Miguel Alcón Doganoc
Advanced Data Structures
June 27, 2019

Summary of the presentations

1 Jumplists

A jumplist is a jump-and-walk dictionary data structure. In essence, it is an ordered list whose nodes have an additional pointer: the jump pointer, which makes it similar to the binary search tree (BST). This modification speed up searches. Its performance is within a constant factor of an optimal BST. As a type of list, a jumplist supports the basic operations like balancing, searching, insertion and deletion. A jump pointer cannot point to any successor. It is formed by two integers: the size of the subjumplist of the node and the size of the next sublist.

My partner Joao Hugo Ballesteros centered his experiments in comparing the jumplist data structure with the well-known dictionary std::map of C++. As a result of these experiments, he observed that jumplists outperform the class std::map in general, but in extreme cases, the other has a better performance. However, with the O3 option of C++, the std::map outperforms by far the jumplists.

2 Sparse Tables

The Sparse Table wants to solve the sequential file maintenance problem. I.e., we want to maintain a partial order in an array, while allowing insertions and deletions. There were lots of proposals to solve this problem. For instance, the first proposed solution to this problem was to let gaps between elements in the list. Another one was to redistribute (adding gaps again) after some numbers of insertions, like the librarian sort. All the costs of these proposed solutions were an amortized cost, until 2017. In this year, someone proposed the first deamortized algorithm to solve the problem, the Sparse Tables.

A sparse table is, in essence, an unbalanced ternary tree, where some of the nodes point to usable positions of the array, and the others point to buffers (occupied positions). Clean operation is the most important one. It is in charge of empty all usable positions after some insertions. Parent nodes take care of cleaning their children nodes. My partner Ludvig Janiuk compared a sparse table with the naive algorithm. As a result of this experiment, he noticed that read and write operations over the sparse table is 2 times (in $\log_{10}$) faster than the naive algorithm.

3 Spectral Bloom Filters

The Spectral Bloom Filter (SBF) is an extension of the original Bloom Filter to multi-set, allowing estimates of multiplicities of individual keys with a small error probability. It also allows filtering elements whose multiplicities are within a requested spectrum, as well as deletion, and update. Its main application is the membership test in a fraction of the storage size, but it is also used in the sliding window protocol and streaming data.

My partner Wilmer Vidal Urichi focused his experiments in comparing two possible algorithms that the SBF can use to apply the membership test, which are minimum selection (MS), and minimal increase (MI). In the first experiment, MI had a minus error rate than MS, while in the paper they are equal. Wilmer mentioned that it could happen because of his implementation. The other experiment was the same as the first one but with a different number of hash functions. MS and MI errors followed the same curve of error variance. As a final comment, he said that SBF is a great extension, with efficient maintenance, dynamic, etc.