CS 229r: Algorithms for Big Data Fall 2013

Lecture 23 — November 19, 2013

Prof. Jelani Nelson *Scribe: Thomas Steinke*

1 Overview

Today we will finish cache-oblivious algorithms and begin covering MapReduce.

2 Cache-Oblivious Lookahead Array (COLA)

The Cache-Oblivious Lookahead Array (COLA) [BFCF$^+$07] is a cache-oblivious version of the buffered repository tree from the last lecture.

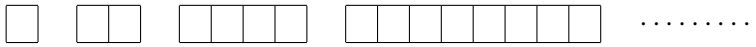
Recall that the buffered repository tree is a 2-3-4 tree where each node has a buffer of size B. It achieves $O(\log N)$ query time and $O(\frac{1}{B}\log N)$ amortized update time.

In problem 1 of problem set 9, we saw how to obtain $O(\frac{1}{\varepsilon}\log_B N)$ query time and $O(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N)$ update time for $\varepsilon \in [\frac{1}{\log B}, 1]$ using a tree with branching factor B^ε and size $\Theta(B)$ buffers. This is provably optimal [BF03].

Today we will obtain the same performance as the buffered repository tree in a cache-oblivious setting. Note that we can obtain the same performance as the B^ε tree in the cache oblivious setting [BDF$^+$10], but we will not discuss this today.

2.1 Basic Setup

The basic COLA datastructure is a series of arrays of exponentially growning size. The i^{th} array stores 2^i elements in sorted order. The arrays are stored sequentially.



Each array is either full or empty. If N items are stored, then the i^{th} array is full if and only if the i^{th} bit in the binary representation of N is 1.

Queries To query the datastructure we perform a binary search in each array. This takes $\log(N/B)$ IOs per array. However, the first $\log(B)$ arrays can be searched in $O(1)$ IOs. This means a query can be executed with $\log(N/B)\cdot(\log(N)-\log(B)) = (\log(N/B))^2$ IOs.

Insertions Insertion proceeds as follows. If the first array is empty, we insert into it. If not, we empty the first array and insert both elements into the next array if that is empty. If the second

array is full, we empty it and try inserting all four elements into the third array and so on. In the end, we fill the first empty array by merging the insert with all preceeding arrays.

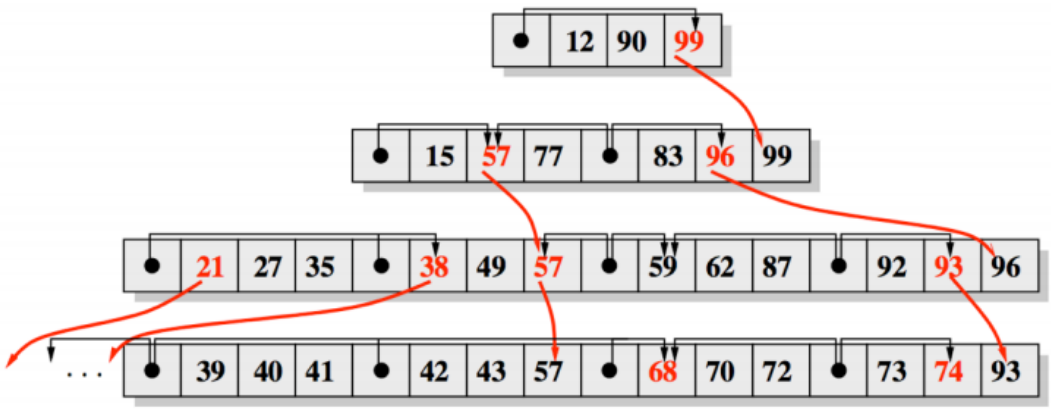
For now, assume $M/B \gtrsim \log(N)$. The cost of a merge is $O(\text{number of items merged}/B)$ IOs. Each item participates in at most $\log N - \log B$ non-free merges. (We don't count merges involving the first $O(\log B)$ arrays as these can happen in memory.) If we imagine each item being given $O(\log(N/B)/B)$ 'credits' upon insertion and decucting $O(1/B)$ credits when it participates in a merge, we see that the total number of IOs is $O\left(\frac{N}{B}\log\left(\frac{N}{B}\right)\right)$. Thus the amortized cost of an insert is $O(\log(N/B)/B)$.

Removing the Assumption We assumed that $M/B \gtrsim \log(N)$. This is necessary for merge to run efficiently. This can be removed for a constant factor increase in space usage. The idea is to break up the merging of $\log(N)$ arrays into pairwise merges. We start by merging the inserted item with the first array into a buffer. Then merge the second array with the buffer into another buffer and so on until everything has been merged.

2.2 Fractional Cascading

Queries in the basic COLA datastructure are very expensive. We will speed them up with "fractional cascading" [CG86]. The idea is that we can save on the binary search in an array by using information from the binary search in the previous array. In particular, we will 'sprinkle' lookahead pointers in each array, which will help us search the next array.

Every eighth item in array i is *duplicated* in array $i-1$, with pointers from the duplicate to the original. We call these entries "lookahead spots". In addition, every fourth item in array $i-1$ will have two pointers to the nearest lookahead spots in the array, one for each side. (If there is none, we have a null pointer.)



Now the idea is that, if the search in array $i-1$ fails, it still allows us to narrow down the locations to search in array i. In particular, if the search in array $i-1$ finds a location where the key k *should* be, then it finds the nearest lookahead spots a and b on both sides of this location (using the pointers in every fourth cell). These lookahead spots then identify an interval in array i. We need only search this inverval, which has length at most eight. Thus executing a query reduces to $O(\log(N/B))$ IOs.

Assertion/Exercise Insertions can still be completed with amortized $O(\log(N/B)/B)$ IOs.

(This technique is apparently used in practice by Tokutek. Fast insertions are important.)

3 Dynamic Cache-Oblivious B-Trees

Now we will show how to obtain even faster queries using dynamic cache-oblivious B-tress [BCR02] (see also [BDFC05]). In particular, we will be able to perform queries with $O(\log_B N)$ IOs in the worst case and updates with $O(\log_B N)$ amortized IOs.

The idea is to have a B-tree of depth $\log(\log(N))$, where the branching factor varies from level to level. Define the volume of a node to be the size of the subtree rooted at it. Each node at layer i has volume approximately 2^{2^i}. The branching factor at the root is approximately $\sqrt{n}$ at the next layer it is $\sqrt[4]{n}$. The elements in each node are stored in a static cache-oblivious B-tree (van Emde Boas layout; see Lecture 22 §2.4).

Queries Finding the correct child to search in a node at layer i requires $O(\log_B(2^{2^i})) = O(2^i/\log(B))$ IOs. Summing this over $i \in [O(\log(\log(N)))]$ gives $O(\log_B(N) + \log(\log(N)))$ IOs per query (each layer i requires at least one query).

Insertions As usual, insertions happen at the leaves and splits propagate upwards. The amortized cost of an insertion is the cost of a query (to find where to insert) plus the amortized cost of propagating splits up the tree. Note that we need to rebuild the static B-tree whenever we split nodes. A split at level i happens with frequency approximately 2^{-2^i}. The amortized cost of an insertion is also $O(\log_B(N) + \log(\log(N)))$ IOs.

Removing $\log\log N$ Term We can reduce both queries and insertions to $O(\log_B(N))$ IOs by requiring that each subtree is stored in contiguous memory. This means that when operations happen at levels below $\log(\log(B))$, it only takes $O(1)$ IOs.

Space Usage A node at level i has branching factor at most $2 \cdot (2^{2^{i-1}} + 2^{2^{i-2}} + 2)$. So the total space usage is

$$\prod_{i=0}^{\log\log N} 2 \cdot (2^{2^{i-1}} + 2^{2^{i-2}} + 2) = O(2^i \cdot 2^{2^i}) = O(N \log N).$$

This is suboptimal. We can remove the $\log N$ factor by replacing each leaf with an array of $O(\log N)$ items (rather than a single item). Then we have a tree on $N/\log N$ leaves, which uses space $O(N)$.

4 MapReduce

MapReduce is a system developed at Google [DG08] for massive parallel processing. There are open-source variants like Apache's Hadoop. It is used by Yahoo, Facebook, and others. Although it has also received criticism [DS08].

MapReduce is intended to be used in a setting where there are lots of machines, each with some limited memory and processing power and a large computational task that can be decomposed as follows.

The input to MapReduce is key-value pairs $(k_1, v_1) \cdots (k_n, v_n)$, which are processed in three phases:

1. **Map** gets a key-value pair (k_i, v_i), performs some computation, and outputs a list of new key-value pairs $(k'_{i,1}, v'_{i,1}) \cdots (k'_{i,n'_i}, v'_{i,n'_i})$.
2. **Shuffle** takes all the new key-value pairs and sorts them so that all values with the same key are put together.
3. **Reduce** is given *one* key and *all* the corresponding values $(k'_j, (v'_{j,1} \cdots v'_{j,n'_j}))$. It performs some computation and produces the final output.

Note that MapReduce can have several Map/Shuffle/Reduce rounds. That is, the output of Reduce on the first round is given to Map on the second round and so on.

For algorithms to suit the MapReduce system, we want algorithms with

- small memory,
- few machines,
- few rounds, and
- small total work.

References

[BCR02] Michael A. Bender, Richard Cole, and Rajeev Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 195–207, 2002.

[BDF+10] Gerth Stølting Brodal, Erik D Demaine, Jeremy T Fineman, John Iacono, Stefan Langerman, and J Ian Munro. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1448–1456. Society for Industrial and Applied Mathematics, 2010.

[BDFC05] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[BF03] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 307–315. ACM, 2003.

[BFCF+07] Michael A Bender, Martin Farach-Colton, Jeremy T Fineman, Yonatan R Fogel, Bradley C Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 81–92. ACM, 2007.

[CG86] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DS08] David J. DeWitt and Michael Stonebraker. Mapreduce: A major step backwards, 2008. http://homes.cs.washington.edu/∼billhowe/mapreduce_a_major_step_backwards.html.