BP-tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees

VLDB 2023

Prashant Pandey, University of Utah
Joint work with: Helen Xu (Berkeley Lab), Amanda Li (MIT), Brian Wheatman (Johns Hopkins), Manoj Marneni (Utah)
Dictionary

Queries
- membership
- predecessor/successor
- range queries

Updates
- insertions
- deletions
Balanced search trees
- Insert $O(\log N)$
- Queries $O(\log N)$
External memory model [Aggarwal and Vitter 1988]

- **How computations work:**
  - Data is transferred in blocks between RAM and disk.
  - The number of block transfers dominate the running time.

- **Goal: Minimize number of block transfers**
  - Performance bounds are parameterized by block size $B$, memory size $M$, data size $N$. 
B-tree: a classic indexing data structure

B/B+ Trees [BM72] are used everywhere

- In memory indexing [ZCO+15]
- Databases [K98]
- Filesystems [RBM13]

B-Trees are asymptotically optimal for point operations
B-tree: in external memory model [Brodal and Fagerberg 2003]

B/B\(^+\) Trees [BM72] are used everywhere

- In memory indexing [ZCO+15]
- Databases [K98]
- Filesystems [RBM13]

\[
\begin{align*}
\text{Insert} & \quad O \left( \log_B \frac{N}{M} \right) \text{ I/Os} \\
\text{Search} & \quad O \left( \log_B \frac{N}{M} \right) \text{ I/Os}
\end{align*}
\]
B-trees: in external memory model [Brodal and Fagerberg 2003]

B/B\(^+\) Trees [BM72] are used everywhere

- In memory indexing [ZCO+15]
- Databases [K98]
- Filesystems [RBM13]

\[
\text{Insert, Search } \quad \{ \quad O \left( \log_B \frac{N}{M} \right) \text{ I/Os} \quad \}
\]

\[
\text{Search} \quad \log_B \frac{N}{M} \quad \text{Insert} \quad \log_B \frac{N}{M}
\]

Buffered B-trees
In this talk: point-range tradeoff in in-memory B-trees
Motivation: B-tree point-range tradeoff

They exhibit a tradeoff between point operations (updates/queries) (OLTP) and long range scans (OLAP) as a function of node size.

Long range scans are critical for:
- Real time analytics [PTPH12]
- Graph processing [DBGSS22, PGK21, PWXB21]

Large nodes speed up range scans at the cost of point inserts.
We introduce BP-tree to overcome the traditional tradeoff between point operations and range scans in in-memory B-trees.
Evaluation on YCSB benchmarks

Performance of B-tree, Masstree [MKM2012], OpenBwTree [WPL+2018] and BP-tree on YCSB [CST+10]

We match the performance of point operations and improve range queries by 2x
B-tree insert/range scan trade-off

Large node sizes improve range scan throughput, but slow down inserts.

- 2.4x faster for range queries
- 16x slower for inserts

Best case for point inserts (used in YCSB)
Larger nodes improve range query performance

Increasing the size of nodes decreases the number of nodes accessed during long range queries and thus the number of random memory accesses.
But larger nodes require more shifting on every insert

However, simply increasing the node size does not solve the problem because larger nodes require more work to maintain during inserts.

Traditionally, B-trees (and B⁺-trees) use a sorted array to maintain elements in the nodes.
B-tree insert/range query trade-off
The BP-tree can improve both insert and range queries

The BP-tree can improve long range operations without sacrificing point operations

1.05x faster insert
1.5x faster range query (used in YCSB)
Buffered Partitioned Array (BPA) design

The BP-tree overcomes the point-range tradeoff by using large nodes with an insert-optimized data structure in the leaves called the Buffered Partitioned Array (BPA).

One way to think about the BPA is like collapsing the last two levels of a B-tree into one insert-optimized array-like data structure.

Buffers insertions

Bypasses full searches of the BPA

Stores sub-ranges unsorted to reduce shifting
Example insertions in a BPA

Sort log and count how many elements are destined for each block

<table>
<thead>
<tr>
<th>Log</th>
<th>Header</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 8</td>
<td>7 15 19 89 13 8</td>
<td>17</td>
</tr>
<tr>
<td>25 8 22</td>
<td>7 15 19 89 13 8</td>
<td>17</td>
</tr>
<tr>
<td>25 8 22 27</td>
<td>7 15 19 89 13 8</td>
<td>17</td>
</tr>
</tbody>
</table>

Sort and redistribute all elements evenly because a block overflowed

2+0=2
1+0=1
2+3=5
2+0=2

7 17 25 50 8 13 19 22 27 32 89 93 95
Example range query: `iterate_range(start = 7, length = 2, f)`
B-tree vs BP-tree point operations

B-tree Insert

BP-tree Insert

BP-tree Find

B+-tree Insert

B+-tree Find

Throughput (million ops/s)

Leaf size (bytes)
B-tree vs BP-tree range queries

Throughput (billion elts/s)

Leaf size (bytes)

B+-tree long scan
BP-tree long scan
B+-tree short scan
BP-tree short scan
To what extent do big nodes help range queries?

Traditionally node sizes are small (up to 256 bytes) [CGM01, HP03, B18]

Range queries continue to improve with very large nodes
Costs of memory access in the Disk-Access Model

DAM [AV88] is a classical model that measures disk page accesses (or cache-line accesses, in RAM)

Each memory block fetch has unit cost.
Random vs sequential access cost in the affine model

The affine model [ABZ96,BCF+19] accounts for sequential block accesses being faster than random.

Originally designed for disks and accounted for disk seek vs read.

Small memory

Sequential accesses are cheaper than random accesses

Big memory

Total cost = 3 + 2\alpha
Empirically verifying the affine model in memory

We find the affine model also holds true for RAM

Continues to hold even when the block size goes past 1 page (4Kb)
System Setup

- 48-core 2-way hyperthreaded Intel® Xeon® Platinum 8275CL CPU @ 3.00GHz
- Cache
  - 1.5MiB of L1 cache,
  - 48 MiB of L2 cache,
  - 71.5 MiB of L3 cache across all of the cores
- 189 GB of memory
- All experiments on a single socket with 24 physical cores and 48 hyperthreads
- All times are the median of 5 trials after one warm-up trial
Evaluation on YCSB benchmarks

Relative performance of reference B-tree, Masstree [MKM2012], OpenBwTree [WPL+2018] compared to the BP-tree on uniform random workloads generated from YCSB [CST+10].

We added workloads to test long scans (both iteration and map).

![Normalized Performance Chart]

- Original (OLTP)
- Added (OLAP)
Figure 9: Relative performance compared to the BP-tree on zipfian workloads generated from YCSB.
Conclusion

BP-tree overcomes the decades-old point range tradeoff in B-Trees: it can increase the performance for workloads that include both point operations and long scans.

I/O models (External memory and Affine) apply to in-memory data structures.

Relaxing ordering constraint in B-tree nodes can help achieve overcome tradeoffs.

Code available at: https://github.com/wheatman/concurrent-btrees
Table 1: Throughput (thr., in operations per second) and normalized performance of point operations in the B-tree and BP-tree. Point operation throughput is reported in operations/s. We use N.P. to denote the normalized performance in the B-tree (BP-tree) compared to the best B-tree (BP-tree) configuration for that operation (1.0 is the best possible value).

<table>
<thead>
<tr>
<th>Node size (bytes)</th>
<th>Insert</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Insert</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thr.</td>
<td>N.P.</td>
<td>Thr.</td>
<td>N.P.</td>
<td>Header size (elts)</td>
<td>Block size (elts)</td>
<td>Total size (bytes)</td>
<td>Thr.</td>
<td>N.P.</td>
<td>Thr.</td>
<td>N.P.</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>8.72E6</td>
<td>0.47</td>
<td>2.66E7</td>
<td>0.92</td>
<td>4</td>
<td>4</td>
<td>384</td>
<td>1.05E7</td>
<td>0.54</td>
<td>2.96E7</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>1.56E7</td>
<td>0.84</td>
<td>2.81E7</td>
<td>0.97</td>
<td>4</td>
<td>8</td>
<td>640</td>
<td>1.42E7</td>
<td>0.73</td>
<td>2.96E7</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>1.86E7</td>
<td>1.00</td>
<td>2.86E7</td>
<td>0.98</td>
<td>8</td>
<td>8</td>
<td>1280</td>
<td>1.63E7</td>
<td>0.84</td>
<td>3.05E7</td>
<td>0.96</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>1.74E7</td>
<td>0.93</td>
<td>2.84E7</td>
<td>0.98</td>
<td>8</td>
<td>16</td>
<td>2304</td>
<td>1.83E7</td>
<td>0.94</td>
<td>3.09E7</td>
<td>0.98</td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td>1.34E7</td>
<td>0.72</td>
<td>2.91E7</td>
<td>1.00</td>
<td>16</td>
<td>16</td>
<td>4608</td>
<td>1.87E7</td>
<td>0.97</td>
<td>3.16E7</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td>8.04E6</td>
<td>0.43</td>
<td>2.60E7</td>
<td>0.89</td>
<td>16</td>
<td>32</td>
<td>8704</td>
<td>1.87E7</td>
<td>0.97</td>
<td>3.12E7</td>
<td>0.99</td>
<td></td>
</tr>
<tr>
<td>16384</td>
<td>4.27E6</td>
<td>0.23</td>
<td>1.59E7</td>
<td>0.55</td>
<td>32</td>
<td>32</td>
<td>17408</td>
<td>1.94E7</td>
<td>1.00</td>
<td>3.02E7</td>
<td>0.96</td>
<td></td>
</tr>
<tr>
<td>32768</td>
<td>2.20E6</td>
<td>0.12</td>
<td>1.50E7</td>
<td>0.52</td>
<td>32</td>
<td>64</td>
<td>33792</td>
<td>1.84E7</td>
<td>0.95</td>
<td>2.97E7</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>65536</td>
<td>1.12E6</td>
<td>0.06</td>
<td>1.40E7</td>
<td>0.48</td>
<td>64</td>
<td>64</td>
<td>67584</td>
<td>1.73E7</td>
<td>0.89</td>
<td>1.73E7</td>
<td>0.55</td>
<td></td>
</tr>
</tbody>
</table>
Table 2: Throughput (thr., in expected elements per second) of range queries of varying maximum lengths (max_len) in the B-tree and BP-tree. We also report the normalized performance (N.P.) compared to the best-case performance for each operation (up to 1.0).

<table>
<thead>
<tr>
<th>Node size (bytes)</th>
<th>B-tree</th>
<th></th>
<th></th>
<th>BP-tree</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Short (max_len = 100)</td>
<td>Long (max_len = 100,000)</td>
<td></td>
<td>Short (max_len = 100)</td>
<td>Long (max_len = 100,000)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Map</td>
<td>Iterate</td>
<td>Map</td>
<td>Iterate</td>
<td>Map</td>
<td>Iterate</td>
</tr>
<tr>
<td></td>
<td>Thr.</td>
<td>N.P.</td>
<td>Thr.</td>
<td>N.P.</td>
<td>Thr.</td>
<td>N.P.</td>
</tr>
<tr>
<td>-------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>256</td>
<td>8.56E8</td>
<td>0.77</td>
<td>9.48E8</td>
<td>0.72</td>
<td>1.88E9</td>
<td>0.25</td>
</tr>
<tr>
<td>512</td>
<td>9.58E8</td>
<td>0.86</td>
<td>1.05E9</td>
<td>0.80</td>
<td>2.12E9</td>
<td>0.28</td>
</tr>
<tr>
<td>1024</td>
<td>1.01E9</td>
<td>0.91</td>
<td>1.13E9</td>
<td>0.85</td>
<td>2.69E9</td>
<td>0.36</td>
</tr>
<tr>
<td>2048</td>
<td>1.08E9</td>
<td>0.97</td>
<td>1.20E9</td>
<td>0.91</td>
<td>4.23E9</td>
<td>0.56</td>
</tr>
<tr>
<td>4096</td>
<td>1.11E9</td>
<td>1.00</td>
<td>1.26E9</td>
<td>0.95</td>
<td>5.18E9</td>
<td>0.69</td>
</tr>
<tr>
<td>8192</td>
<td>1.10E9</td>
<td>0.99</td>
<td>1.28E9</td>
<td>0.97</td>
<td>5.97E9</td>
<td>0.80</td>
</tr>
<tr>
<td>16384</td>
<td>1.08E9</td>
<td>0.98</td>
<td>1.29E9</td>
<td>0.98</td>
<td>6.60E9</td>
<td>0.88</td>
</tr>
<tr>
<td>32768</td>
<td>1.08E9</td>
<td>0.97</td>
<td>1.30E9</td>
<td>0.98</td>
<td>7.18E9</td>
<td>0.96</td>
</tr>
<tr>
<td>65536</td>
<td>1.09E9</td>
<td>0.98</td>
<td>1.32E9</td>
<td>1.00</td>
<td>7.50E9</td>
<td>1.00</td>
</tr>
</tbody>
</table>

| | | | | | | |
|-------------------|---|---|---|---|---|---|---|---|
|                   | Header size (elts) | Block size (elts) | Total size (bytes) | Thr. | N.P. | Thr. | N.P. | Thr. | N.P. | Thr. | N.P. |
|-------------------|---|---|---|---|---|---|---|---|---|---|---|---|
| 256               | 4 | 4 | 384 | 4.76E8 | 0.53 | 7.15E8 | 0.59 | 7.32E8 | 0.14 | 1.20E9 | 0.22 |
| 512               | 4 | 8 | 640 | 6.86E8 | 0.76 | 8.93E8 | 0.73 | 1.32E9 | 0.25 | 1.71E9 | 0.32 |
| 1024              | 8 | 8 | 1280 | 7.91E8 | 0.88 | 9.45E8 | 0.78 | 1.72E9 | 0.32 | 1.85E9 | 0.35 |
| 2048              | 8 | 16 | 2304 | 8.98E8 | 1.00 | 1.07E9 | 0.88 | 2.46E9 | 0.46 | 2.54E9 | 0.47 |
| 4096              | 16 | 16 | 4608 | 8.99E8 | 1.00 | 1.13E9 | 0.93 | 3.17E9 | 0.59 | 3.22E9 | 0.60 |
| 8192              | 16 | 32 | 8704 | 8.86E8 | 0.99 | 1.22E9 | 1.00 | 4.19E9 | 0.78 | 4.25E9 | 0.79 |
| 16384             | 32 | 32 | 17408 | 8.14E8 | 0.91 | 1.17E9 | 0.96 | 4.75E9 | 0.89 | 4.75E9 | 0.89 |
| 32768             | 32 | 64 | 33792 | 6.73E8 | 0.75 | 1.05E9 | 0.87 | 5.21E9 | 0.97 | 5.16E9 | 0.96 |
| 65536             | 64 | 64 | 67584 | 5.74E8 | 0.64 | 9.83E8 | 0.81 | 5.35E9 | 1.00 | 5.35E9 | 1.00 |
Table 3: Throughput (in operations/s) of the BP-tree (BPT), B-tree ($B^+T$), Masstree (MT), and OpenBw-tree (BWT) on uniform random and zipfian workloads from YCSB.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Description</th>
<th>Uniform</th>
<th>Zipfian</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$BPT$</td>
<td>$B^+T$</td>
</tr>
<tr>
<td>A</td>
<td>50% finds, 50% inserts</td>
<td>2.91E7</td>
<td>2.33E7</td>
</tr>
<tr>
<td>B</td>
<td>95% finds, 5% inserts</td>
<td>4.70E7</td>
<td>4.46E7</td>
</tr>
<tr>
<td>C</td>
<td>100% finds</td>
<td>4.99E7</td>
<td>4.81E7</td>
</tr>
<tr>
<td>E</td>
<td>95% short range iterations (max_1en = 100), 5% inserts</td>
<td>2.58E7</td>
<td>2.71E7</td>
</tr>
<tr>
<td>X</td>
<td>100% long range iterations (max_1en = 10,000)</td>
<td>8.89E5</td>
<td>6.90E5</td>
</tr>
<tr>
<td>Y</td>
<td>100% long range maps (max_1en = 10,000)</td>
<td>9.18E5</td>
<td>6.45E5</td>
</tr>
</tbody>
</table>
Why does it work

Figure 4 from paper
Continuous reads
Even bigger than a page

Figure 4: Relative slowdown (time), L3 cache misses, and bytes read during the scan test.