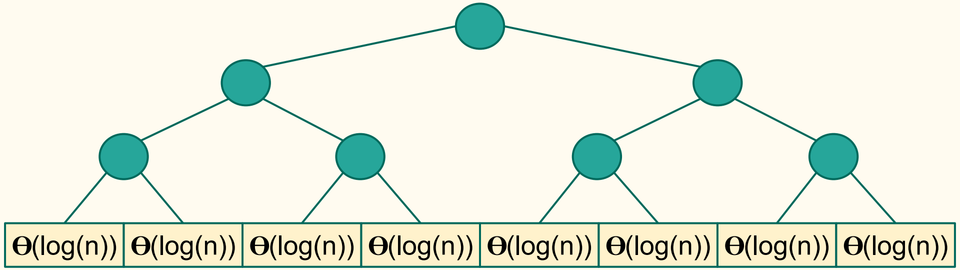
Recent Improvements to Packed Memory Arrays

Brian Wheatman

Packed Memory Arrays

PMA is an ordered dictionary data structure built on a contiguous array [Itai+ 81]

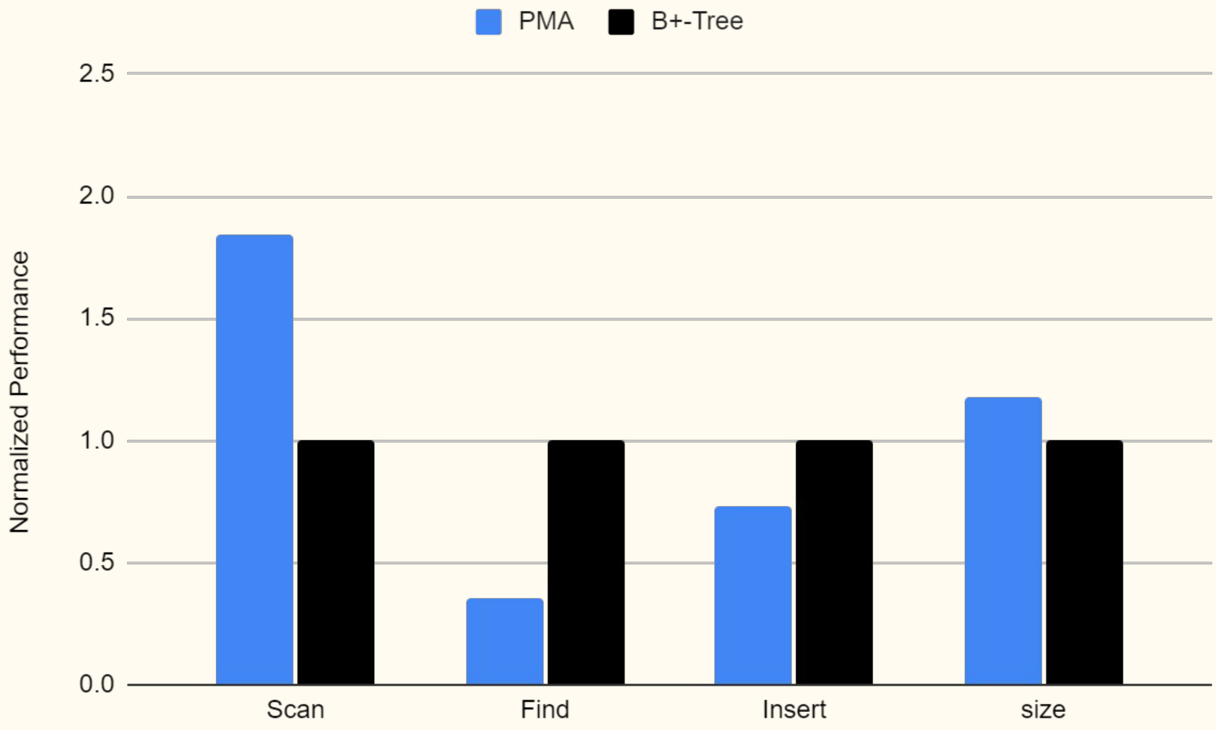
- Stores N elements in sorted order in O(N) space
- Inserts and deletes in amortized $O(\log^2(N))$
- Point queries in $O(\log(N))$
- Extremely efficient scans due to contiguous memory



PMA theoretical behavior

Operation	Binary Tree	B-Tree	PMA
Insert	$O(\log(n))$	$O(\log_B(n))$	$O(\log^2(n))$
Contains	$O(\log(n))$	$O(\log_B(n))$	$O(\log(n))$
Scan	$O(n)$	$O(n/B)$	$O(n/B)$
Size	$O(n)$	$O(n)$	$O(n)$

Baseline PMA Performance Summary



PMA Empirical Behavior

Excellent scans, 2x faster than B-Trees

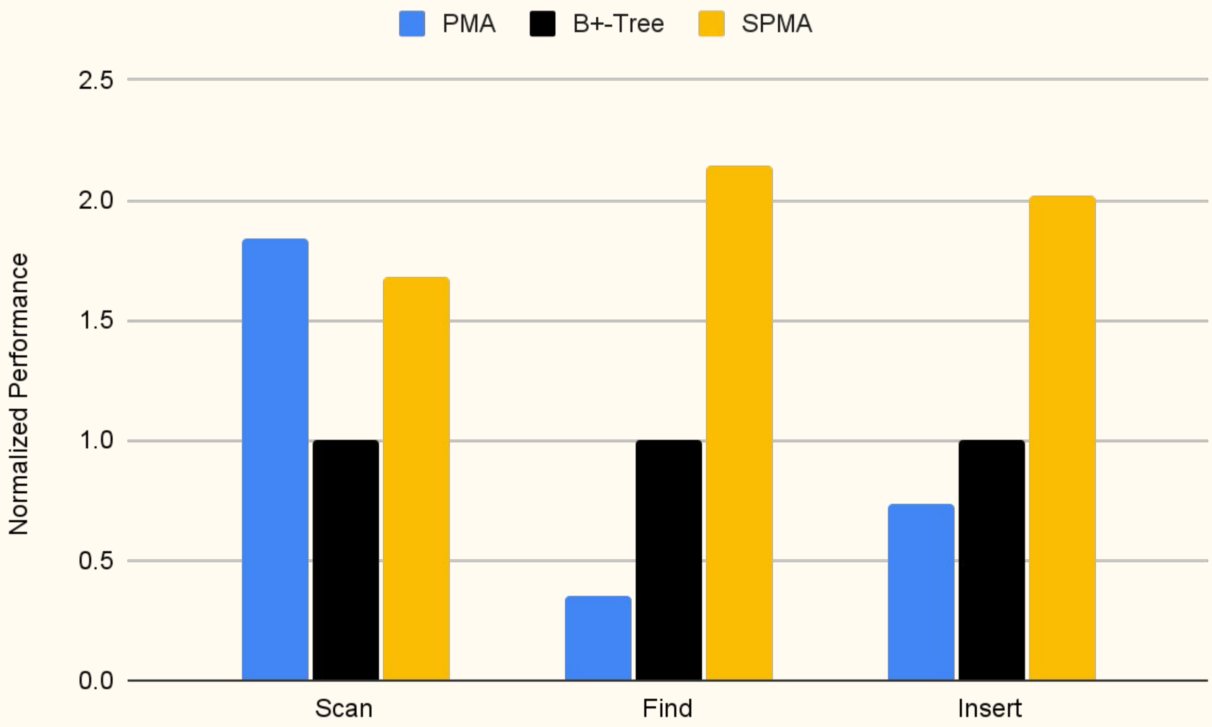
Three issues

1. Bigger than B-Trees
2. Slower point queries
3. Slower insertions

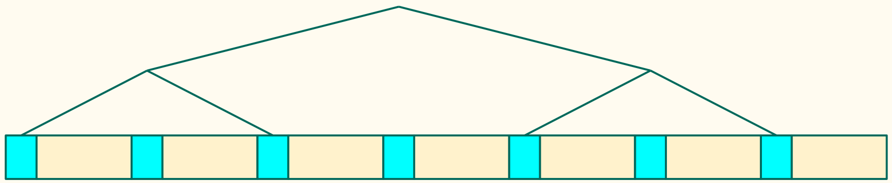
We are going to fix these in three steps

1. First speed up point queries, which will improve serial inserts
2. Add parallel batch inserts to speed up inserts even more
3. Decrease the size

Search Optimized PMA (SPMA) Performance



Searching in a PMA

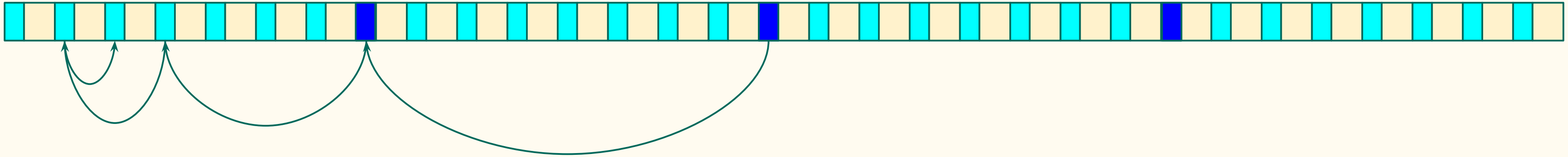


Perform a binary search on the first element of each leaf (leaf head)

Search inside the leaf for the element

Searching ends up being a major part of insert as well

Searching in a PMA



Initially each step of the binary search is a cache miss

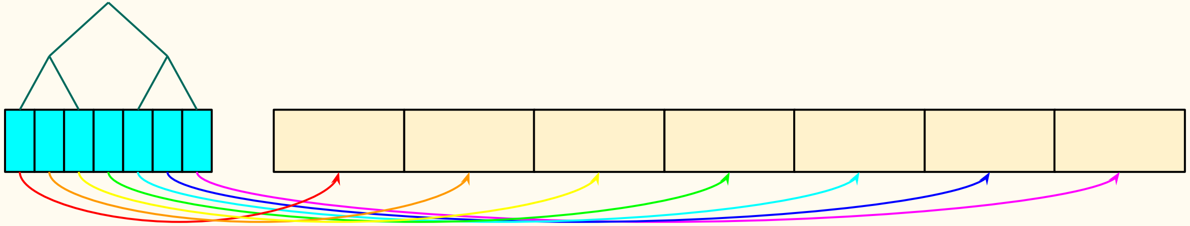
At equilibrium the top of the possible binary search tree will be in cache

However, for cache line of size B only the first element in useful

So for a cache of size M we get M/B useful elements

So the first log(M/B) steps of the search will likely be in cache

Search Optimized PMA (SPMA)

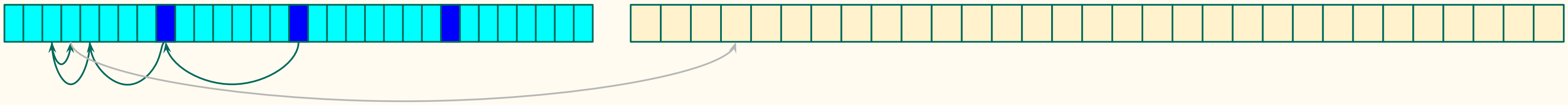


Move the leaf heads to a separate array to enable faster queries

Maintain a mapping between the two structures

For more information see Optimizing Search Layouts in Packed Memory Arrays

Searching in a SPMA - Linear



On the first search the last few steps of the search are now in the same cache line

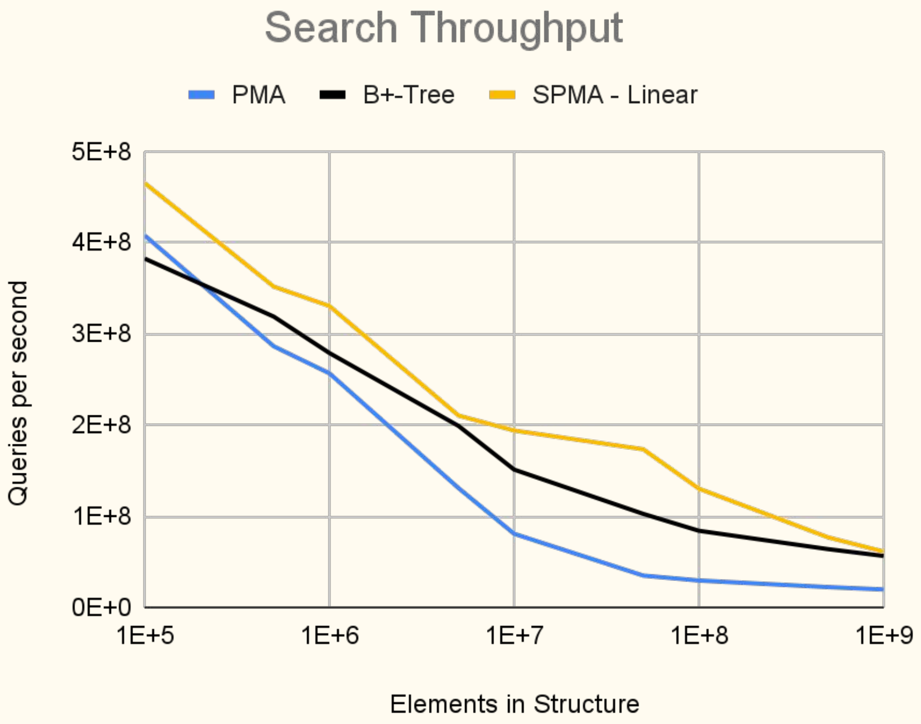
So we save log(B) steps of the search at the bottom

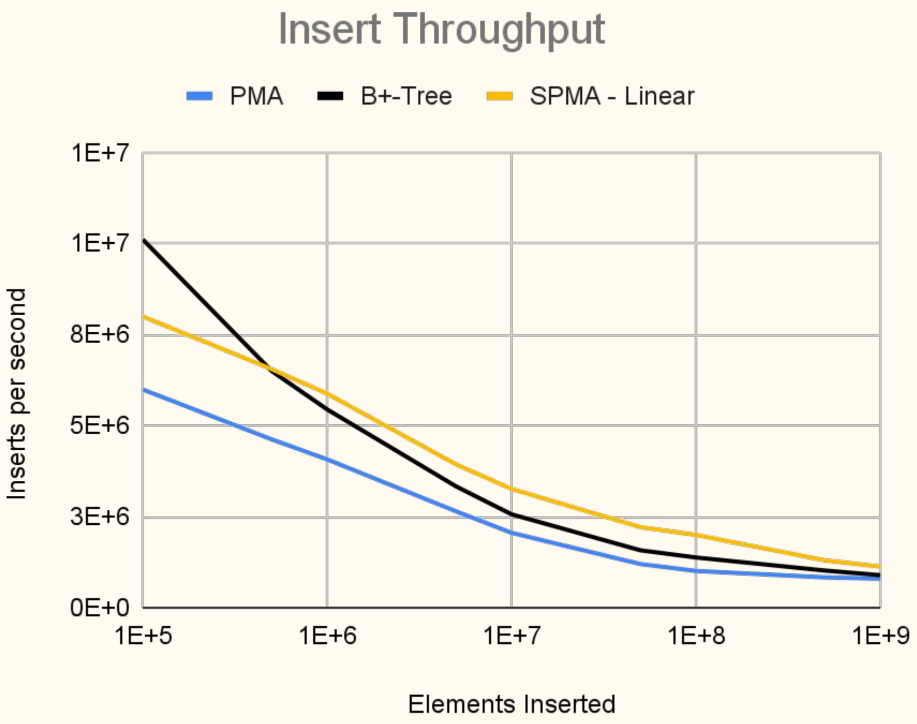
But we need to pay to load the data

At equilibrium the top of the possible binary search tree will be in cache

So the first log(M/B) steps and the last log(B) of the search will be in cache

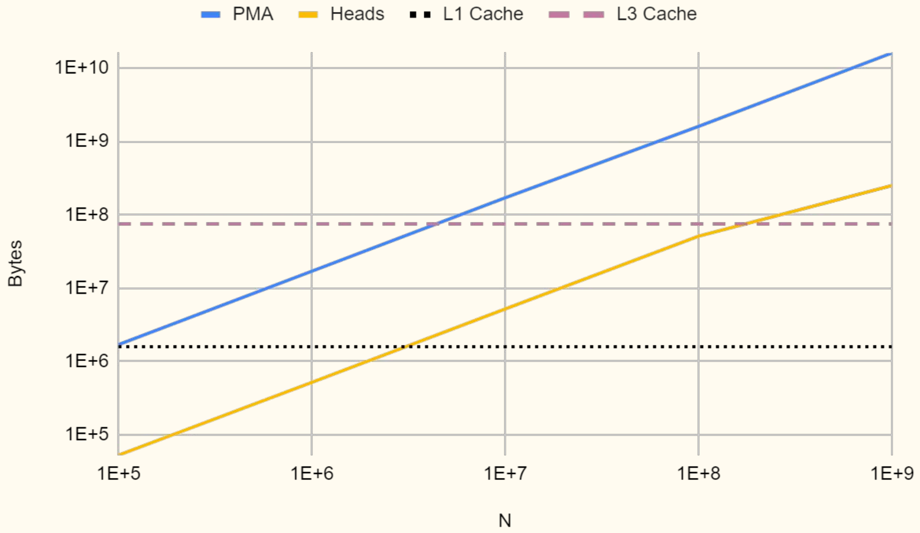
SPMA - Linear Performance



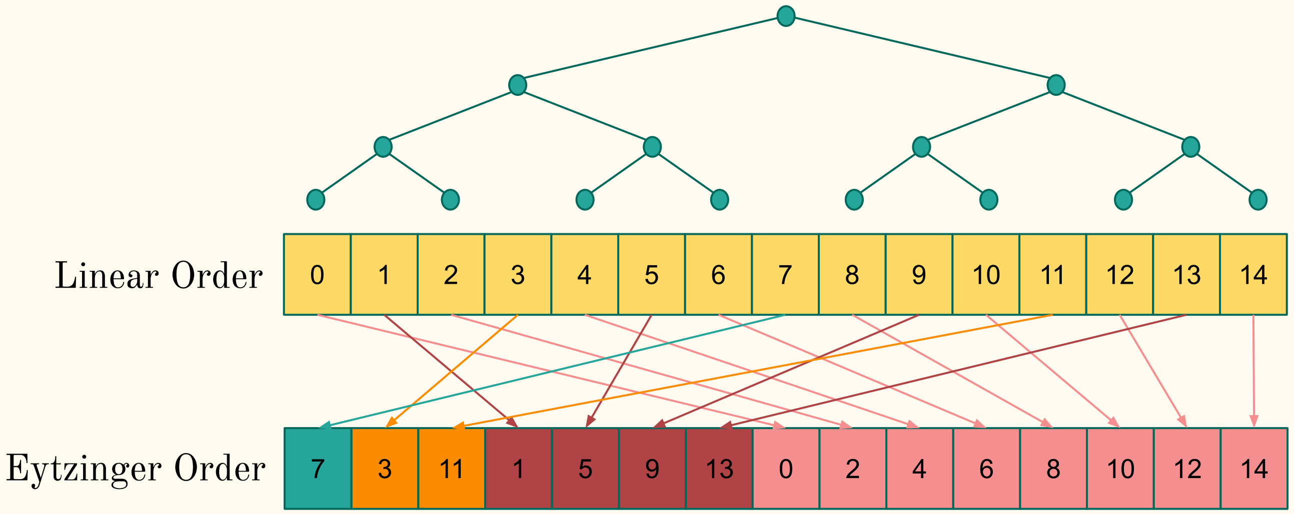


SPMA Memory Usage

The smaller head structure can also remain in cache for larger structures



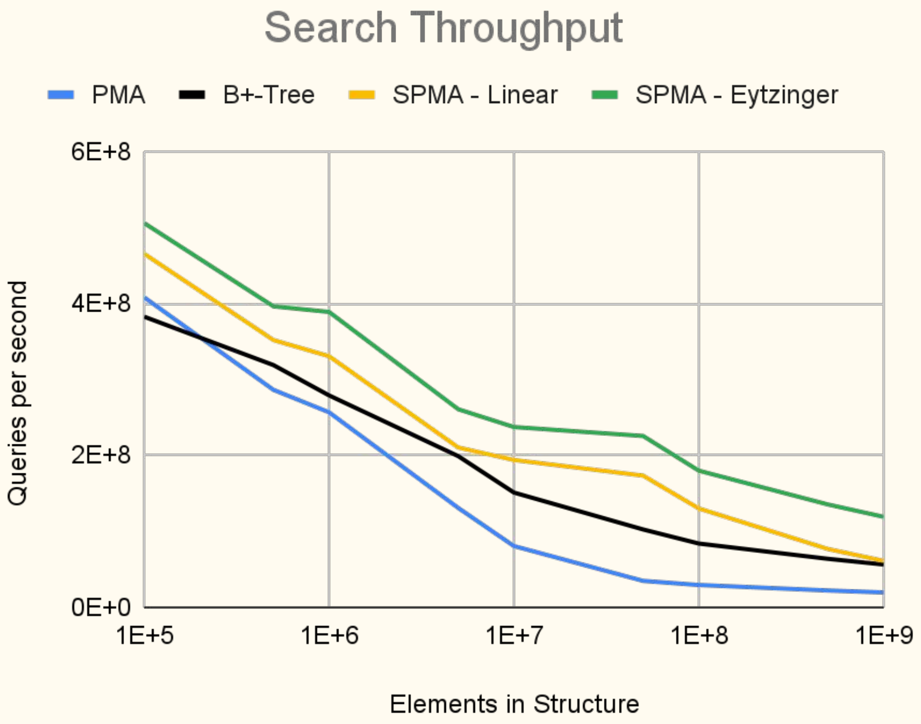
SPMA - Eytzinger [Eytzinger 1590]

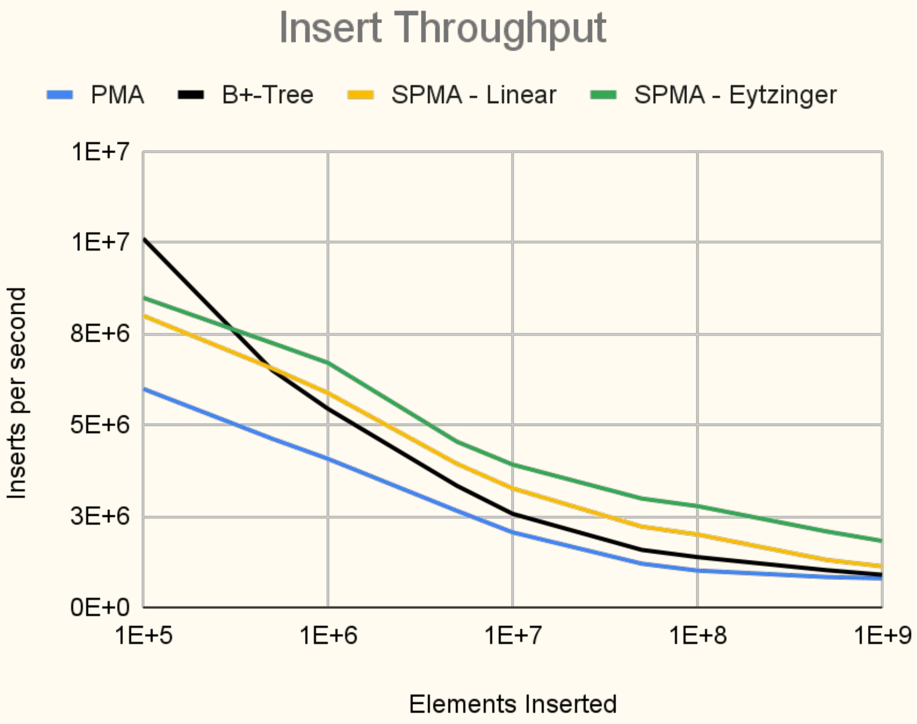


This makes the unpredictable jumps through the array nearly perfectly predictable

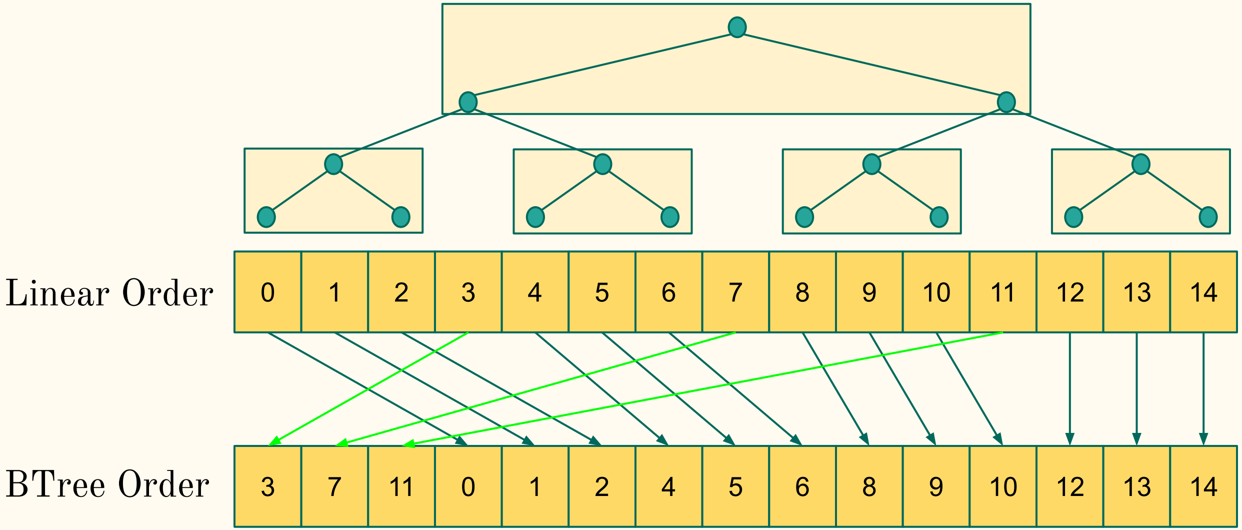
After cell i, the options are 2i or 2i+1, this allows the prefetcher to collect the data

SPMA - Eytzinger Performance





SPMA - BTree [Khuong 17]



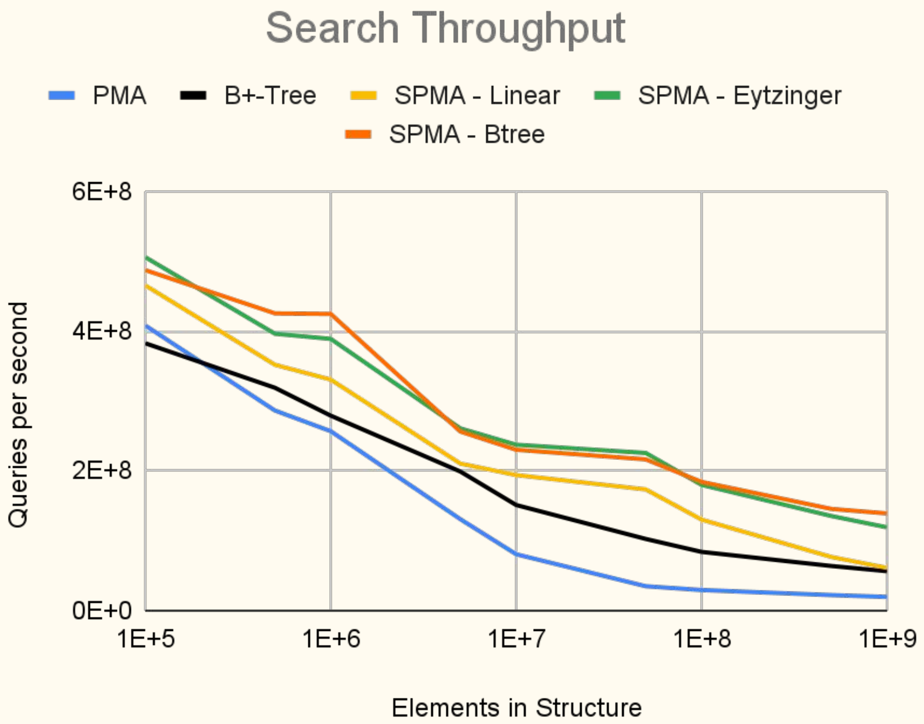
Group multiple levels of what will be part of the main binary search tree together

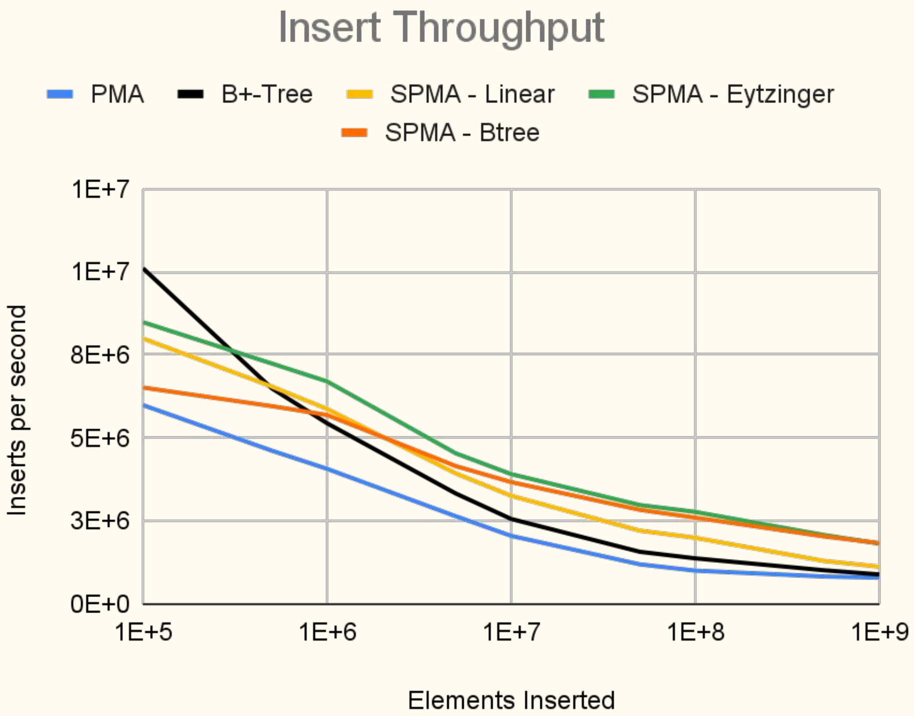
Each cache line fetch grabs log(B) levels, dividing the total cost by log(B)

Asymptotic performance of SPMAs

Data Structure	Search	Insert	Range Query
PMA	$O(\lg(n/M))$	$O(\lg(n/M) + \lg^2(n)/B)$	$O(\lg(n/M) + k/B)$
SPMA-Linear	$O(\lg(n/MB))$	$O(\lg(n/MB) + \lg^2(n)/B)$	$O(\lg(n/MB) + k/B)$
SPMA-Eytzinger	$O(\lg(n/M))$	$O(\lg(n/M) + \lg^2(n)/B)$	$O(\lg(n/M) + k/B)$
SPMA-Btree	$O(\log_B(n/M))$	$O(\log_B(n/M) + \lg^2(n)/B)$	$O(\log_B(n/M) + k/B)$
B-Tree	$O(\log_B(n/M))$	$O(\log_B(n/M))$	$O(\log_B(n/M) + k/B)$

SPMA - BTree Performance





Parallel Batch Inserts - Overview

1. Batch merge
    a. Merge elements into the PMA
2. Counting nodes
    a. Determine which regions of the PMA need to be rebalanced
3. Redistribute nodes
    a. Rebalance the required regions

For more information see CPMA: An efficient batch-parallel compressed set without pointers

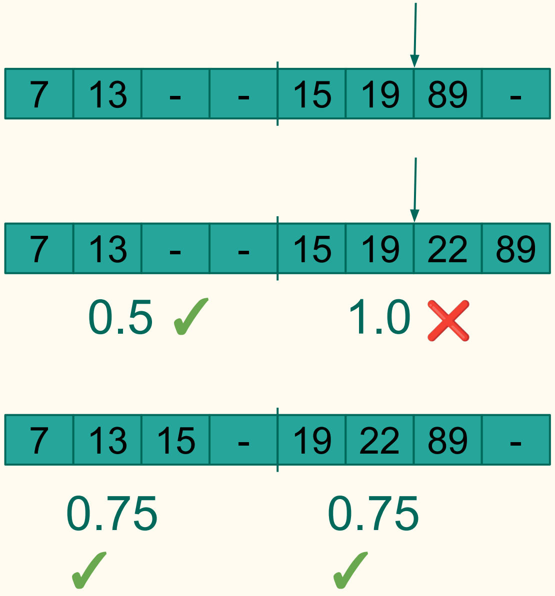
Review Inserts in Packed Memory Arrays

(1) Search (22)

(2) Place (22)

(3) Count

(4) Redistribute

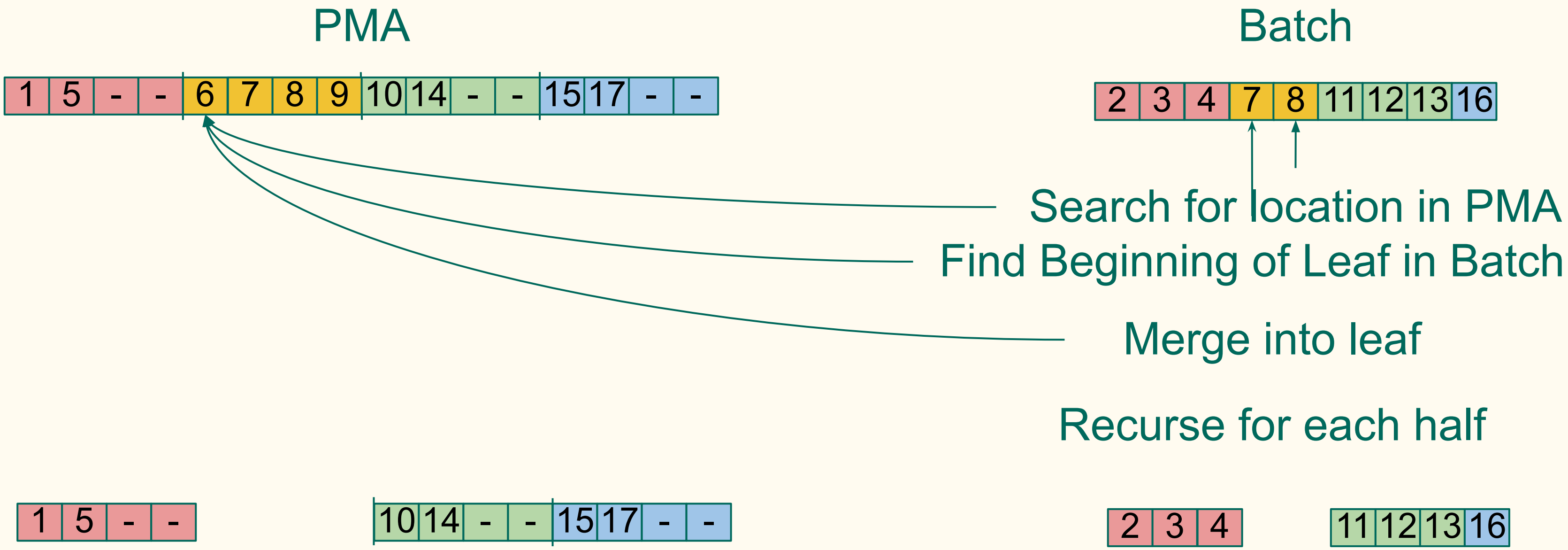


An insert (or delete) is broken into 4 steps

1. Search for which leaf the element will go into
2. Place the element into the leaf shifting around nearby elements
3. Count to determine the region to redistribute
4. Redistribute the required region

Parallel Batch Inserts - Batch Merge

Merge the elements in the batch into correct positions in the PMA



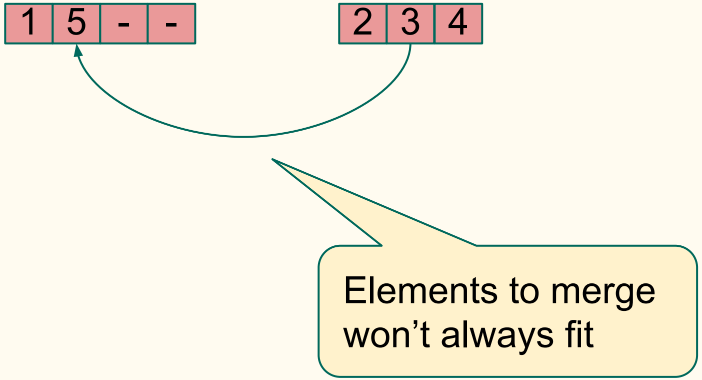
Save work by minimizing the number of searches and the sizes of the searches

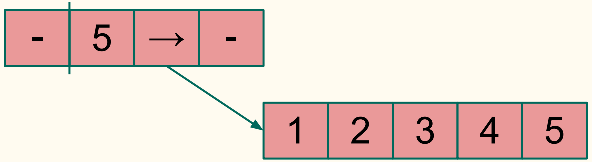
Batch Merge - overflowing leaves

Sometimes we need to merge more elements into a leaf than can fit

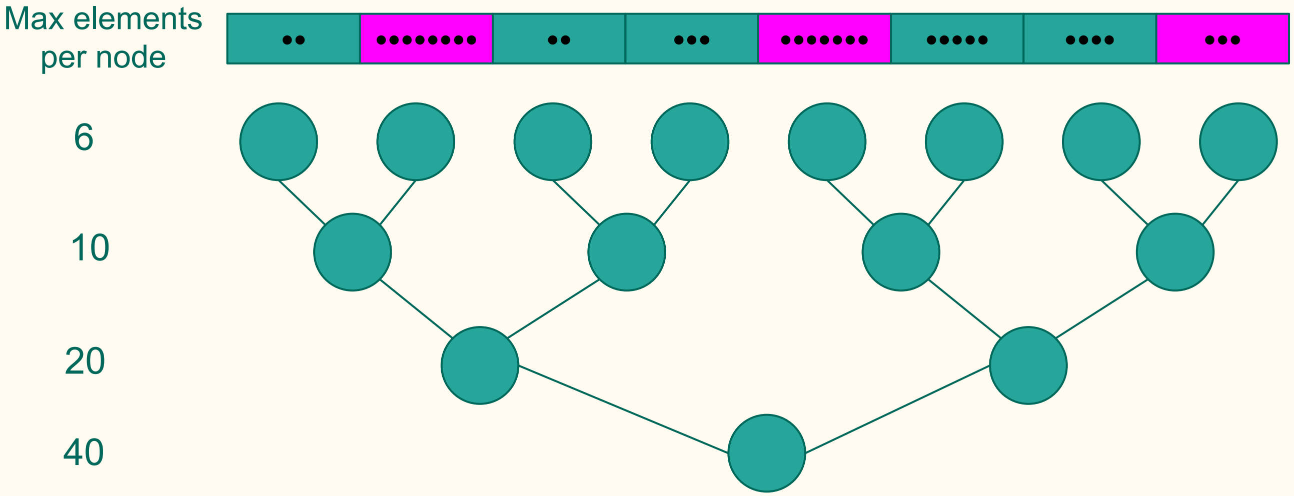
With parallel modifications going on we cannot merge into neighboring leaves

Temporarily store the elements out of place with a pointer and a count





Finding overfull PMA nodes

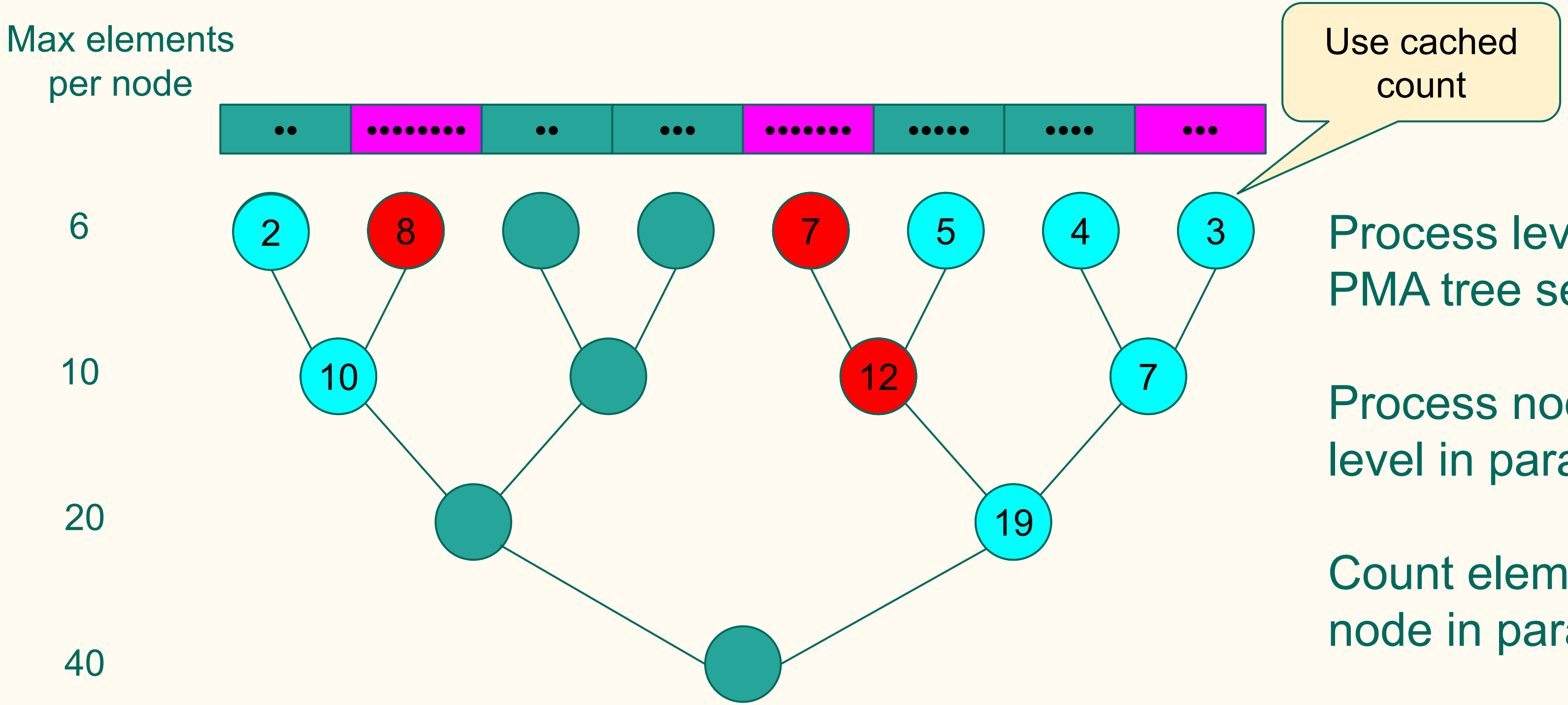


Walk up the PMA tree from each modified leaf counting nodes until a satisfied node is found

Serially this is trivially optimally done by simply caching counts

In parallel nodes can be counted multiple times before they are cached

Parallel Batch Inserts - Parallel Counting



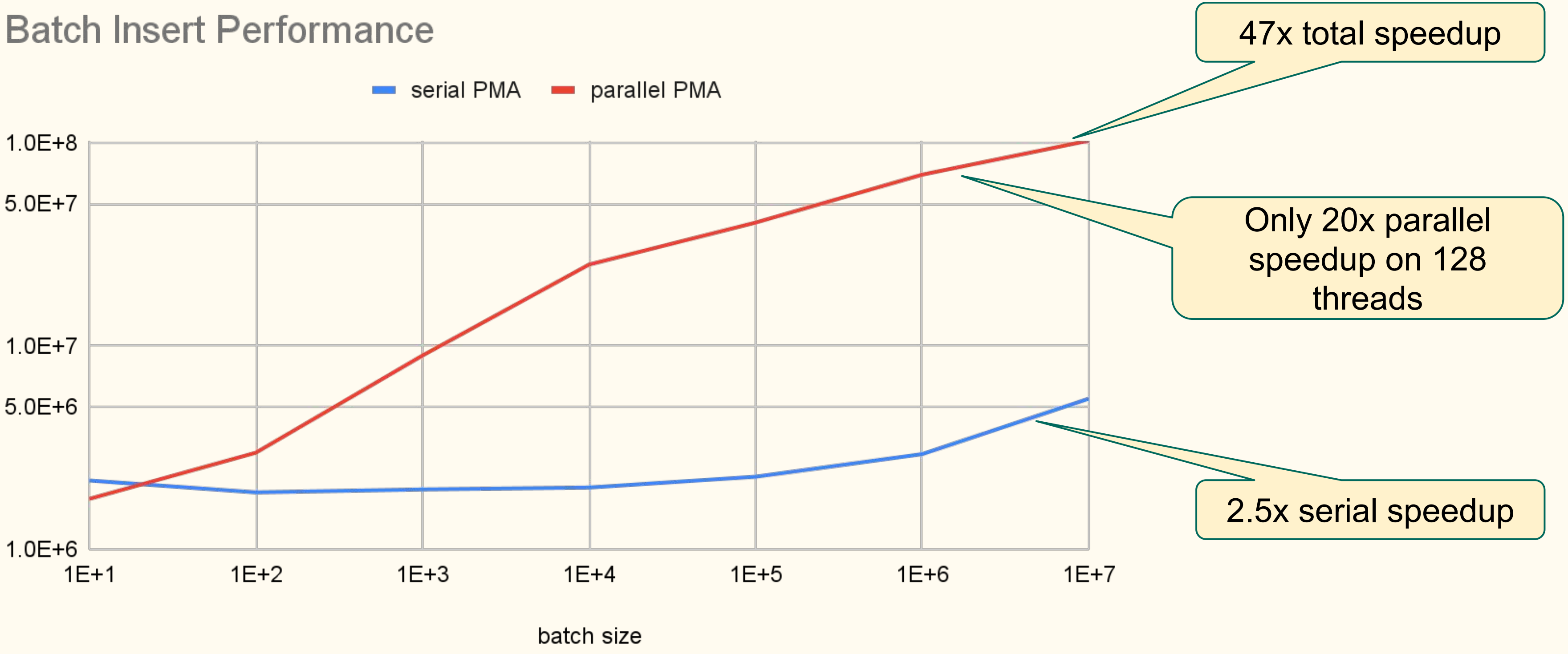
Process levels of the PMA tree serially

Process nodes of a level in parallel

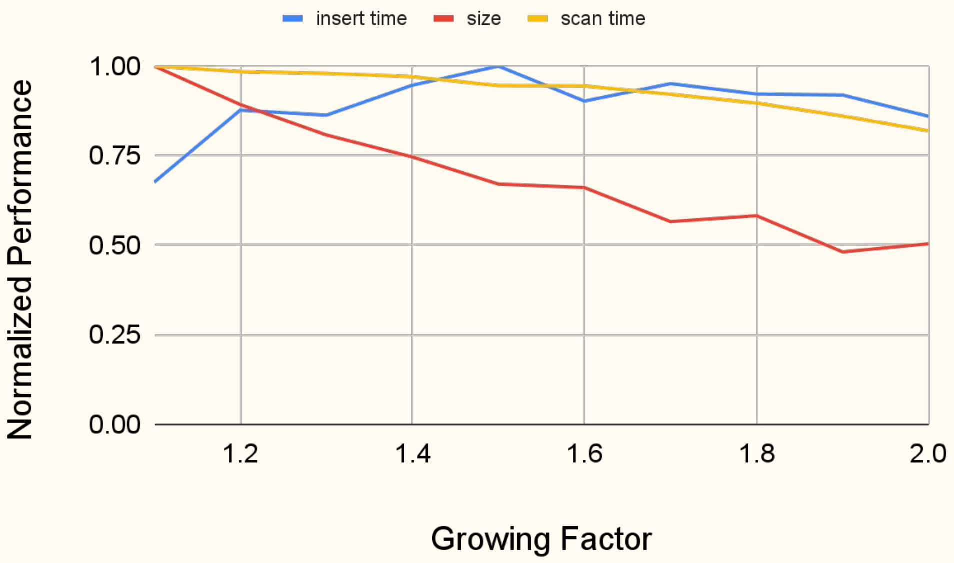
Count elements of a node in parallel

A **work-efficient counting algorithm** for PMA batch updates that counts every necessary cell exactly once.

Batch Insert Performance



PMA Space Usage

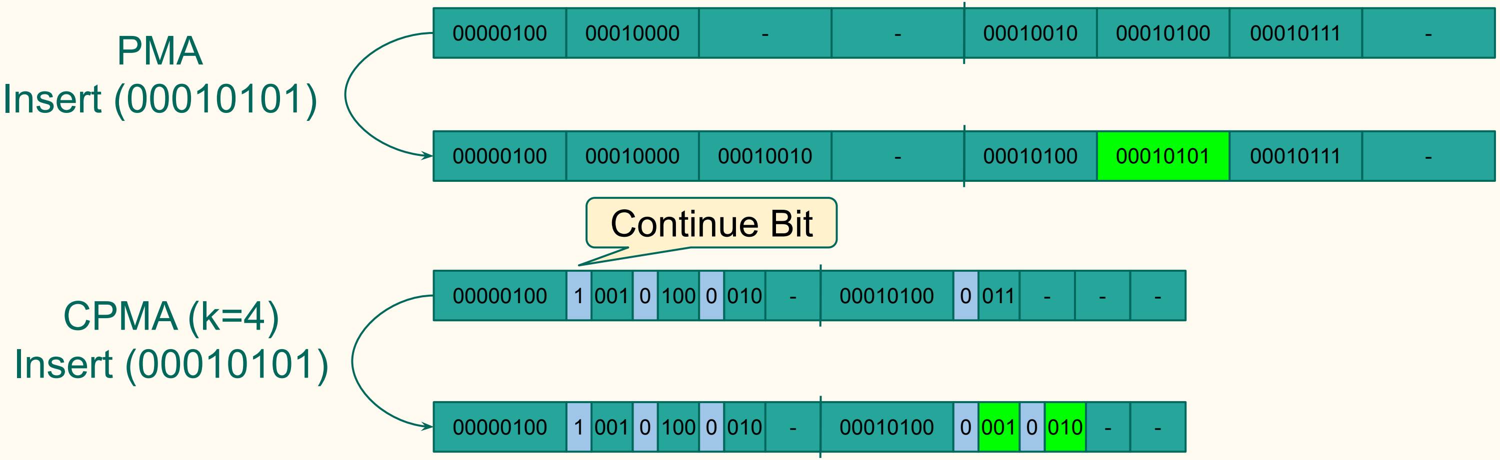


We find we can reduce the growing factor to 1.2 to improve the space usage and scan time, and match the insert performance at 2x growing factor

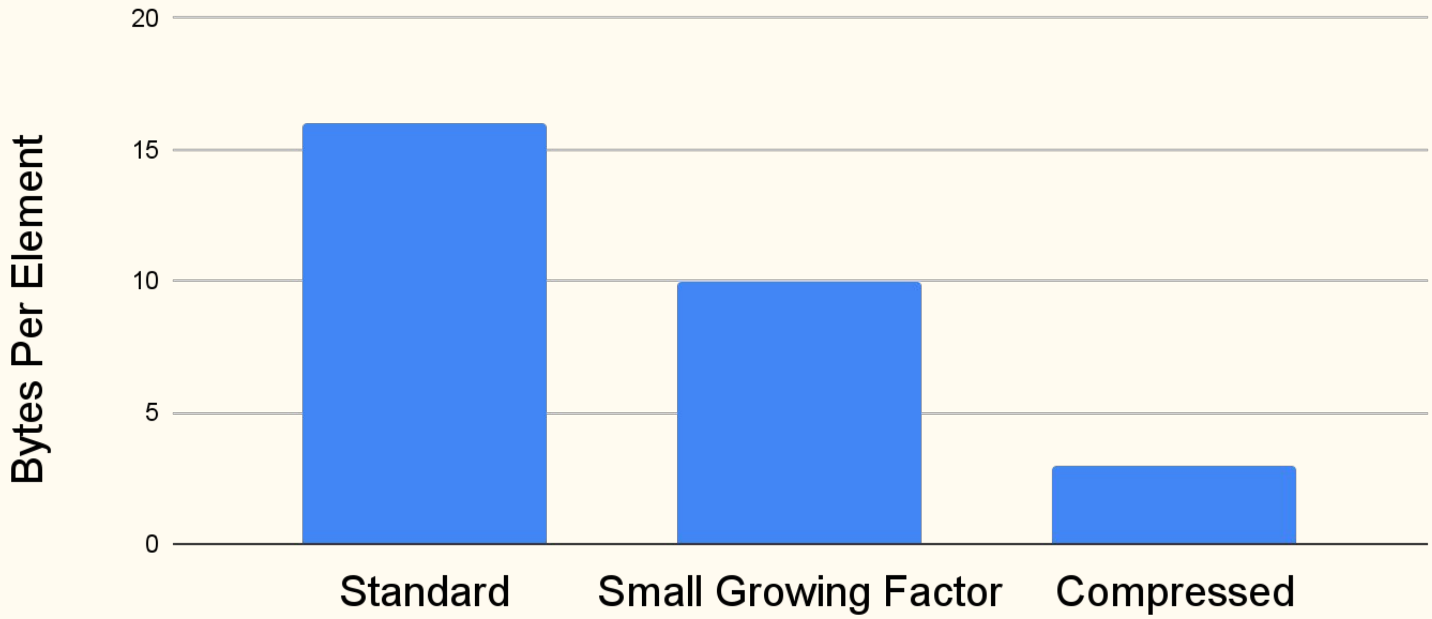
Compression

Compression helps minimize memory footprint to maximize use of available memory bandwidth

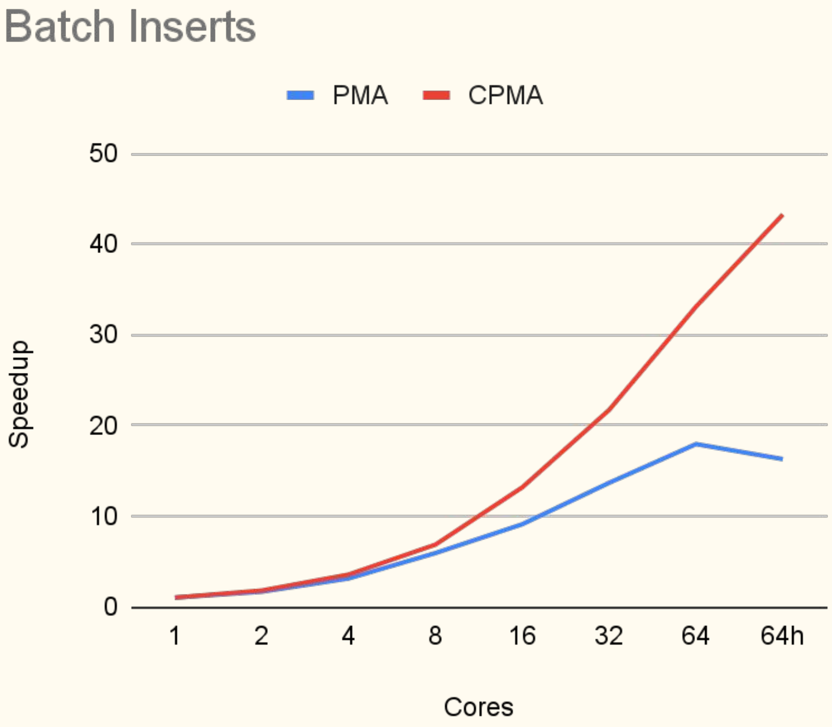
Store leaf heads uncompressed, delta compress the rest of the leaf

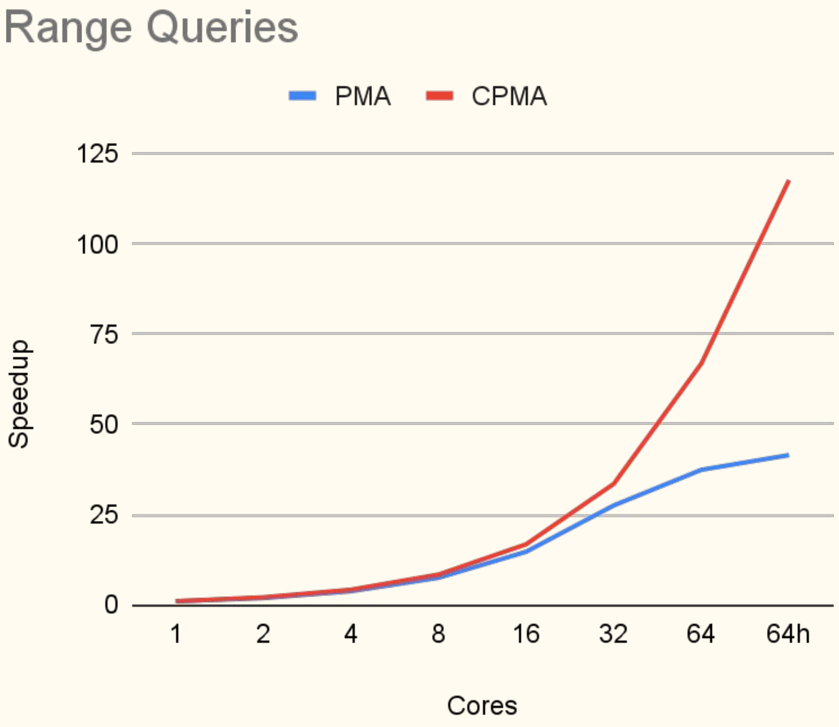


Size

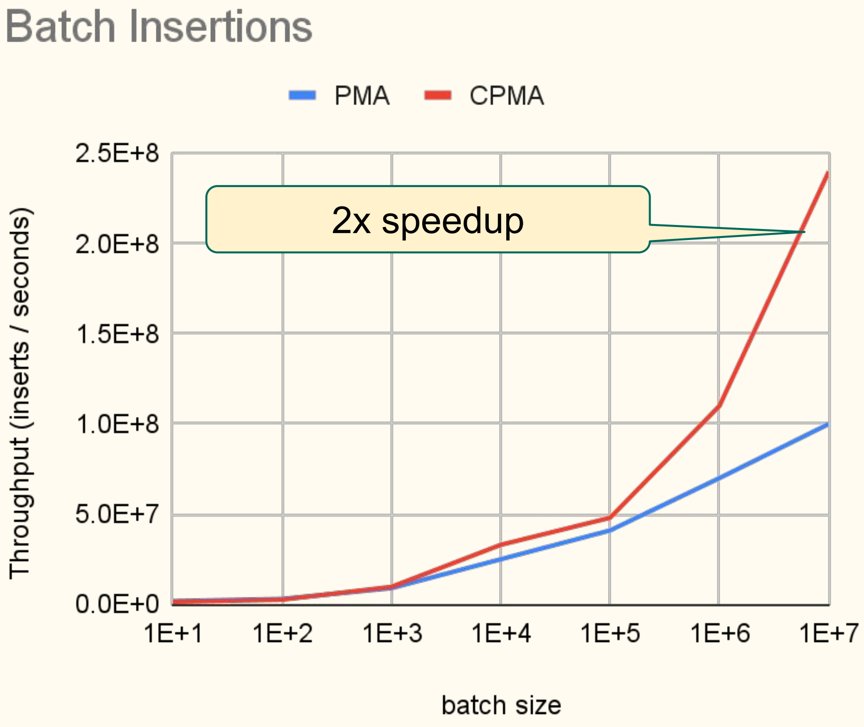

20
15
10
5
0
Bytes Per Element
Standard
Small Growing Factor
Compressed

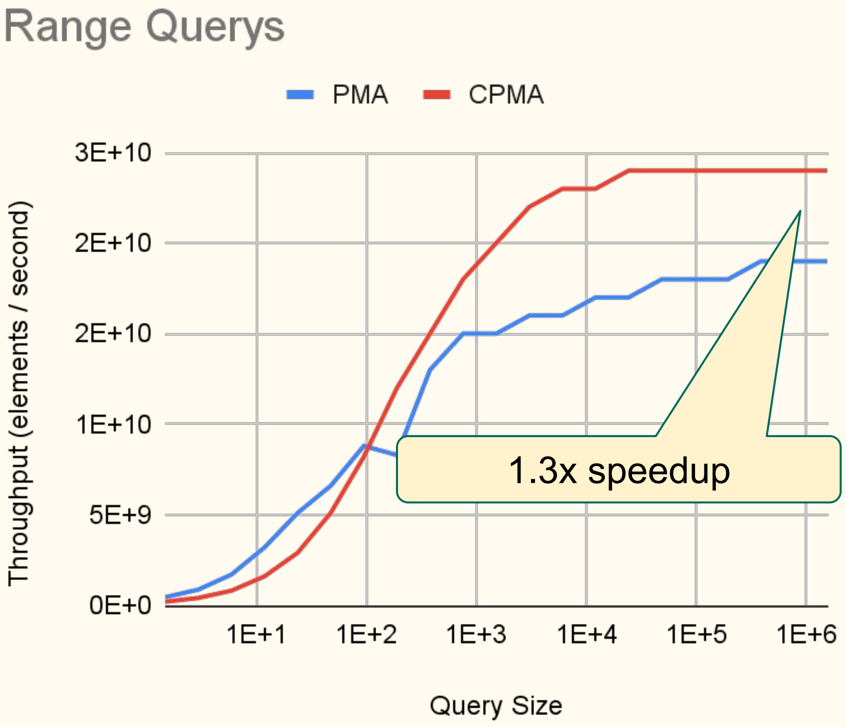
Compression improves Scalability





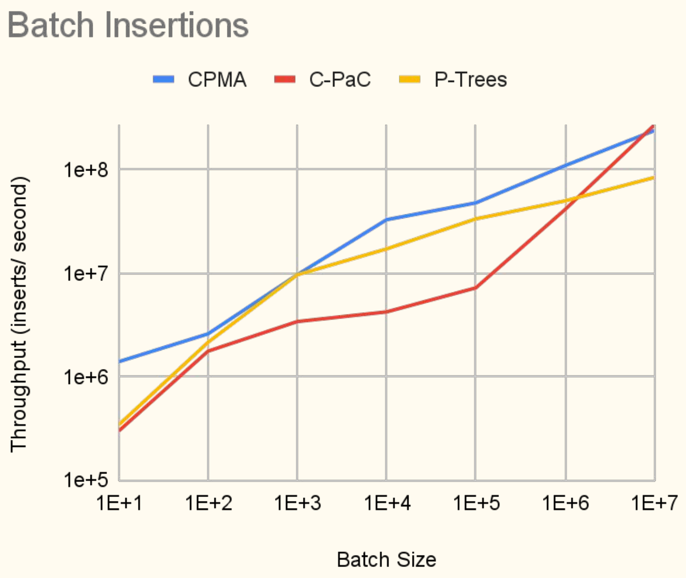
Compression improves throughput

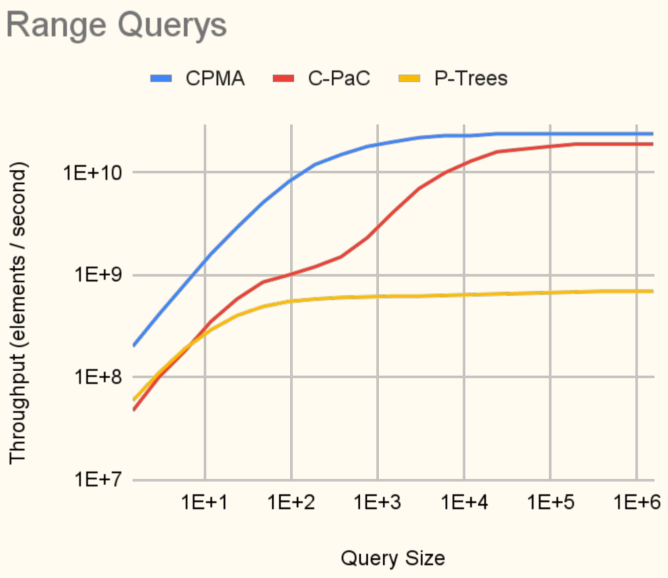




Compared to tree based batch parallel sets

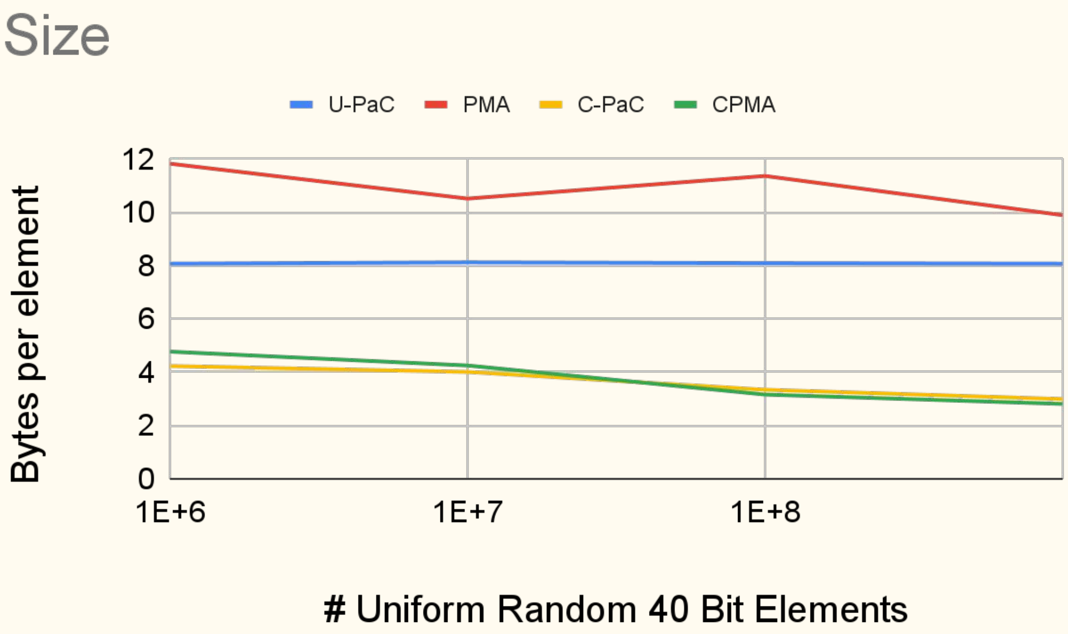
We compare to compressed cache optimized trees (C-PaC) and binary trees (P-Trees) both with parallel batch updates





While PMA is a bigger than a uncompressed cache optimized tree

The CPMA is the same size as a compressed cache optimized tree



Packed Memory Arrays

Packed Memory Arrays are an excellent choice for ordered sets

By optimizing for the memory system we have an improved structure which can outperform existing structures across a wide range of benchmarks

https://github.com/wheatman/Packed-Memory-Array

Backup

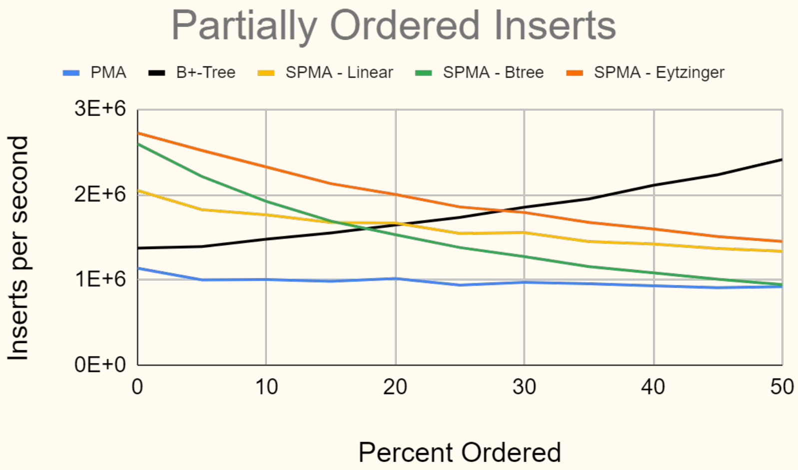
Compression Ratio

# Elts	U-PaC	PMA	PMA / U-PaC	C-PaC	CPMA	CPMA / C-PaC	CPMA / PMA
1E6	8.07	11.82	1.46	4.23	4.77	1.13	.40
1E7	8.12	10.51	1.30	4.01	4.25	1.06	.40
1E8	8.09	11.36	1.40	3.34	3.16	.95	.28
1E9	8.07	9.89	1.23	2.99	2.81	.94	.28

Limitations of PMAs

PMAs are designed for throughput, they are (as described) an amortized data structures which can have linear work worst case behaviors

PMAs can have bad input distributions which make updates slower



SPMA partially ordered elements

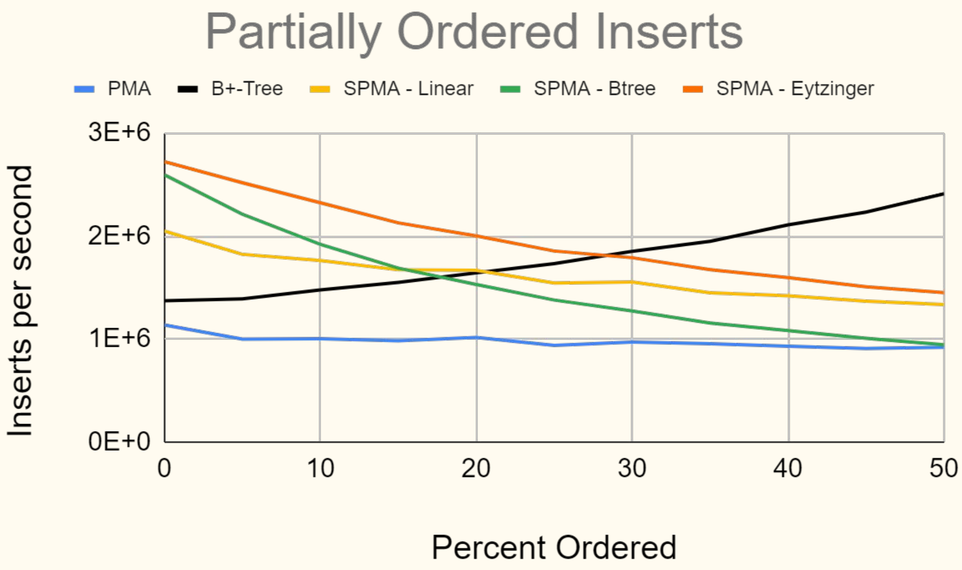
Insert 100 million elements

p chance of being a new minimum

1-p chance of being random

At p = 1 the SPMA are 2-4x slower than random, while the B+-Tree is 6x faster

Worst case B-Tree is 1.5x faster than worst case SPMA



Parallel PMA Updates

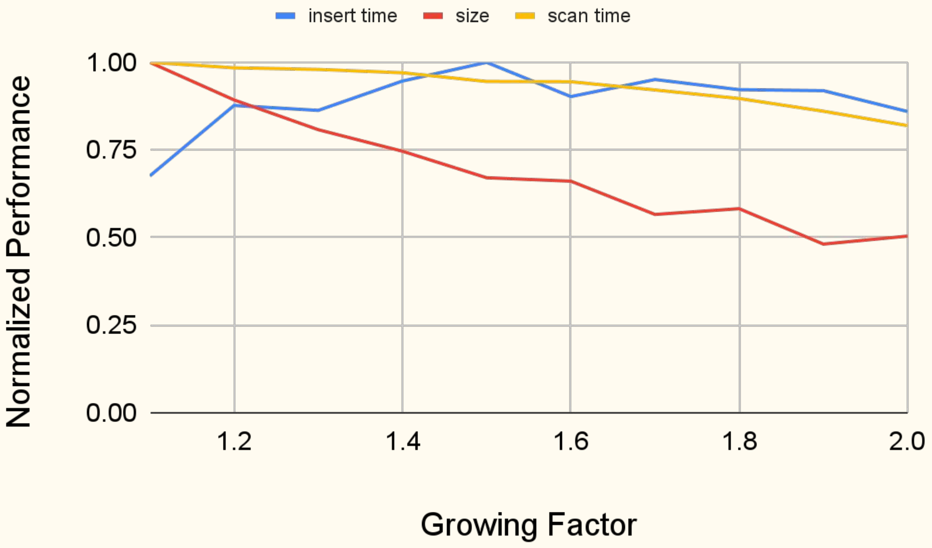
Batch Parallel updates

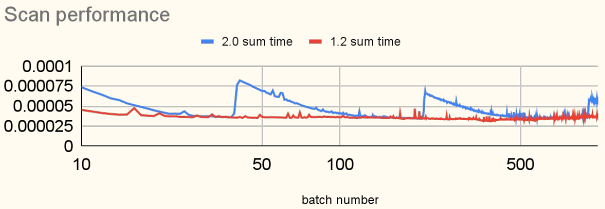
- Supply the data structure with a batch of elements, and it adds all of them
- The data structure internally parallelize the operations
- See CPMA: An efficient batch-parallel compressed set without pointers

Thread-safe concurrent updates

- Different threads can add elements to the structure concurrently
- See A Parallel Packed Memory Array to Store Dynamic Graphs

PMA Space Usage





We find we can reduce the growing factor to 1.2 to improve the space usage and scan time, and match the insert performance at 2x growing factor