560, 461 |
| Index1 | 27, 444, 871, 054 | 43, 760, 976 |

Table 1: Construction time

### Query time

Several sets of queries are created in order to measure execution time for `range()` and `rangeTopK()`. Each set contains 10, 000 queries with a given range from 10% to 100% of the whole range of time-stamps. 100% of the whole range means that `range()` is computed on a period the goes from the nearest to the farthest date of the wikimedia dump. This approach allows us to see how execution times change if range increases.

Chart 5 compares execution time of `range()` for all sets of queries. With increasing of range and K the time gap between Index1 and Baseline increases linearly.

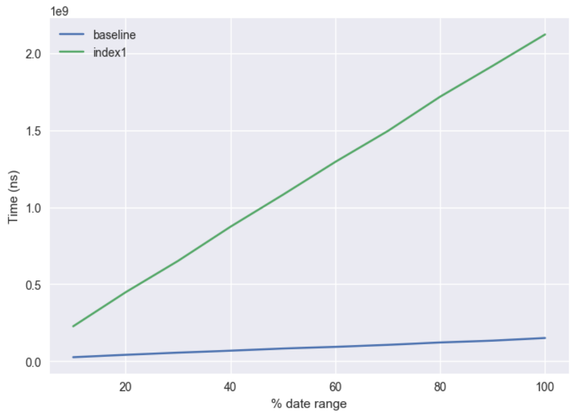


Figure 5: Range execution time

Function `rangeTopK()` needs a parameter K therefore the time executions of two implementations is analyzed by showing results for each range by increasing number of K. Chart 6 shows a strange behavior in fact Index1 should be more

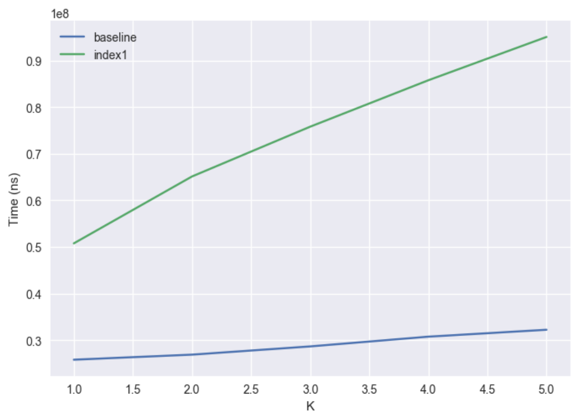


Figure 6: rangeTopK 10% execution time

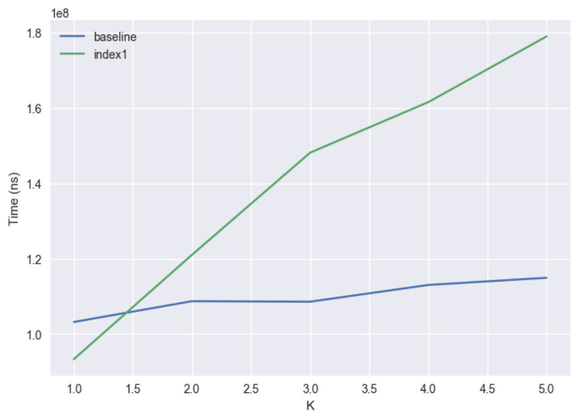


Figure 7: rangeTopK 90% execution time

faster then Baseline thanks to RMQ. When queries are computed almost on the whole maximum range of dates, then Index1 seems the best one for small value of K, as shown in charts 7-8.

## Conclusion

Index1 should be more faster then Baseline in term of query time, but it's not. According to SDSL documentation each RMQ takes constant time but it needs about one microsecond and it's linearly dependent on K, besides all queries made by Baseline scan data which probably are already in cache. In fact when queries are computed on the maximum range of data with $K = 1$ Index1 is clearly the fastest between the two. With a bigger dataset (e.g. $5,000,000$ visits or $5,000,000,000$), the baseline will be

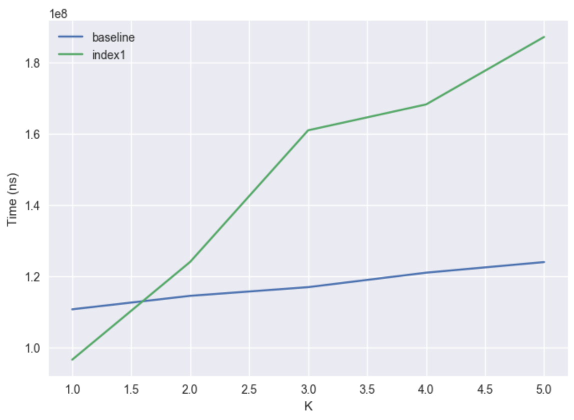


Figure 8: rangeTopK 100% execution time

much slower and the RMQ based solution will be always faster for reasonable value of K.

## References

[1] Lecture's page
`https://goo.gl/HHqoj7`

[2] Wikimedia dump
`https://dumps.wikimedia.org/other/pagecounts-raw/`

[3] Cereal
`http://uscilab.github.io/cereal/`

[4] SDSL
`https://github.com/simongog/sdsl-lite`

[5] Matplotlib
`https://matplotlib.org`

[6] CMake
`https://cmake.org`

[7] G-Test
`https://github.com/google/googletest`

[8] Perf
A performance analyzing tool in Linux

[9] Succinct Data Structure Tutorial at SIGIR 2016
`https://github.com/simongog/sigir16-topkcomplete`