Miguel Alcón Doganoc
Advanced Data Structures
June 21, 2019

Presentations

1 Jumplists

1.1 Introduction

A jump-and-walk dictionary data structure.

Dictionary data structures have a long standing history in Theoretical Computer Science. Jumplist support basic operations Ordered list whose nodes have an additional pointer: the jump pointer. Speed up searches. Similarity with the binary search. Linear storage. Their performance are within a constant factor of optimal BSTs. Randomized jumplists in one paper. He implemented a deterministic version of jumplists because the results on different papers the deterministic version has better performance.

Jump pointer cannot point to any successor of x, they must meet the following conditions. The jump pointer must point forward the list

Every pointer also has two integers: the size of the subjumplist of that node, and the size of the next sublist of that node. And therefore the size of the list starting at can be given as size(X) = 1 + nsize[x] + jsize[x].

The deterministic has some key differences with respect to the original. For starters, the list is not circular, so the last node will not point to the header.

There is an etra condition for the jump pointers: For any node y, if junp[y] = y and y is not the last node of the list, then there exists a node where k[x] ¡ key[y] and jump[x] = next[y].

Also the jump pointer of a node is not chosen at random, it is chosen based a balancing criteria.The goal is to mantain, for every node, the ratio of the size of the next sublist and the jump sublist within a specific constant range.

1.2 Operations

Operations are similar in both the randomized and the deterministic one.

- Balancing (on insertions and deletions): we need to update the jump pointers, and it is crucial. It works in O(n).
- Contains: works by finding the node with largest less or equal key to the key we are trying to find. Then check if the node returned has the same key as the query.
- Insert: first find the would-be predecessor of the node we are trying to insert, increasing by 1 the size of the path we are traversing. You make next[x] point to y (the node we are inserting) and if x was not the last node of the list, then jump[y] takes the pointer of the jump of its successor, and it from its successor so on.
- Delete: find the wold-be predecessor of the node we are trying to delete, increasing by 2 the size of the path we are traversing. If we got the node to be deleted x via jump pointer, the value of this pointer is set to the successor of x. The next[y] where y is the predecessor fo x is set to the successor of x. If this next sublist of x is empty, the deletion terminates.......

1.3 Results

Inclusion, jumplist outperforms std::map except in extreme cases. Contains, jumplist outperforms std::map except in extreme cases. Inclusion, jumplist outperforms.

1.4 Conclusion

The DS is very good, and still room for improvement. It supports repeated keys, while std::map not.

1.5 Questions

With O3 C++ option std::map outperforms by far the jumplists. Jumplists implementation does not use C++ templates while map does it.

2 Sparse Tables

Problem: sequenctial file maintenance problem. We want to mantain a partial order in an array, allow insertions and deletions.

Applications: Persistent DS, graphs algorithms, sorting,...

Solution: let gaps between elements in the list But this is not the 'final solution'. Redistribute (adding gaps again) after some numbers of insertions. Librarian sort. Amortized cost until 2017, first deamortized algorithm.

Unbalanced ternary tree. Nodes indexing usable positions of the array, others pointing to buffers (ocupied positions). Clean operation: let all usable positions empty after insertions. Parents are in charge of cleaning its children nodes.

Less than linear overhead of space. (If we have 1000 array, only 1/5 of the space would be use for storing the real data, so approx. we have a 5k size array).

Results: read and write is 2 (log10) times faster than the naive algorithm.

3 Spectral Bloom Filters

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.

Applications: membership test in a fraction of the storage size; SBF: dynamic, translates into new applications: Sliding windows, Streaming data.

Bloom filters can have errors.

Spectral bloom filters replaces the bit vector V with a vector m of counters, C. The counters in C roughly represent multiplicities in S. Initially, all counters in C are set to 0. When inserting an item form S, we increase the counters by 1. It also allows from deletions by decreasing the counters, and 'updates' (by performing a deletion followed by an insert).

Construction and Maintenance. Querying.

When performing an insert to an item x, increase only the minimal counters. When performing lookup query, return the value of the minimal counters. For insertion on r occurrences of x this method can be executed iteratively.

The challenge of storing the SBF: while the dta structure implmentation of the original BF is a simbple bit-vector, the implemntation of the SBF consists of embedding the counteirs in ther binary representation, consecutively in a base array of sine N bits. In the static case the counters are paced without any gap between them, totaling N bits, whereas to support dynamic changes we add some slack bits between counters. This representation introduces a challenge in executing the lookup operations.

Experiment1: compare two algorithms when testing membership: minimum selection MS and minimal increase MI. MI has minus error rate than MS, while in the paper they are equal. Can be because of his implementation.

Experiment2: same test with different number of hash functions. MS and MI errors follow the same curve of variance, from high error to low and then it grows again at the end.

Conclusions: great extension, efficinet maintenance, application for streaming data, dynamic, etc.