Lecture 21: BP-tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees

Helen Xu
hxu615@gatech.edu
Recall: B-trees are classical indexing structures

B/B+-trees are used everywhere

- In-memory indexing
- Databases
- Filesystems

B-trees are \textit{asymptotically optimal} for point operations, e.g., insert, find
Recap: B-tree structure

Often similar to the cache-line size
Recap: B+-tree structure

Leaf nodes are chained together with pointers

Often used in practice

Pivots

... ≈ B children ...

Often similar to the cache-line size

All elements appear in leaves

Leaf nodes

B

... ≈ N / B leaves ...

O (log_B N)
OLAP vs OLTP Workloads

- **Online analytical processing (OLAP)** and **online transaction processing (OLTP)** are two different use cases for data-processing systems.

- OLAP is optimized for **complex data analysis and reporting**, while OLTP is optimized for transactional processing and **real-time updates**.

- Traditionally, systems are **optimized for one or the other**, but recently there has been exploration into **combining both functionalities** into one system.

Problem: B-tree insert-range tradeoff

- B-trees exhibit a **tradeoff** between point inserts (OLTP) and long range queries (OLAP) **as a function of node size**.

- Long range queries are critical for real-time analytics [PTPH12] and graph processing [DBGS22, PGK21, PWXB21].

**Graph:**

- **Range Query**
- **Find**
- **Insert**

**Legend:**

- **Node size (Bytes):** $2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}$
- **Normalized Performance**

Large nodes speed up **range scans at the cost of point inserts**
YCSB Point Operations

Point operations:
- $\text{Insert}(k, v)$: insert a key-value pair $(k, v)$
- $\text{Find}(k)$: return a pointer to the element with the smallest key that is at least $k$

Example: $\text{Find}(15)$

![Diagram showing tree structure with keys 2, 3, 5, 7, 14, 16, 19, 20, 22, 23, 24, 27, 29. The process of finding the correct pivot and searching in the node for the correct element is illustrated.]
Recap example: Insert 8 into B+-tree

Root

Target leaf is full: need to do a data page split

Target leaf splits
Recap example: Insert 8 into B+-tree

Need to adjust pivots

Promote 5 as a pivot
Ordered Range Operations

The importance of ordered iteration in range operations (scans) depends on the use case.

For example, the YCSB requires **range iteration** (in sorted order) to simulate an application of threaded conversations:

```
Iterate_range(start, length, f): applies the function f to length elements in order (by key) starting with the elements with the smallest key that is at least start
```

Example: Load the first 50 messages on some date
Unordered Range Operations

On the other hand, some applications may not necessarily need access to the keys in order.

For example: graph processing, feature storage in machine learning, file system metadata management.

Therefore, we consider another primitive not in YCSB: Map_range(start, end, f): applies the function f to all elements with keys in the range [start, end)

Example: Iterate through David’s neighbors in any order.
Recall: Range operations in B+-trees

Example: Get me 5 elements in sorted order with min key 15.

Step 1: Do a find for the element with the smallest key at least 15

Step 2: Iterate forward 5 steps or until the end, whichever comes first

We can use a similar method for the other range API of [start, end) by just modifying the end condition.
B-tree insert/range query trade-off

There is **no one best node size for all operations** - large node sizes improve range query throughput, but slow down inserts.
B-tree insert/range query trade-off

There is **no one best node size for all operations** - large node sizes improve range query throughput, but slow down inserts.

**Question:** How can we achieve good performance on all of these operations?
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 tradeoff

How can we improve performance overall despite the insert/range tradeoff?
The BP-tree can improve long ranges without sacrificing point operations.
BP-tree design
Motivation: Leaf nodes are the hotspots in B-tree variants

- Every insert will modify at least one leaf.
- Only one in every $O(B)$ inserts will affect the internal nodes.

All elements appear in leaves

$O(N/B)$ elements are internal

$O(\log_B N)$
Buffered Partitioned Array (BPA) Design

- The BP-tree overcomes the insert-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.
Example: Insertions in a BPA
Example: Insertions in a BPA
Example: Insertions in a BPA

Insert(22)

Insert(27)

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

25 8 22 7 15 19 89 13 8 17 32 50 93 95

2 + 0 = 2
1 + 0 = 0
2 + 3 = 5
2 + 0 = 2
Example: Insertions in a BPA

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

Insert(22)

Sort and redistribute all elements evenly because at least one block overflowed:
Example range query:
iterate_range(start = 7, length = 2, f)
Example range query:
iterate_range(start = 7, length = 2, f)

Sort the log and first relevant block, initialize the pointers:

log_ptr
blocks_ptr
Example range query:

`iterate_range(start = 7, length = 2, f)`

Sort the log and first relevant block, initialize the pointers:

Advance the pointers to perform sorted iteration:
Example range query:
iterate_range(start = 7, length = 2, f)
Bitvector Optimization

• To avoid unnecessary sorting, the BPA keeps a bit vector of length `num_blocks` that denotes whether the elements in each block are currently sorted.

• It sorts a block during a range query if and only if the corresponding bit in the bit vector is unset.

• The bit vector is maintained during inserts / range queries.
BP-tree concurrency control
Recall: Reader-Writer Concurrency

• A reader-writer lock allows **concurrent access for read-only operations**, whereas write operations require exclusive access.

• That is, multiple threads can read the data in parallel, but **an exclusive lock is needed for writing/modifying data**.

• All other threads (both writers and readers) are blocked when the lock is taken in write mode.
Recall: Optimistic concurrency control

Concurrency control is defined at the node level, so we can use the same reader/writer concurrency scheme for inserts as regular B-trees.

Most modifications to a B+-tree will not require a split or merge.

Instead of assuming that there will be a split/merge, optimistically traverse the tree using read latches.

If you guess wrong, repeat traversal with the pessimistic algorithm.

From Utah CS6530
Range Query Concurrency Control

In regular B-trees, range operations are read-only, so we can just take read locks top-down, left-right.
Example: iterate_range(start=25, length=2)
Example: iterate_range(start=25, length=2)
Example: iterate_range(start=25, length=2)
Example: iterate_range(start=25, length=2)
Example: iterate_range(start=25, length=2)
Example: iterate_range(start=25, length=2)

Question: Can we use this scheme for range query concurrency in the BP-tree?
Adapting B-tree Range Query Concurrency for the BP-tree

Problem: Range queries in the BPA **might modify the array** (to sort the log / blocks), so we can’t always take a reader lock on the leaves.

Naive solution: Take read locks on the way down, then always take **writer locks on the leaves**.

The naive solution causes **performance issues** because write locks are exclusive, so this method bottlenecks other threads.
Using the bitvector to avoid taking the write lock

We use the bitvector optimization to **avoid contention on the write lock** when the input distribution is skewed.

For each leaf touched in a range query in the BP-tree:

Take the read lock and check 1) whether the log is sorted, and 2) whether the relevant blocks are sorted (using the bitvector)

If both are true, the iteration can proceed with just the read lock

If at least one is false, upgrade the read lock to a write lock
Evaluation
We evaluate the BP-Tree on several tests using the Yahoo! cloud serving benchmark (YCSB) and compare it to a selection of different structures.

The YCSB has two phases:

- **Load** - add some base number of elements
- **Run** - perform concurrent operations defined by some workload

Only the run phase is timed

For concreteness, each phase has 100M operations. The YCSB also allow definition of input distribution (e.g., uniform random, skewed, etc.)
BP-tree system/experiment 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

Performance of B-tree, Masstree [MKM2012], OpenBwTree [WPL+2018] and BP-tree on YCSB [CST+10] with 100M ops in both the load and run phase.

BP-tree matches the performance of point operations and improves range queries by 1.5x
B-tree vs BP-tree point operations

Throughput (million ops/s)

Leaf size (bytes)

BP-tree Find
BP-tree Insert
B⁺-tree Find
B⁺-tree Insert
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
Performance Modeling of Large Nodes
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

![Diagram showing normalized performance vs. node size with lines for range query, find, and insert, with 64KB node size highlighted.]
Recall: Cost of access in Disk-Access Model (DAM)

The DAM [Aggarwal and Vitter, ’88] is a classical model that measures disk page access (or cache-line accesses, in RAM).

Each memory block fetch has unit cost.
Recall: Random vs Sequential Access Cost in the Affine Model

The affine model \([ABZ96, BCF+19]\) accounts for sequential block accesses being faster than random (due to prefetching, etc.).

Random access has unit cost, and \textbf{sequential access has cost} \(\alpha < 1\).

Originally designed for disks and accounted for disk seek vs read.
Finding the empirical parameters with the scan test

We perform the following scan test to empirically derive $\alpha$:

Allocate a contiguous array of $X$ bytes ($X$ is large, in the GB range)

for block size $\ell$ from 1 to $X$ in powers of 2:

in parallel and in random order, scan over the entire array in separate blocks of size $B$

Measure time as a function of varying block size

![Diagram showing time measurement for different block sizes $\ell$]
Finding the empirical parameters with the scan test

We also need to find $r$, the cost of reading a random location in DRAM, and $\omega$, the cost of writing to a random location in DRAM.

By setting $\ell = Z$ (the cache-line size), we can compute the latency of reading a random cache line in DRAM by dividing the total time by the number of lines read.

\[
\ell = Z
\]
Results of scan test

We found the following:

- $\alpha = 0.3$
- $r = 1.95$ ns
- The machine has a cache line size $Z = 64$ bytes, and we use the heuristic of $w = 2r$. 

Expression for cost in terms of $Z, \alpha, r, w, N$

Empirical findings for parameter values

Predicted time for operations
Empirically validating the affine model in memory

- We find the affine model also holds true for RAM using the scan test.
- Interestingly, it continues to hold even when the block goes past 1 page (4Kb) - more on that later.
Why does scan performance continue to improve after page sizes?

Although the cache-line prefetcher does not cross page boundaries [Intel manual], we continue to see performance improvements after 4Kb block reads.

To try to understand why, we used the Intel Performance Counter Monitor (PCM) to measure the extra bytes read by the memory controller and the L3 cache misses:

L3 misses can account for speedup in blocks up to page size

Extra bytes read by DRAM memory controller may explain larger blocks
Summary

• B-trees (and any other blocked data structure, e.g., B-skip lists) exhibit a tradeoff between point and range operations depending on the node size.

• The affine model provides a way to analytically determine the benefits of larger node sizes during scans.

• 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.
BACKUP PAST HERE
### Optimal Search-Insert Tradeoff

[Brodal, Fagerberg 03]

<table>
<thead>
<tr>
<th>Insert</th>
<th>Point Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O \left( \frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}} \right)$</td>
<td>$O \left( \log_{1+B^\varepsilon} N \right)$</td>
</tr>
</tbody>
</table>

#### Optimal tradeoff

- **(function of $\varepsilon=0...1$)**

#### B-tree

- **($\varepsilon=1$)**
  - $O \left( \log_B N \right)$
  - $O \left( \log_B N \right)$

#### $\varepsilon$ values

- **$\varepsilon=1/2$**
  - $O \left( \frac{\log_B N}{\sqrt{B}} \right)$
  - $O \left( \log_B N \right)$

- **$\varepsilon=0$**
  - $O \left( \frac{\log N}{B} \right)$
  - $O \left( \log N \right)$

---

BP-tree 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
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>B-tree</th>
<th></th>
<th></th>
<th></th>
<th>BP-tree</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
<td>Find</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thr.</td>
<td>N.P.</td>
<td>Thr.</td>
<td>N.P.</td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Header size (elts)</td>
<td>Block size (elts)</td>
<td>Total size (bytes)</td>
<td>Insert</td>
<td>Thr.</td>
<td>N.P.</td>
<td>Find</td>
<td>Thr.</td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
</tbody>
</table>
### BP-tree raw range data

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>BP-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short (max_len = 100)</td>
<td>Long (max_len = 100,000)</td>
<td>Short (max_len = 100)</td>
</tr>
<tr>
<td></td>
<td>B-tree</td>
<td>BP-tree</td>
</tr>
<tr>
<td></td>
<td>Map</td>
<td>Iterate</td>
</tr>
<tr>
<td></td>
<td>Thr.</td>
<td>N.P.</td>
</tr>
<tr>
<td>256</td>
<td>8.56E8</td>
<td>0.77</td>
</tr>
<tr>
<td>512</td>
<td>9.58E8</td>
<td>0.86</td>
</tr>
<tr>
<td>1024</td>
<td>1.01E9</td>
<td>0.91</td>
</tr>
<tr>
<td>2048</td>
<td>1.08E9</td>
<td>0.97</td>
</tr>
<tr>
<td>4096</td>
<td>1.11E9</td>
<td>1.00</td>
</tr>
<tr>
<td>8192</td>
<td>1.10E9</td>
<td>0.99</td>
</tr>
<tr>
<td>16384</td>
<td>1.08E9</td>
<td>0.98</td>
</tr>
<tr>
<td>32768</td>
<td>1.08E9</td>
<td>0.97</td>
</tr>
<tr>
<td>65536</td>
<td>1.09E9</td>
<td>0.98</td>
</tr>
</tbody>
</table>
# BP-tree YCSB raw data

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_len = 100), 5% inserts</td>
<td>2.58E7</td>
<td>2.71E7</td>
</tr>
<tr>
<td>X</td>
<td>100% long range iterations (max_len = 10,000)</td>
<td>8.89E5</td>
<td>6.90E5</td>
</tr>
<tr>
<td>Y</td>
<td>100% long range maps (max_len = 10,000)</td>
<td>9.18E5</td>
<td>6.45E5</td>
</tr>
</tbody>
</table>
BP-tree on Zipfian

Figure 9: Relative performance compared to the BP-tree on zipfian workloads generated from YCSB.