ISSN: 3139-0862

B-Trees and B Plus Trees in Data Management: A Comparative Analysis of Search and Range-Query Performance

Aditya Goyal[#*1], Nidhi Chauhan[#2], Sairaj Bhoir[#3], Reshma Sonar[#]
[#]*B.Tech Computer Science and Engineering Students, Department of Computer Science and Engineering, MIT World Peace University, Survey No. 124, Paud Road, Kothrud, Pune 411038, Maharashtra, India*
[1]adityagoyal3007@gmail.com, [2]nidhiee002@gmail.com, [3]sairaj.bhoir2713@gmail.com

Abstract—

Database indexes help a system find records without scanning an entire table. B-Trees and B Plus Trees both organize ordered data, but they store entries differently: a B-Tree may keep entries in internal nodes as well as leaves, while a B Plus Tree keeps them at the leaf level and links the leaves in key order. This paper compares the two structures for point searches and range queries. We estimate node capacity from a simplified 4 KB page layout, then evaluate Python implementations across 15 trials. Node visits are counted as a proxy for page accesses, and query time is measured separately. For a range query returning 10000 keys from a tree of 500000 keys, the B Plus Tree recorded 59.27 node accesses on average, compared with 88.21 for the B-Tree. With equal node capacities the counts were nearly the same, which suggests the reduction came mainly from higher fan-out. The B Plus Tree also completed large range scans faster, although the timing difference may partly reflect the traversal code. Point-search times showed no consistent difference. Because the tests ran in memory, the results do not measure physical disk input and output.

Keywords—B-Tree; B Plus Tree; database indexing; point search; range query; fan-out; node accesses

I. INTRODUCTION

A database table can hold millions of rows. If every query had to read all of them, even simple lookups would become slow as the data grew. An index avoids this by keeping a second, ordered structure that points to the rows, so the system can jump to the part of the table it needs. The index costs extra storage and has to be updated whenever the data changes, but for most workloads the trade is worth it.

B-Trees were introduced by Bayer and McCreight as a way to keep large ordered indexes balanced while records are inserted and deleted [1]. Unlike a binary search tree, each node holds many keys and many child pointers. That keeps the tree shallow, which matters when every node is a page that has to be read from storage. The B+ Tree is a close relative: internal nodes guide the search, while data entries sit in the leaves, and the leaves are chained together in key order [2], [3].

The names are used loosely, so we fix our meaning. Comer already noted that the term "B-tree" is applied to several variations and that the leaf-only variation is the very popular one [2]. In this paper a B-Tree may store entries in any node, and a B+ Tree stores them only in the leaves.

Workloads differ. A point search asks for one key, for example a single roll number. A range query asks for everything between two keys, for example all roll numbers from 1000 to 2000. The two structures do not necessarily behave the same on both. We ask a narrow question: how do the structural differences show up in point searches and range queries, and how much of the difference comes from the structure itself rather than from how many entries fit in a node?

The objectives of this paper are as follows. (1) Explain the structural differences between B-Trees and B+ Trees. (2) Derive node capacity, fan-out and tree height from an explicit page layout so the calculation can be reproduced. (3) Measure node accesses and query time for point searches and range queries in our own implementation, over repeated independent trials. (4) Separate the effect of linked leaves from the effect of higher fan-out, and state what the measurements do and do not show.

II. RELATED WORK

Bayer and McCreight's original paper treats the B-Tree as a dynamic index that must stay balanced and keep access costs low on external storage as records come and go [1]. Comer's survey describes the B+-tree as the very popular variation in which all keys reside in the leaves and the leaves are linked left to right, so that sequential processing is easy, and calls the B-tree the de facto standard organization for indexes in a database system [2]. Database textbooks present the B+ Tree in their chapters on indexing [3]. Graefe's survey reviews B-tree fundamentals together with the transactional techniques and optimizations that real systems add [4].

On the theory side, Yi studies one-dimensional range indexes in external memory. He proves, in a "dynamic indexability" model that counts only the block transfers needed for the result, that the query–update tradeoff of known dynamic B-trees is optimal when log_B(N/M) is constant [5]. We use this paper only for the general point that block transfers, not CPU steps alone, are the natural cost of an index. Database documentation gives a practical view: PostgreSQL states that its B-tree indexes handle equality and range queries on data that can be sorted [6]. Documentation describes one system and is not a controlled comparison.

Recent work shows that node organization is still an active research topic. Kwon et al. propose the DB+-tree, a variant of the B+-tree for main-memory database systems [7]. We cite it only as an example of this and did not examine its evaluation.

The sources describe or optimize one structure at a time. This paper adds a small side-by-side comparison designed to distinguish the effect of fan-out from the effect of linked leaves.

ISSN: 3139-0862

III. Structure And Fan-Out

A. B-Tree

A B-Tree node holds a sorted list of keys and one more child pointer than it has keys. To search, we find the position of the target key in the node and either stop, if the key is there, or follow the child pointer for that range. Because entries can be stored in internal nodes, a search can end before it reaches a leaf. Nodes split when they overflow, which keeps all leaves at the same depth.

B. B+ Tree

In a B+ Tree, internal nodes hold separator keys and child pointers, while data entries sit in the leaves, and the leaves are linked in key order. A range query does one root-to-leaf descent to find the lower bound and then follows the links until it passes the upper bound, without climbing back up the tree.

C. Node layout and capacity

Fan-out, the number of children a node can have, is set by how many entries fit in one node. We use a simplified fixed-size model of a 4096-byte page: a 16-byte header (entry count, node type and flags), 8-byte keys, 8-byte child pointers, and 8-byte record pointers (a row identifier). Real systems use variable-length keys and slot directories, so exact capacities differ, but the proportions are the useful part. Table I shows the layout.

TABLE I. Node layout and capacity for a 4096-byte page

Node type	Bytes used by m entries	Max entries m	Fan-out
B-Tree node	16 + m(8+8) + (m+1)8 = 4080	169 keys	170
B+ internal node	16 + m·8 + (m+1)8 = 4088	254 keys	255
B+ leaf	16 + m(8+8) + 8 (next leaf) = 4088	254 entries	–

Each B-Tree key carries a record pointer and the node has m+1 child pointers. A B+ internal key has no record pointer. One more B-Tree key would need 4104 bytes, which does not fit.

Solving each inequality gives the capacities: 4096−16−8 = 4072 bytes divided by 24 bytes per B-Tree key gives 169.67, so 169 keys; divided by 16 bytes per B+ internal key gives 254.50, so 254 keys; and divided by 16 bytes per leaf entry gives 254.50, so 254 entries. The B+ internal node has about 1.5 times the fan-out of the B-Tree node (255 against 170). These same formulas are used in the code.

If every node were completely full, a B-Tree of height h could hold $170^h - 1$ entries, and a B+ Tree could hold $254 \cdot 255^{h-1}$. Table II applies this to four sizes. At ten million entries the B+ Tree needs three levels and the B-Tree four, and at a billion the figures are four and five. Real trees are partly empty after splits. In our experiment the B-Tree nodes were on average 69% full and the B+ leaves 70% full, so real heights can be larger than this best case.

TABLE II. Minimum possible height with completely full nodes

Number of entries	B-Tree	B+ Tree
1,000,000	3	3
10,000,000	4	3
100,000,000	4	4
1,000,000,000	5	4

Computed from the capacities in Table I.

IV. Materials and Methods

We wrote both trees in Python 3. The B-Tree follows the standard textbook algorithm with minimum degree t = 85, so a node holds at most 169 keys. The B+ Tree splits leaves and internal nodes on overflow and keeps a next-leaf pointer in each leaf. We tested both against a sorted-list oracle (random point and range queries, including missing keys, empty ranges and full ranges) and checked structural invariants: sorted keys, minimum occupancy, all leaves at one depth, and a complete sorted leaf chain. The tests also used deliberately tiny nodes to force many splits. The benchmark repeats a full-range scan check in every trial.

We compare three trees built on identical data: the B-Tree (169 keys per node), a B+ Tree using the page layout of Table I (254 entries per leaf, 255 children per internal node), and an equal-capacity control B+ Tree with the same capacity as the B-Tree (169 entries per leaf, 170 children per internal node). The control separates the effect of the leaf links from the effect of higher fan-out.

Two kinds of measurement. Node accesses count every node a search or scan visits, one per node. They are a model of page reads. The program runs entirely in memory and performs no disk I/O, so node accesses do not include seek time, caching, prefetching or operating-system effects. Node accesses are deterministic for a given tree and query set. Query time is wall-clock time from Python's high-resolution timer, which depends on the interpreter and the machine, and we treat it more cautiously.

Procedure. We ran 15 independent trials. In each trial and for each dataset size (10,000, 100,000 and 500,000 keys) a seeded generator drew distinct random integers and inserted them in random order, and the same keys and queries were used for all three trees. Each trial ran 2,000 point searches for existing keys and 200 range queries for each of 10, 100, 1,000 and 10,000 consecutive keys (ranges larger than one fifth of the dataset were skipped). Each timing is a warm-up run followed by five passes (three for range queries) with garbage collection disabled, and the trial's value is the median pass. We report node accesses as mean ± standard deviation across trials, and time as the median across trials with the interquartile range (IQR). We also report the paired per-trial time ratio and in how many trials the B+ Tree was faster. The runs used Python 3.13.16 on a shared cloud virtual machine with an Intel Xeon 2.10 GHz processor, so absolute times are not portable.

Reproducibility. The code, tests, raw per-trial data, analysis scripts, and the full run log with file checksums are available from the authors.

V. Results

Table III gives node accesses at 500,000 keys, and Tables IV and V give times. Each value is a summary of 15 trials.

ISSN: 3139-0862

TABLE III. NODE ACCESSES PER QUERY, 500,000 KEYS (MEAN ± SD OVER 15 TRIALS)

Measure	B-Tree	B+ Tree (page layout)	B+ Tree (equal capacity)
Height (levels)	3	3	3
Nodes in index	4,258	2,830	4,271
Point search	2.99 ± 0.00	3.00 ± 0.00	3.00 ± 0.00
Range, 10 keys	3.09 ± 0.03	3.05 ± 0.02	3.09 ± 0.02
Range, 100 keys	3.86 ± 0.03	3.56 ± 0.03	3.86 ± 0.02
Range, 1,000 keys	11.50 ± 0.10	8.61 ± 0.06	11.47 ± 0.11
Range, 10,000 keys	88.21 ± 0.88	59.27 ± 0.64	87.83 ± 0.79

Height and node counts are means over trials; heights were the same in every trial.

A. Point search

Point searches cost the same in all three trees: 2.99 node accesses in the B-Tree and 3.00 in the B+ Tree, with three levels each. The B-Tree's ability to stop at an internal node barely helps. Only 0.8% of its keys are stored above the leaf level, and 0.8% of our point searches ended early. Time did not separate them either: the median time was 1.4 µs for the B-Tree and 1.3 µs for the B+ Tree, and the B+ Tree was faster in only 10 of 15 trials.

B. Range queries: node accesses

In the page-layout comparison the B+ Tree read fewer nodes on larger ranges. For 1,000 keys it read 8.61 against 11.50, and for 10,000 keys it read 59.3 against 88.2, which is 33% fewer. Its index also had 34% fewer nodes. At 100,000 keys the 10,000-key range gave 57.0 against 90.8, so the pattern is not specific to one size. In the equal-capacity control the B+ Tree needed 87.8 against 88.2, and its index had 4,271 nodes against 4,258. The difference had disappeared. The ratio also matches the layout: 255 against 170 children per node is 1.50, and the measured ratio of node accesses was 1.49. Fig. 1 shows this.

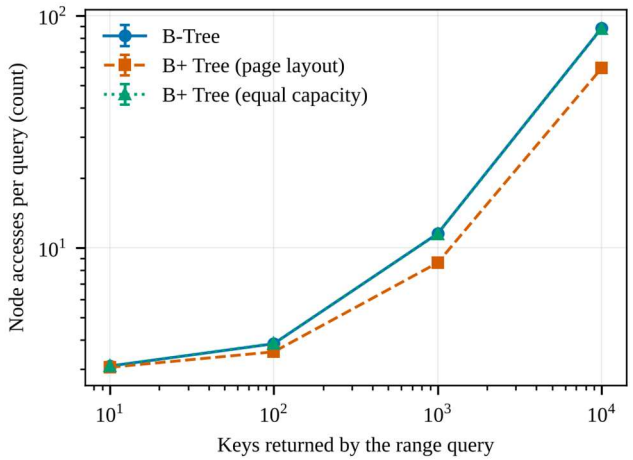


Fig. 1. Node accesses per range query, 500,000 keys. Points are means over 15 trials; bars show the minimum to maximum (mostly hidden by the markers). The B-Tree and equal-capacity curves overlap.

C. Range queries: time

TABLE IV. TIME PER QUERY IN MICROSECONDS, 500,000 KEYS (MEDIAN OVER 15 TRIALS, IQR IN BRACKETS)

Query (µs)	B-Tree	B+ Tree (page layout)	B+ Tree (equal capacity)
Point search	1.4 (1.3–1.4)	1.3 (1.2–1.7)	1.3 (1.3–1.6)
Range, 10 keys	3.2 (2.9–3.3)	3.6 (3.0–4.1)	2.6 (2.5–3.3)
Range, 100 keys	14.4 (14.0–15.3)	10.1 (9.5–10.5)	9.8 (9.5–10.5)
Range, 1,000 keys	123 (120–132)	79.9 (76.0–86.4)	90.9 (74.5–94.6)
Range, 10,000 keys	1,483 (1,409–1,547)	866 (850–935)	912 (865–1,018)

Python wall-clock time on one shared virtual machine; not portable.

TABLE V. PAIRED TIME RATIO, B-TREE TIME DIVIDED BY B+ TREE TIME

Query	B+ Tree (page layout)	B+ Tree (equal capacity)
Point search	1.06; 10/15 (0.76–1.78)	1.07; 9/15 (0.71–1.71)
Range, 10 keys	0.91; 5/15 (0.59–1.22)	1.06; 11/15 (0.76–1.49)
Range, 100 keys	1.46; 15/15 (1.06–1.93)	1.47; 15/15 (1.12–1.97)
Range, 1,000 keys	1.55; 15/15 (1.33–2.07)	1.59; 15/15 (1.21–2.14)
Range, 10,000 keys	1.70; 15/15 (1.30–2.37)	1.59; 15/15 (1.41–2.07)

Median ratio (minimum–maximum over trials); number of trials in which the B+ Tree was faster. A ratio above 1 means the B+ Tree was faster.

For ranges of 100 keys or more the B+ Tree was faster in all 15 trials, in both configurations. For the 10,000-key range the median ratio was 1.70 (page layout) and 1.59 (equal capacity). The equal-capacity result matters: with the same node accesses, the B+ Tree scan was still about 1.6 times faster here. We believe the main reason is that following a leaf link is a simple loop, while our B-Tree range query is a recursive in-order traversal that goes back up and down the tree. Part of the gap is therefore our implementation, and a carefully optimized B-Tree traversal might narrow it. For 10-key ranges and point searches the times were not consistently different (the B+ Tree was faster in 5 and 11 of 15 trials for 10-key ranges). Fig. 2 shows the medians and IQRs.

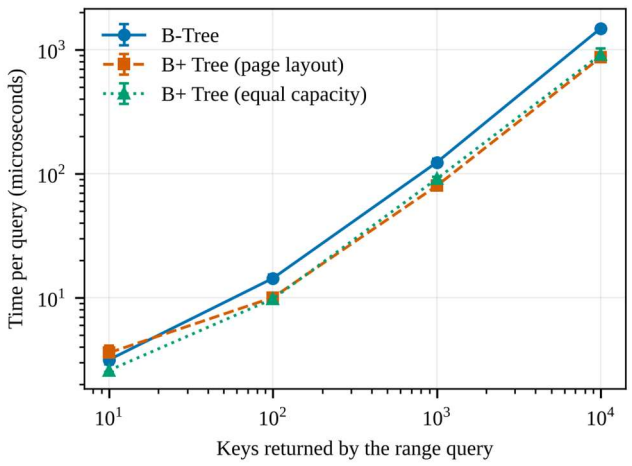


Fig. 2. Time per range query, 500,000 keys. Points are medians over 15 trials; bars show the interquartile range.

ISSN: 3139-0862

VI. Discussion

Three things stand out. First, for point searches the structure barely mattered at these sizes. What matters is the number of levels, and that depends on fan-out. Table II suggests the B+ Tree can save a level at some sizes, but our experiment did not reach those sizes, so we cannot confirm it empirically.

Second, the node-access advantage for range queries came from capacity. Storing entries in internal nodes costs a B-Tree very little in wasted space, since those nodes hold only about 0.8% of the keys. What it costs is room: every key carries a record pointer, so fewer entries fit per page and a scan touches more pages. When we gave the B+ Tree the same capacity, the advantage in node accesses vanished. This comes from one experiment with one key type, so it should be tested further before it is treated as general.

Third, linked leaves still seem to matter, in a different way. They did not reduce node accesses at equal capacity, but they made large scans faster in our runs. On a real disk, reading consecutive leaves can also benefit from prefetching, which our in-memory model does not capture. This is consistent with the common use of the B+ variant in database systems [2].

VII. Limitations

The experiment runs entirely in memory. Node accesses model page reads but are not disk I/O, and we did not measure seek time, caching, prefetching or buffer-pool behavior. Times come from interpreted Python on a shared virtual machine and vary from run to run; we therefore report medians, IQRs and paired ratios over 15 trials, but absolute times are not portable and part of the time gap reflects how we wrote the two traversals. The page layout is a simplified fixed-size model without variable-length keys. We used random integer keys inserted in random order, so sorted or skewed data may behave differently. We did not test deletions, concurrent access or mixed workloads. The largest dataset has 500,000 keys, far below the sizes in Table II. Finally, this is a focused comparison using seven sources, not a systematic literature review.

VIII. Conclusions

B-Trees and B+ Trees both give logarithmic search over ordered data and differ mainly in where entries are stored. In our measurements point searches cost the same in both. For range queries, the B+ Tree with a 4 KB page layout needed about 33% fewer node accesses than the B-Tree, but with equal node capacity the node accesses were practically identical, so most of that saving came from fan-out. Linked leaves still made large scans faster in our implementation, by a factor of about 1.6 at equal capacity, though part of that comes from how the traversals were coded.

These results do not establish that one structure is better in every setting. They show why node capacity and traversal method should be considered separately. Future work should repeat the comparison on a disk-based implementation with a controlled buffer pool, add deletions and mixed workloads, and try sorted, skewed and string keys.

References

[1] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972, doi: 10.1007/BF00288683.

[2] D. Comer, "The ubiquitous B-tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, Jun. 1979.

[3] A. Silberschatz, H. F. Korth and S. Sudarshan, *Database System Concepts*, 6th ed. New York, NY, USA: McGraw-Hill, 2010, ch. 11.

[4] G. Graefe, "Modern B-tree techniques," *Foundations and Trends in Databases*, vol. 3, no. 4, pp. 203–402, 2011.

[5] K. Yi, "Dynamic indexability and the optimality of B-trees," *Journal of the ACM*, vol. 59, no. 4, art. 21, 2012, doi: 10.1145/2339123.2339129.

[6] PostgreSQL Global Development Group, "Index types," *PostgreSQL Documentation*. [Online]. Available: https://www.postgresql.org/docs/current/indexes-types.html (accessed Oct. 18, 2026).

[7] Y. Kwon, S. Lee, Y. Nam, J. C. Na, K. Park, S. K. Cha and B. Moon, "DB+-tree: A new variant of B+-tree for main-memory database systems," *Information Systems*, vol. 119, art. 102287, 2023, doi: 10.1016/j.is.2023.102287.