Project Report: B-Tree Micro-Optimizations in PostgreSQL 17.4

Sricharan Koride
University of Southern California
Los Angeles, CA, USA
koride@usc.edu

Saikarthik Pendela
University of Southern California
Los Angeles, CA, USA
pendela@usc.edu

ABSTRACT

This project explores the internals of PostgreSQL 17.4 by introducing two targeted B-Tree index optimizations: a leaf-page lookahead prefetch mechanism and a linear-scan shortcut for small leaf pages. Working in a Docker-based development environment, we built a debug-enabled PostgreSQL instance, integrated GDB/VS Code debugging tools, and loaded the IMDB/JOB benchmark dataset to evaluate our changes. The prefetching feature overlaps I/O and CPU work by anticipating sibling leaf page access during range scans, while the linear-scan shortcut replaces binary search on very small pages to reduce overhead from branch mispredictions. Both features are exposed through configurable GUC parameters, allowing flexible session-level experimentation. Benchmarking showed that although performance gains were sometimes modest or inconsistent, the implementation successfully demonstrated how database internals can be extended and tuned. More importantly, the project provided hands-on experience with PostgreSQL's storage engine, index access methods, and performance evaluation in a controlled environment.

1 INTRODUCTION

Indexes are fundamental to database performance. PostgreSQL, like most relational database systems, relies on B-Trees as its default access method for point queries, range queries, and ordered scans. Since the Lehman–Yao algorithm [1], B-Trees have been refined to handle concurrency, recovery, and crash-safety, becoming a backbone of modern DBMS internals.

Despite decades of optimization, there is still room to explore lightweight enhancements at the micro-level. Many query workloads, especially analytical benchmarks such as IMDB/JOB [2], repeatedly traverse B-Tree leaf pages during range scans. At this granularity, even small inefficiencies can accumulate, affecting latency and resource usage. Likewise, the traditional binary search algorithm inside leaf pages may be overkill when the page holds only a handful of entries.

In this project, we implemented and evaluated two B-Tree micro-optimizations inside PostgreSQL 17.4 [3]:

(1) **Leaf-page lookahead prefetch.** During a range scan, the system proactively issues a prefetch request for the next sibling leaf page, overlapping I/O with CPU work to reduce stall time.
(2) **Linear search for small leaf pages.** When a leaf page contains fewer than a configurable threshold of items (default: 4), the binary search is replaced with a simple linear scan, which avoids branch misprediction and loop overhead.

Both optimizations are exposed as Grand Unified Configuration (GUC) parameters, disabled by default, and can be toggled at the session level. This allows us to control experimental conditions and isolate their effects on query execution.

The goal of the project is not to achieve breakthrough performance improvements, but to gain hands-on experience modifying a mature database kernel. The project emphasizes understanding PostgreSQL's index access method framework, integrating new GUC parameters, and validating correctness through benchmarks. Performance analysis is still important, but regressions are acceptable as long as the system remains correct and stable.

The remainder of this report is organized as follows: Section 2 describes the implementation of the new GUC parameters and the changes to PostgreSQL's B-Tree source code. Section 3 outlines our experimental setup using the IMDB/JOB benchmark. Section 4 presents results and analysis. Section 5 concludes with key lessons and future directions.

2 IMPLEMENTATION

2.1 GUC Integration

We introduced three GUC parameters:

- `btree_leaf_prefetch` (bool, default off).
- `btree_binsrch_linear` (bool, default off).
- `btree_binsrch_linear_threshold` (int, default 4).

These were declared in `nbtree.h`, defined in `nbtutils.c`, and registered in `guc_tables.c`.

2.2 Leaf-Page Prefetch

To improve sequential index scan performance, we implemented a leaf-page lookahead prefetching mechanism in PostgreSQL's B-tree access method. The modification is confined to the `_bt_readpage` function (`src/backend/access/nbtree/nbtsearch.c`), which is responsible for loading qualifying tuples from a leaf page during an index scan.

Design. We introduced a new configuration parameter (`btree_leaf_prefetch`), exposed as a GUC variable. When enabled, the system issues a non-blocking I/O hint for the sibling leaf page that the scan is expected to visit next. The implementation leverages PostgreSQL's existing `PrefetchBuffer` facility, which requests the kernel to schedule asynchronous reads for a given block without altering normal control flow.

Guard Conditions. Prefetching is triggered only under the following conditions:

- The current page is a *leaf* page (internal pages are ignored).
- The scan direction indicates that more items may be found in the sibling page (`currPos.moreRight` for forward scans, `currPos.moreLeft` for backward scans).
- The page is not the rightmost or leftmost leaf, respectively, ensuring that a valid sibling exists.

- The feature toggle `btree_leaf_prefetch` is set to on.

When all conditions are met, a call to `PrefetchBuffer` is issued on either `opaque->btpo_next` (forward scans) or `opaque->btpo_prev` (backward scans).

Correctness. Importantly, prefetching does not affect correctness. The scan continues to operate identically even if the prefetched page is not in cache by the time it is needed, or if the index structure has changed due to concurrent updates. Prefetch is treated purely as a hint to the buffer manager and can be safely ignored by the kernel or overridden by later I/O.

2.3 Linear Search for Small Pages

Linear Scan on Small Leaf Pages. To reduce overhead on very small leaf pages, we implemented a sequential linear scan shortcut in PostgreSQL's B-tree search routine. The modification is confined to the `_bt_binsrch` function, which is normally responsible for locating the position of a key within a page using binary search.

Design. We introduced a new configuration parameter (`btree_binsrch_linear`), exposed as a GUC variable, along with a tunable threshold parameter (`btree_binsrch_linear_threshold`, default = 4). When enabled, if the number of items on a leaf page does not exceed the threshold, the algorithm performs a simple linear walk through the items instead of invoking the standard binary search. Each comparison uses the same `_bt_compare` routine as binary search, ensuring consistent ordering semantics. This reduces branch misprediction and loop overhead for tiny pages where binary search offers no asymptotic advantage.

Guard Conditions. Linear scan is triggered only under the following conditions:

- The current page is a *leaf* page (internal pages always retain binary search).
- The number of data items in the page is less than or equal to `btree_binsrch_linear_threshold`.
- The feature toggle `btree_binsrch_linear` is set to on.

If these conditions are not satisfied, the standard binary search logic is executed as usual.

Correctness. The linear scan preserves correctness because it reuses the same comparison routine (`_bt_compare`) and applies the same update rules to `low` and `high` pointers as binary search. At the end of the scan, the insertion position is guaranteed to match that of the original algorithm. For larger pages, or when disabled, the fallback to binary search ensures no behavioral changes. The only effect of the linear path is reduced instruction overhead on very small leaf pages, potentially improving performance in sparse or recently-split indexes.

3 EXPERIMENTAL SETUP

3.1 System Specifications

3.2 Software Environment

- **PostgreSQL:** Version 17.4, compiled from source with debug symbols enabled.

Table 1: Hardware and system specifications of the experimental environment.

Component	Specification
Device Model	Lenovo Legion 5 15ACH6H
Processor	AMD Ryzen 7 5800H with Radeon Graphics (3.20 GHz)
Installed RAM	16 GB (3200 MT/s)
Graphics Card	8 GB (Multiple GPUs installed)
Storage	1.86 TB (423 GB used)
Operating System	Windows 11, 64-bit, x64-based processor

- **Docker:** Used to containerize the environment with large shared memory allocation (`-shm-size=32g`) to support PostgreSQL workloads and debugging.
- **Development Tools:**
  - GDB for low-level debugging of backend processes.
  - Visual Studio Code with the C/C++ extension pack for source-level debugging and breakpoint attachment.
  - WSL (Windows Subsystem for Linux) as the build and execution environment.

4 PERFORMANCE ANALYSIS

4.1 Linear Scan Only

Table 2 summarizes the effect of enabling the linear-scan shortcut in place of binary search on small leaf pages. Overall, the results show a slight regression: the median improvement across queries is −1.90%, and the mean is −10.78% with a relatively high standard deviation of 30.54. This indicates that while some queries benefit from the feature, many others experience slowdowns.

Out of 113 queries tested, 41 queries (36.3%) showed performance improvements, while 68 queries (60.2%) degraded, and 4 queries (3.5%) showed no observable change. The best observed improvement was +15.93%, but the worst degradation reached −173.24%, highlighting that the shortcut can backfire significantly when applied to pages larger than the threshold. The interquartile range (Q1 = −10.00%, Q3 = +1.73%) further suggests that most queries clustered around neutral or slightly negative improvements.

These results align with expectations: linear scan is only advantageous for very small leaf pages (e.g., with $\leq$ 4 items). For larger pages, the added checks and sequential comparisons introduce overhead, leading to regressions. Thus, while the optimization is functionally correct and useful in niche cases, it is not broadly beneficial for the JOB workload.

4.2 Prefetch Only

Table 3 summarizes the effect of enabling leaf-page lookahead prefetching. Overall, the results again show a slight regression: the median improvement across queries is −0.68%, and the mean improvement is −11.58% with a standard deviation of 36.34. This suggests that while some queries benefit, many queries suffer slowdowns, and the variation across queries is relatively high.

Out of 113 queries tested, 44 queries (38.9%) improved, 62 queries (54.9%) degraded, and 7 queries (6.2%) showed no change. The best observed improvement was +20.00%, while the worst degradation

Table 2: Performance summary for *Linear Scan only*. Negative values indicate regressions vs. baseline.

Metric	Value
Overall Median Improvement	−1.90%
Mean Improvement	−10.78%
Standard Deviation	30.54
Queries with Improvement	41 (36.3%)
Queries with Degradation	68 (60.2%)
Queries with No Change	4 (3.5%)
Best Improvement	+15.93%
Worst Degradation	−173.24%
25th Percentile (Q1)	−10.00%
75th Percentile (Q3)	+1.73%

was −203.57%. The interquartile range (Q1 = −6.25%, Q3 = +2.32%) shows that most queries fell in a narrow band near zero, with slight regressions being more common than improvements.

These results are consistent with expectations: prefetching is designed to hide I/O latency by overlapping CPU and disk operations. However, since the JOB dataset fits largely in memory, the prefetch calls provide little benefit and can even add overhead. As a result, most queries see neutral or slightly negative performance impact. Prefetching may show stronger gains on larger, I/O-bound datasets.

Table 3: Performance summary for *Prefetch only*. Negative values indicate regressions vs. baseline.

Metric	Value
Overall Median Improvement	−0.68%
Mean Improvement	−11.58%
Standard Deviation	36.34
Queries with Improvement	44 (38.9%)
Queries with Degradation	62 (54.9%)
Queries with No Change	7 (6.2%)
Best Improvement	+20.00%
Worst Degradation	−203.57%
25th Percentile (Q1)	−6.25%
75th Percentile (Q3)	+2.32%

4.3 Combined Optimizations (Prefetch + Linear)

Table 4 summarizes the effect of enabling both optimizations together. Unlike the individual cases, the combined setting shows a slight overall gain: the median improvement across queries is +1.52%. However, the mean improvement remains negative at −5.11% with a relatively high standard deviation of 33.57, indicating that while many queries benefit, others experience significant regressions.

Out of 113 queries tested, 68 queries (60.2%) improved, 41 queries (36.3%) degraded, and 4 queries (3.5%) showed no change. The best observed improvement was +34.91%, while the worst degradation was −186.96%. The interquartile range (Q1 = −4.00%, Q3 = +6.86%) shows that the majority of queries clustered around small but positive gains, although variability remains substantial.

These results suggest that combining both optimizations occasionally creates synergies that benefit more queries compared to enabling each individually. However, the wide spread of outcomes and the negative mean indicate that the combined path is still workload-dependent. For the JOB benchmark, which is memory-resident, overhead remains a limiting factor. Nevertheless, the combined mode demonstrates that controlled interaction between optimizations can sometimes improve overall performance, even if individual features alone regress.

Table 4: Performance summary for *Prefetch + Linear*. Negative values indicate regressions vs. baseline.

Metric	Value
Overall Median Improvement	+1.52%
Mean Improvement	−5.11%
Standard Deviation	33.57
Queries with Improvement	68 (60.2%)
Queries with Degradation	41 (36.3%)
Queries with No Change	4 (3.5%)
Best Improvement	+34.91%
Worst Degradation	−186.96%
25th Percentile (Q1)	−4.00%
75th Percentile (Q3)	+6.86%

5 CONCLUSION

This project provided hands-on experience with PostgreSQL's index access methods. We successfully added new GUC-controlled features for leaf-page prefetch and linear scan on tiny pages, validated them on IMDB/JOB workloads, and analyzed their effects. While performance gains were limited, the project deepened our understanding of DBMS internals, including buffer management, scan execution, and extensibility via GUCs.

ACKNOWLEDGMENTS

We thank the CSCI 543 teaching staff for providing the Docker environment, benchmark scripts, and guidance.

REFERENCES

[1] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670.

[2] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[3] PostgreSQL Development Group. 2025. PostgreSQL 17.4 Documentation. https://www.postgresql.org/.