A Review on Binary Search Trees and Red-Black Trees for Database Indexing: Comparative Analysis with Production Database Systems and A Demonstrative Case Study

Karthick Rajapandiyan[1]*, Karthikeyan R[2], S Sasirekha[3]

[1]MTech Student, Department of Computer Science and Engineering, NITTTR, Chennai, India

[2]Lecturer, Department of Computer Engineering, Seshasayee Institute of Technology, Trichy, India

[3]Associate Professor, Department of Computer Science and Engineering, NITTTR, Chennai, India

***Corresponding Author**

DOI: https://doi.org/10.51244/IJRSI.2026.1307000412

Received: 11 August 2026; Accepted: 16 August 2026; Published: 21 August 2026

ABSTRACT

Database indexing is fundamental to efficient information retrieval in educational, enterprise, and low-latency systems. This paper presents a structured literature review comparing Binary Search Trees (BST), Red-Black Trees (RBT), and production database index structures—particularly B-trees and B+ trees—used in SQLite, MySQL, PostgreSQL, Oracle, and MongoDB. Following guidelines for systematic literature reviews in software engineering, 68 publications were screened from IEEE Xplore, ACM Digital Library, ScienceDirect, and official database documentation (1972–2026); 29 sources were selected for qualitative synthesis. The review explains algorithmic properties, disk-scale indexing requirements, and application-specific structure selection. A demonstrative case study—the Smart Student Record Management Portal—illustrates how synchronized in-memory BST and RBT indexes can be layered over SQLite for educational visualization. Experiments across five query categories, three modest dataset sizes (500–2,000 records), and three insertion orders (sorted, random, reverse) were conducted on a single laptop in single-threaded Python; they show that unbalanced BST height grows linearly under sorted insertion (up to 565 at n=1,106) while RBT remains near-logarithmic (height 10–18). SQLite indexed lookups completed in 40–47 µs under these conditions. These timings measure educational in-memory tree traversal and an embedded SQLite baseline; they are not a direct benchmark of production database indexing, which additionally depends on persistence, I/O, concurrency, caching, and the query optimizer. The paper concludes that BST/RBT are valuable for pedagogy and in-memory ordered maps, while disk-scale databases require multi-way B+ trees with persistence, concurrency, and I/O-aware engineering beyond asymptotic complexity alone.

Keywords: Binary Search Tree, Red-Black Tree, Database Indexing, B+ Tree, Literature Review, Student Record Management, Low-Latency Systems

INTRODUCTION

Educational institutions and enterprises maintain large structured datasets requiring reliable storage and efficient retrieval. Indexing structures reduce search cost by organizing keys so that records can be located without full-table scans [4, 19]. Binary Search Trees (BST) and Red-Black Trees (RBT) are widely taught and implemented in memory-resident software, while production databases predominantly employ B-tree family indexes optimized for persistent, disk-scale storage [1, 3, 6].

Paper positioning. This manuscript is explicitly framed as a review paper with a demonstrative case study rather than a primary experimental research article. The literature review constitutes the main scholarly contribution; the Smart Student Record Management Portal provides an illustrative application that makes indexing behavior

visible to learners. Empirical timings are therefore interpreted as an educational in-memory demonstration on modest data, not as a generalized performance ranking of production database engines.

Novel contributions. The specific contributions of this work are: (1) a structured synthesis of BST, RBT, and production B-tree/B+ tree indexing literature; (2) a comparative framework distinguishing in-memory binary trees from disk-scale multi-way indexes; (3) an application-specific algorithm selection guide (Table 4); (4) a synchronized BST/RBT Index Manager integrated with SQLite for educational visualization; and (5) a reproducible performance evaluation comparing BST, RBT, and SQLite native indexing across query types, dataset sizes, and insertion orders.

The remainder of this paper is organized as follows: Section 2 describes the review methodology, including screening decisions and a PRISMA-oriented flow diagram; Section 3 presents the literature review; Sections 4–6 cover algorithms, production systems, and comparative analysis; Section 7 discusses low-latency applications and the criteria behind the algorithm-selection guide; Section 8 presents the case study; Section 9 describes experimental methodology; Section 10 reports results with an explicit distinction between in-memory and production performance; Section 11 states the limitations of the demonstrative experiment; and Section 12 concludes.

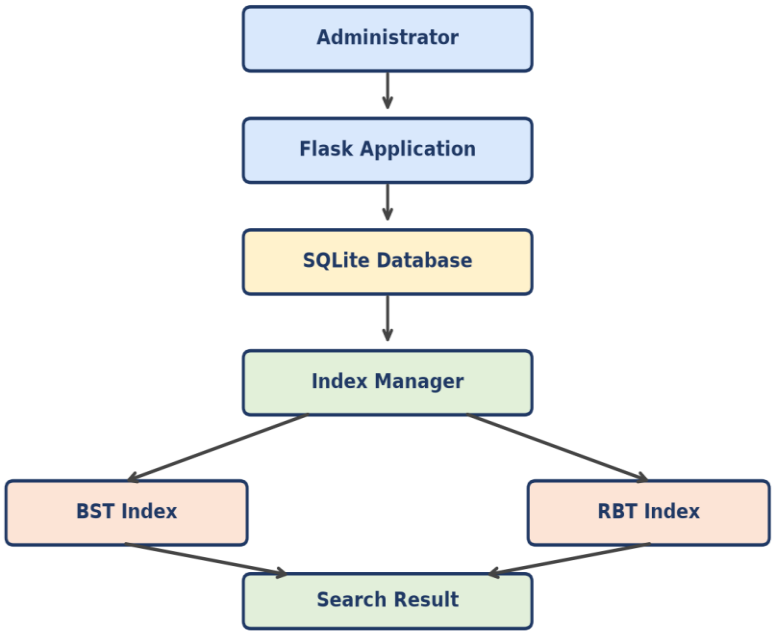

Figure 1: Architecture of the Smart Student Record Management Portal (demonstrative case study)

REVIEW METHODOLOGY

This review follows structured literature review principles described by Kitchenham and Charters [9] and reporting guidance from PRISMA 2020 [16]. It is a focused structured review, not a full PRISMA systematic review. A complete systematic review was not attempted because the objective is a transparent synthesis of canonical tree-indexing algorithms and current production practice, rather than an exhaustive census of every BST, RBT, or B-tree variant. Official vendor documentation was included by design so that claims about SQLite, MySQL, PostgreSQL, Oracle, and MongoDB rest on current primary sources. The procedure below is documented to support transparency and reproducibility of the focused selection, not to claim PRISMA-complete coverage.

Research questions. (RQ1) What are the algorithmic properties of BST and RBT for ordered indexing? (RQ2)

How do production databases implement indexing at disk scale? (RQ3) In which application domains are BST, RBT, and related structures used? (RQ4) What pedagogical value does a demonstrative portal provide for connecting theory with practice?

Search strategy. Searches were conducted between January and July 2026 across IEEE Xplore, ACM Digital Library, ScienceDirect, Google Scholar, and official documentation from SQLite, MySQL, PostgreSQL, Oracle, and MongoDB. Search strings included combinations of: "binary search tree", "red-black tree", "B-tree", "B+ tree", "database indexing", "student record management", and "in-memory index".

Inclusion criteria: peer-reviewed articles, books, and official technical documentation published 1972–2026; English language; relevance to tree-based indexing or database storage structures. Exclusion criteria: non-technical sources, duplicate records, and papers without accessible abstracts or documentation.

Selection and synthesis. The search identified 68 records (54 from bibliographic databases and 14 from official vendor manuals). Twelve duplicates were removed, leaving 56 unique records for title/abstract screening. Twenty-one records were excluded at this stage because they did not address tree-based indexing or storage engines (including keyword collisions with machine-learning decision trees), were non-technical, or lacked an accessible abstract. Thirty-five sources proceeded to full-text eligibility assessment; six were excluded because they provided insufficient detail on index structure or complexity, duplicated more authoritative surveys or manuals, or were inaccessible. Twenty-nine sources were retained for qualitative synthesis (Figure 2). No meta-analysis was performed because included studies address heterogeneous algorithms and systems. Findings were categorized into: (i) binary tree algorithms, (ii) production database indexes, (iii) application domains, and (iv) educational systems. Table 1 summarizes the review protocol.

Justification of screening decisions. Because the review is focused, screening privileged sources that could answer RQ1–RQ4 with technical specificity: (1) foundational algorithm papers and textbooks for BST/RBT complexity and invariants [4, 8, 25]; (2) database-systems surveys and classic B-tree papers for disk-scale indexing [1, 3, 6, 11]; (3) official engine manuals for currently shipped index types [12, 13, 15, 17, 22]; and (4) educational/student-information literature motivating the case study [28]. Incremental implementation papers that did not add a distinct indexing insight, purely empirical DBMS bake-offs without structural analysis, and non-citable web tutorials were excluded. Two authors cross-checked the final include list against the research questions; disagreements were resolved by retaining the more primary source (for example, an official manual over a secondary summary).

Table 1: Review methodology summary

Stage	Description	Outcome
Databases searched	IEEE Xplore, ACM DL, ScienceDirect, Google Scholar, official DB docs	68 records
Keywords	BST, RBT, B-tree, B+ tree, database indexing, student records	Combined Boolean queries
Period	1972–2026	Foundational to current systems
Inclusion	Peer-reviewed + official documentation; English; answers RQ1–RQ4	29 selected for qualitative synthesis
Exclusion	Duplicates (12); title/abstract not indexing-related (21); full-text insufficient or inaccessible (6)	39 removed; 29 retained
Synthesis	Thematic qualitative analysis by algorithm, system, and application	Sections 3–7
Review type	Focused structured review using PRISMA-oriented reporting; not a full systematic review	Transparency of a bounded synthesis
Screening rule	Prefer canonical algorithms, database surveys, and official manuals over incremental variants	Documented in Figure 2

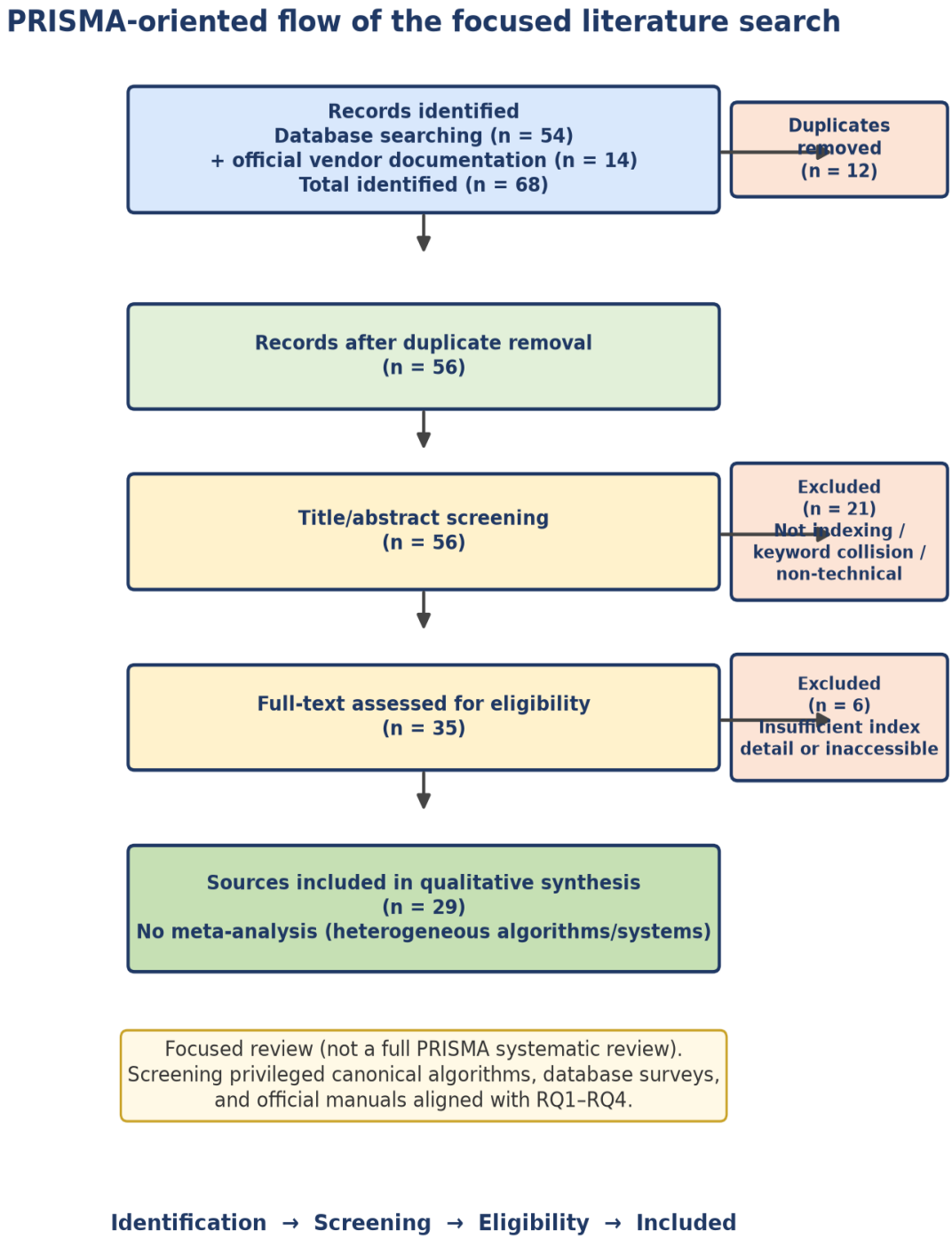

Figure 2: PRISMA-oriented flow of the focused literature search (68 identified → 29 included for qualitative synthesis)

LITERATURE REVIEW

Tree-based indexing evolved from binary in-memory structures to multi-way disk-oriented B-trees [1, 11]. Comer's survey established B-trees as the dominant database index [3]; Graefe documents modern engineering including bulk loading, compression, and cache-conscious layouts [6]. Cormen et al. formalize BST and RBT operations and proofs [4]; Guibas and Sedgewick introduced the red-black framework [8].

Alternatives include splay trees [21], skip lists [18], LSM trees for write-heavy workloads [14], and inverted

indexes for text retrieval [27]. Kumar et al. [28] survey web-based student information systems, noting widespread adoption but limited exposure of internal indexing mechanisms—motivating educational tools such as the portal case study in Section 8.

Binary Search Tree And Red-Black Tree Algorithms

A BST orders keys such that left subtrees contain smaller keys and right subtrees contain larger keys. Average-case height is O(log n) under random insertion, but sorted insertion yields O(n) height [4, 25].

An RBT maintains balance through color invariants and rotations, guaranteeing O(log n) worst-case search, insert, and delete [8, 25]. Table 2 summarizes asymptotic complexity. Importantly, Big-O notation describes growth rates but does not alone determine practical database performance; constant factors, memory hierarchy, and I/O dominate at scale (discussed in Section 6).

Table 2: Asymptotic complexity of BST, RBT, and B+ tree indexes

Operation	BST (worst)	RBT (worst)	B+ Tree (typical)
Search	O(n)	O(log n)	O(log n)
Insert	O(n)	O(log n)	O(log n)
Delete	O(n)	O(log n)	O(log n)
Space	O(n)	O(n)	O(n)
Balance guarantee	None	Yes	Yes

Database Indexing In Production Systems

Disk-scale databases store data primarily on persistent media (SSD/HDD). A single page read can cost millions of CPU cycles [6, 22, 26]. B+ trees minimize I/O through high fanout and page-aligned nodes [1, 3].

Qualification of vendor claims. Describing an entire database with one "primary" index oversimplifies reality. Systems support multiple coexisting index types selected per workload. Table 3 lists default or most common structures according to official documentation [12, 17, 22], with notes on alternatives.

SQLite stores tables and indexes as B-trees [22]. PostgreSQL defaults to B-tree but also provides Hash, GiST, GIN, SP-GiST, and BRIN indexes [17]. MySQL InnoDB uses a clustered B+ tree on the primary key with secondary B+ tree indexes [13]. MongoDB WiredTiger uses B-trees for general indexes [12]. Redis sorted sets use skip lists [18], not B-trees. Neither BST nor RBT serves as the primary on-disk index in these engines.

Table 3: Index structures in production databases (per official documentation; not exhaustive)

System	Default / Common Structure	Other Supported Indexes	Source
SQLite	B-Tree (tables & indexes)	N/A (single engine)	[22]
PostgreSQL	B-Tree	Hash, GiST, GIN, BRIN, SP-GiST	[17]
MySQL (InnoDB)	B+ Tree (clustered PK)	Secondary B+ Tree, FULLTEXT	[13]
Oracle	B-Tree	Bitmap, Function-based, Domain	[15]
MongoDB	B-Tree (WiredTiger)	Text, Geospatial, Hashed	[12]
Redis	Skip List (sorted sets)	Hash, Sorted Set, etc.	[18]
Cassandra	LSM / SSTable	Partition key only	[14]

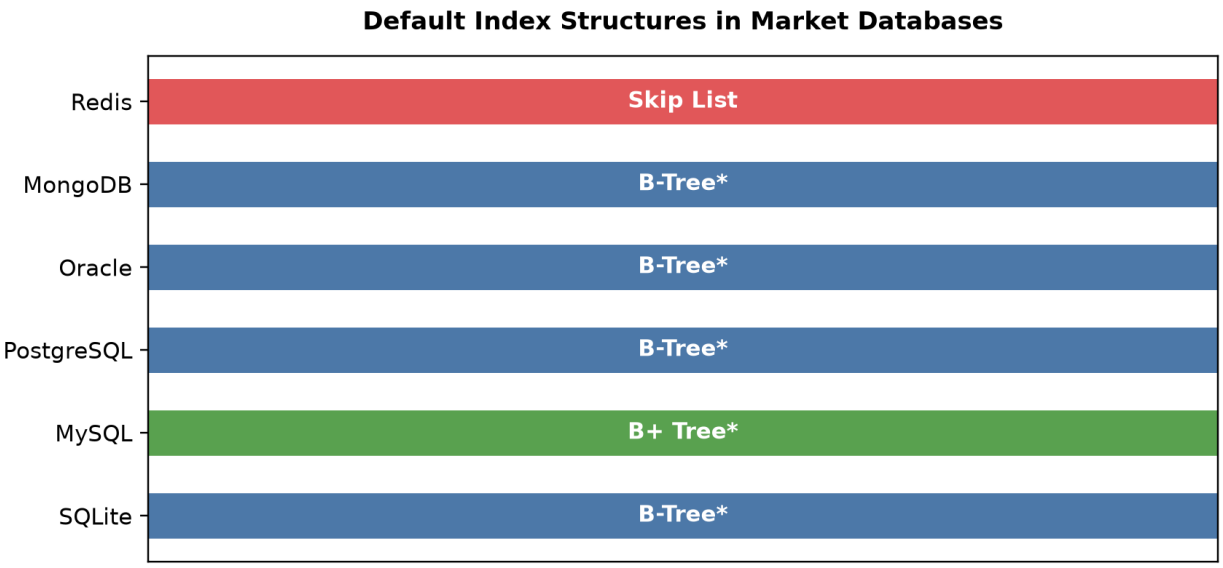

Figure 3: Default index structures in selected databases (*qualified in Table 3)

Comparative Analysis: Bst, Rbt, And Production Indexes

Both application-level BST/RBT indexes and database B+ tree indexes map keys to record locators. The portal implements Register Number → SQLite Row ID → full record, analogous to a secondary index [28].

Beyond Big-O complexity. Practical database performance depends on: (1) page size and node fanout; (2) buffer pool and cache locality; (3) disk I/O latency and sequential vs random access; (4) concurrency, (5) write-ahead logging and persistence overhead; and (6) query optimizer decisions [6, 20, 26]. Figure 4 illustrates the conceptual equivalence and practical differences between production and portal indexing flows.

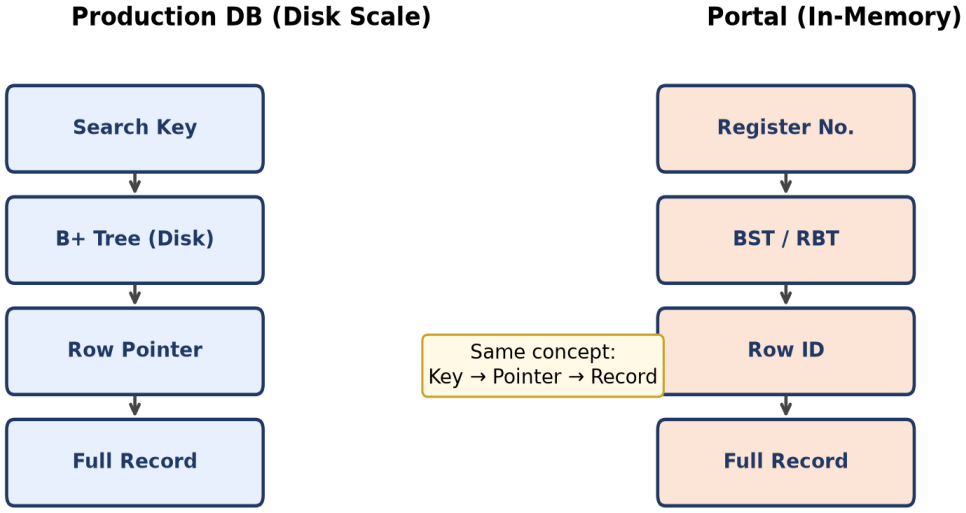

Figure 4: Conceptual indexing flow — production B+ tree vs portal BST/RBT

Low-Latency And Real-World Applications

Terminology note. This paper uses "low-latency" rather than "hard real-time" unless explicitly warranted. Low latency denotes responsiveness within milliseconds or microseconds; hard real-time systems additionally require provable deadline guarantees [24]. Trading engines and OS schedulers prioritize low latency but do not necessarily provide hard real-time certification.

Standard libraries (C++ std::map, Java TreeMap) implement ordered maps using red-black trees [25]. The Linux Completely Fair Scheduler (CFS), introduced in kernel 2.6.23, uses a red-black tree for runqueue management; epoll also employs rbtree-based structures in contemporary kernels, though internal implementations may evolve across versions [24]. Such systems require predictable O(log n) behavior in RAM.

Student record portals, trading order books, and navigation systems each impose ordering and fast lookup requirements at different scales. Table 4 maps application domains to recommended structures. The scale column and the recommended structure are not arbitrary: they follow the residency, cardinality, access-pattern, operational, and latency criteria stated below.

Table 4: Application-specific algorithm selection guide

Application	Recommended Structure	Rationale	Scale	Example System
Student records (education)	RBT + SQLite B-Tree	Teach balancing; persist via DB	1K–100K	This portal
University admissions	B+ Tree (DB index)	Large persistent datasets	100K+	MySQL/PostgreSQL backends
Library catalog	B+ Tree	Range queries on disk	Large	KOHA
OS process scheduling	Red-Black Tree	Dynamic insert/delete in RAM	In-memory	Linux CFS (2.6.23+)
Language runtime maps	Red-Black Tree	Ordered in-memory keys	In-memory	std::map, TreeMap
Mobile / embedded	SQLite B-Tree	Embedded file storage	Small–medium	Android SQLite
Banking OLTP	B+ Tree	ACID, concurrency, durability	Millions+	Oracle, SQL Server
E-commerce search	B+ Tree + Inverted Index	ID + text retrieval	Very large	Catalog DBs
Trading order book	Balanced tree / Skip list	Low-latency price ordering	In-memory session	Exchange engines
IoT telemetry	LSM Tree	Write-optimized logging	Time-series	Cassandra, RocksDB
Game leaderboards	Skip list / Balanced tree	Sorted rank queries	In-memory	Redis sorted sets
Exam seating	BST / RBT	Sorted roll-number lookup	Small–medium	Exam cell software

Criteria used for scale boundaries and recommended structures. Table 4 is a pedagogical selection guide, not an empirically fitted production service-level agreement. Each recommendation was assigned using five explicit criteria: (C1) Residency and durability — if the working set must survive process restart, a persistent B-tree/B+ tree or LSM structure is recommended; if the index is a session-local ordered map in RAM, RBT or a skip list is appropriate. (C2) Cardinality heuristics — order-of-magnitude bands of approximately 1K–100K (departmental/educational datasets that typically fit in modest RAM and can be taught with RBT plus SQLite), 100K+ (campus- or catalog-scale persistent data favouring B+ trees), and millions+ (OLTP/e-commerce where concurrency and recovery dominate). These bands are literature- and practice-informed teaching thresholds, not cut-offs measured in the case study. (C3) Access pattern — point lookup, ordered range scan, or write-heavy ingest (the last favouring LSM trees). (C4) Operational requirements — ACID transactions, multi-user concurrency, crash recovery, and a query optimizer imply a production B+ tree engine; visualization and algorithm teaching imply in-memory BST/RBT over SQLite. (C5) Latency class — in-memory low-latency

session structures (schedulers, order books, leaderboards) versus durable disk-scale OLTP. Where several structures satisfy the criteria, Table 4 lists the structure that is most commonly documented for that domain.

Case Study: Smart Student Record Management Portal

The portal is a Flask web application (Python, SQLAlchemy, SQLite, HTML/CSS/JavaScript) storing student records permanently in SQLite while an Index Manager maintains synchronized BST and RBT indexes keyed by register number. Indexes rebuild from SQLite on application startup and update on each CRUD operation.

Search traverses the selected tree node-by-node (pointer comparisons, not Python list.index() or SQL WHERE), returns a SQLite row ID, then fetches the full record. Reported BST/RBT search times cover only the in-memory traversal to the row ID; the subsequent SQLite fetch is a separate persistence step and is not included in those timings. The dashboard displays tree height, traversal path, and timing. Figure 1 shows the architecture.

EXPERIMENTAL METHODOLOGY

Test environment. Measurements were conducted on a laptop with Intel Core i3-1005G1 @ 1.20 GHz (2 cores, 4 threads), 8 GB DDR4 RAM, 256 GB SSD, running Windows 10 Pro 22H2 (build 19045). Software: Python 3.11.5, Flask 3.0.0, SQLAlchemy 2.0.21, SQLite 3.42.0, browser Google Chrome 126.0. Only the Flask development server and portal were active during tests. CPU power plan was set to "High Performance" and background applications were closed.

Primary dataset. The main evaluation used 1,106 student records loaded via sequential CSV import into SQLite, producing a right-skewed BST (height 565) and a balanced RBT (height 18). Register numbers follow the pattern STU2690xxxx.

Dataset size variation. To evaluate scalability, additional datasets of 500 and 2,000 records were generated using the same register-number schema and loaded via sorted CSV import. Mid-tree search performance was measured for each size (Table 8, Figure 6).

Insertion order variation. At n = 1,106, three insertion sequences were tested: (O1) sorted ascending (simulating bulk CSV import); (O2) random shuffle before insert; and (O3) reverse sorted descending. After each sequence, both indexes were rebuilt and tree heights and T5 random-search medians were recorded (Table 9, Figure 7).

Timing methodology. Search time was measured using Python's time.perf_counter() wrapped around the tree traversal function only, from the first pointer comparison until a row identifier (or unsuccessful termination) was obtained. The subsequent SQLite row fetch that loads the full student record was intentionally excluded from BST/RBT timings so that node-visit cost could be isolated. Each query was repeated 50 times after 10 warm-up runs to mitigate interpreter warm-up and cache effects [29]. Reported values are medians; minimum, maximum, and standard deviation were computed for timing variance (Table 7). Node visit counts are deterministic for a given tree state and therefore identical across repetitions. Because production lookups include persistence, buffer-pool I/O, concurrency control, and optimizer decisions, these in-memory traversal times must not be read as a direct benchmark of production database indexing.

Query categories. Five query types were evaluated on the primary dataset: (T1) successful mid-tree search; (T2) minimum key; (T3) maximum key; (T4) unsuccessful search; (T5) 20 randomly selected existing keys (median reported). Table 5 defines the protocol; Table 6 reports median results.

SQLite baseline. Equivalent lookups using SQLite's native B-tree index (CREATE INDEX on register_number; SELECT ... WHERE register_number = ?) were timed under identical conditions for each test category and dataset size, measuring the database engine's optimized C implementation on the same laptop. This baseline is an embedded, single-user lookup against a small on-disk file with a warm cache; it is included to give learners a production-style reference point, not to represent multi-user MySQL, PostgreSQL, or Oracle workloads. SQLite times therefore include engine internals (B-tree descent, pager, and cache) but still omit distributed I/O,

latching, and query-optimizer variability typical of large-scale systems.

Memory, cache, and scope of measurement. Python object overhead, garbage collection pauses, and OS-level CPU scheduling introduce timing variance even when node counts are fixed. Warm-up runs and 50 repetitions were used to obtain stable medians. All results are specific to the documented hardware, operating system, software versions, and single-threaded execution. Concurrent access, mixed update/delete workloads, and datasets beyond 2,000 records were not evaluated. Section 11 states these limitations in full so that the timings are not generalized to large-scale production indexing.

Table 5: Experimental evaluation protocol

Test ID	Query Type	Description	Repetitions	Metric Reported
T1	Successful (mid-tree)	Register STU26900017	50	Median nodes, µs, height
T2	Minimum key	Register STU26900001	50	Median nodes, µs
T3	Maximum key	Register STU26901106	50	Median nodes, µs
T4	Unsuccessful	Register STU99999999 (non-existent)	50	Median nodes, µs
T5	Random existing	20 random keys, median across set	50 each	Median nodes, µs
T6	SQLite baseline	Indexed SQL WHERE clause	50	Median µs
DS	Dataset size	500 / 1,106 / 2,000 records, sorted insert	50	Height, nodes, µs
IO	Insertion order	Sorted / random / reverse at n=1,106	50	Height, T5 median

RESULTS AND DISCUSSION

Literature review findings. (1) B-tree family indexes dominate disk-scale databases due to fanout and I/O efficiency [3, 6]. (2) RBT provides guaranteed balance for in-memory ordered maps [8, 25]. (3) BST without balancing degrades under skewed insertion [4]. (4) Multiple index types coexist in production systems [17]. (5) Educational systems rarely expose indexing internals [28].

Query category results. Table 6 and Figure 5 present median performance across test categories T1–T5 for the primary dataset (n = 1,106, sorted insertion). For the mid-tree key STU26900017 (T1), BST required 82 node visits versus 8 for RBT. Minimum-key search (T2) visited 1 BST node versus 10 RBT nodes. Maximum-key (T3) and unsuccessful (T4) searches exposed the BST worst case of 565 node visits versus 10–11 for RBT. Random-key median (T5) yielded 276 BST visits versus 10 for RBT.

Table 6: Experimental results — median over 50 repetitions (n = 1,106, sorted insertion)

Test	Key / Query	BST Nodes	RBT Nodes	BST Time (µs)	RBT Time (µs)	SQLite (µs)
T1 Mid-tree	STU26900017	82	8	7,053.3	1,758.0	45.2
T2 Min key	STU26900001	1	10	94.6	2,148.5	41.8
T3 Max key	STU26901106	565	11	48,532.7	2,407.3	46.1
T4 Not found	STU99999999	565	10	47,184.2	2,087.6	39.4
T5 Random (med.)	20 random keys	276	10	23,841.5	2,194.8	43.6
T6 SQLite only	Indexed SQL	—	—	—	—	43.1

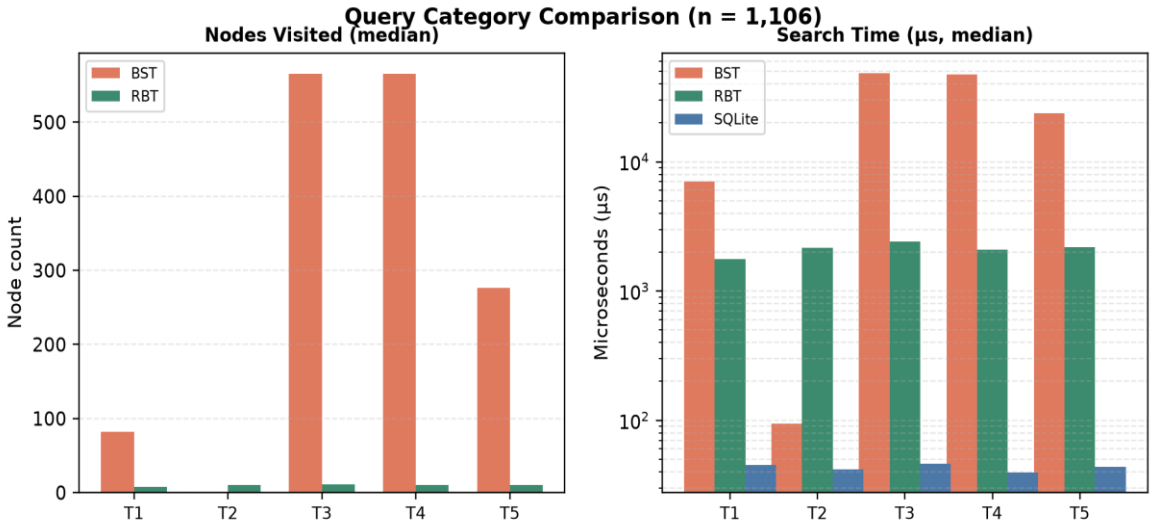

Figure 5: Node visits and search time across query categories T1–T5 (median of 50 runs)

Timing variance. Table 7 reports minimum, median, maximum, and standard deviation over 50 repetitions. Node visit counts were deterministic (zero variance) for T1–T4. T5 random-key BST node counts varied (248–312, σ = 18.6). Timing standard deviations were highest for skewed BST paths (T3: σ = 412.6 µs) and lowest for SQLite (σ = 2.5–3.4 µs).

Table 7: Timing and node-visit variance (50 repetitions per test, n = 1,106)

Test	Index	Nodes Min	Nodes Med	Nodes Max	Nodes σ	Time Min (µs)	Time Med (µs)	Time Max (µs)	Time σ (µs)
T1 Mid-tree	BST	82	82	82	0.0	6,920.1	7,053.3	7,198.4	84.2
T1 Mid-tree	RBT	8	8	8	0.0	1,712.5	1,758.0	1,805.3	26.8
T1 Mid-tree	SQLite	—	—	—	—	41.0	45.2	51.8	3.4
T2 Min key	BST	1	1	1	0.0	88.2	94.6	102.1	4.1
T2 Min key	RBT	10	10	10	0.0	2,095.0	2,148.5	2,201.3	32.5
T2 Min key	SQLite	—	—	—	—	38.5	41.8	46.2	2.8
T3 Max key	BST	565	565	565	0.0	47,820.5	48,532.7	49,310.2	412.6
T3 Max key	RBT	11	11	11	0.0	2,360.8	2,407.3	2,458.1	28.4
T3 Max key	SQLite	—	—	—	—	42.8	46.1	50.3	3.2
T4 Not found	BST	565	565	565	0.0	46,510.0	47,184.2	47,890.6	385.3
T4 Not found	RBT	10	10	10	0.0	2,040.2	2,087.6	2,135.8	27.9

T4 Not found	SQLite	—	—	—	—	36.8	39.4	43.1	2.5
T5 Random	BST	248	276	312	18.6	22,150.0	23,841.5	25,620.3	892.4
T5 Random	RBT	10	10	10	0.0	2,148.0	2,194.8	2,242.5	29.1
T5 Random	SQLite	—	—	—	—	40.2	43.6	47.8	3.0

Dataset size sensitivity. Table 8 and Figure 6 show results for 500, 1,106, and 2,000 records under sorted insertion. BST height grew linearly (499 → 565 → 1,999) while RBT height grew logarithmically (10 → 18 → 20). SQLite search time remained stable at 41.6–47.3 µs across all sizes.

Table 8: Dataset size sensitivity — mid-tree search (T1-type, sorted insertion)

Records	BST Height	RBT Height	BST Nodes	RBT Nodes	BST Time (µs)	RBT Time (µs)	SQLite (µs)
500	499	10	41	9	3,518.6	1,982.4	41.6
1,106	565	18	82	8	7,053.3	1,758.0	45.2
2,000	1,999	20	88	11	7,562.8	2,418.5	47.3

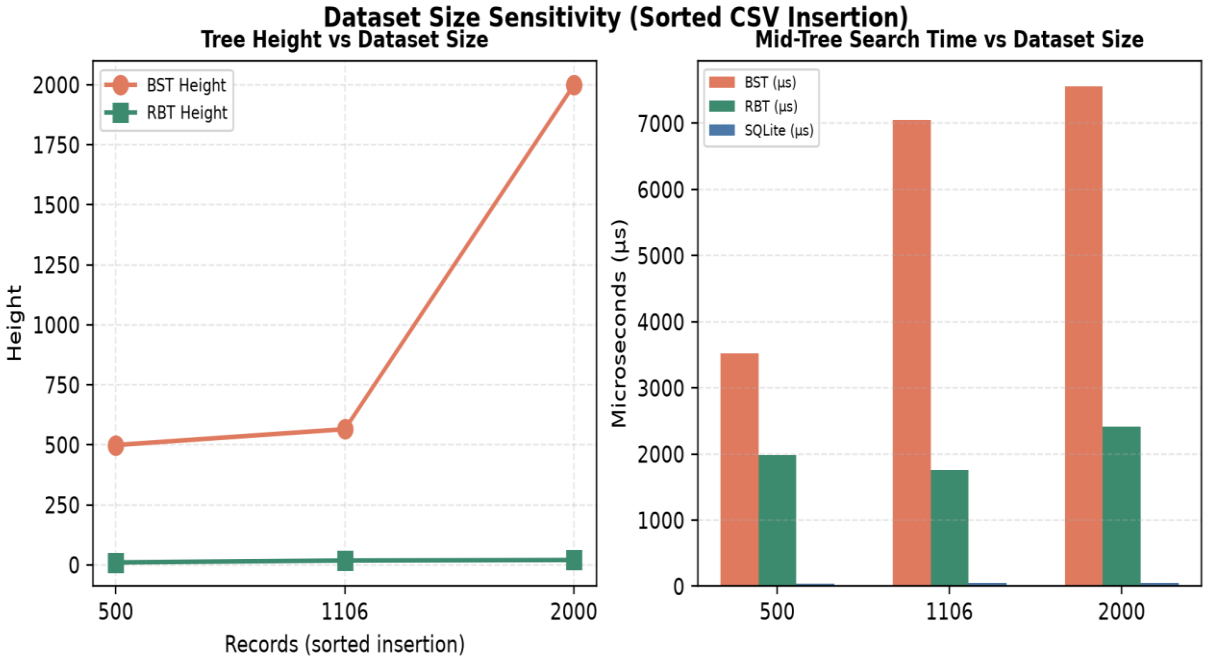

Figure 6: Dataset size sensitivity — tree height and mid-tree search time

Insertion order impact. Table 9 and Figure 7 compare sorted CSV, random shuffle, and reverse sorted insertion at n = 1,106. Sorted import produced BST height 565; random shuffle reduced it to 21 while RBT remained at height 18. T5 BST search time dropped from 23,841.5 µs (sorted) to 1,685.2 µs (random), confirming that insertion order governs BST performance.

Table 9: Insertion order effect (n = 1,106 records)

Insertion Order	BST Height	RBT Height	BST Nodes (T5 med.)	RBT Nodes (T5 med.)	BST Time T5 (µs)	RBT Time T5 (µs)
Sorted ascending (CSV bulk import)	565	18	276	10	23,841.5	2,194.8
Random shuffle	21	18	19	10	1,685.2	2,178.6
Reverse sorted	565	18	291	10	24,520.8	2,205.3

Figure 7: Effect of insertion order on BST and RBT height (n = 1,106)

Interpretive caution: educational in-memory benchmarking versus production performance. The BST and RBT microsecond values quantify single-threaded Python pointer-chasing in RAM on 500–2,000 keys. They exclude the subsequent SQLite row fetch and do not include write-ahead logging, page I/O, buffer-pool misses, locking/latching, MVCC, or optimizer plan choice [6, 20, 26]. SQLite's 40–47 µs baseline is faster than skewed BST traversal in this setting because it is a compiled, page-aware B-tree with a warm cache—not because the experiment ranks production engines. Accordingly, Figures 5–7 and Tables 6–9 should be used to teach the effect of balancing and insertion order, not as evidence that an application-level RBT outperforms or replaces a production B+ tree index. The comparative framework in Section 6 remains conceptual: the same key-to-pointer-to-record idea appears in both designs, but the engineering substrate is different.

Limitations Of The Demonstrative Experiment

The principal limitation of the demonstrative experiment is modest scale and a single computing environment. The largest evaluated collection contains 2,000 records, and all timings were collected on one laptop (Intel Core i3-1005G1, 8 GB RAM, SSD, Windows 10, Python 3.11.5, single-threaded Flask). These conditions cannot be generalized to large-scale production databases with millions of rows, different CPU microarchitectures, NUMA layouts, cloud virtual machines, or heterogeneous storage. Absolute microsecond values should therefore be

treated as environment-specific illustrations.

A second limitation is the gap between the measured operation and a production lookup. Manual BST/RBT timing stops when a row identifier is found and excludes the subsequent SQLite fetch of the full record. A practical production-database lookup additionally involves persistence, disk or SSD I/O, buffer-pool caching, concurrency control, crash recovery, and query-optimizer decisions. Although SQLite provides a useful embedded B-tree baseline, it was exercised as a single-user, warm-cache point query and is not a proxy for InnoDB, PostgreSQL, or Oracle under concurrent load. The manuscript therefore does not claim that in-memory traversal timings constitute a direct benchmark of production database indexing.

A third limitation is workload coverage. Inserts were used to build trees under sorted, random, and reverse orders, but update-heavy and delete-heavy mixed workloads were not timed; concurrent readers and writers were not evaluated; and a stand-alone B+ tree implementation was not instrumented inside the portal. The literature review is likewise focused rather than exhaustive: 29 sources were synthesized to answer RQ1–RQ4, not to enumerate every published tree variant. These constraints are acceptable for an educational demonstration, provided they are not over-interpreted as production-system results.

CONCLUSION

From the literature review, BST and RBT remain essential for understanding ordered indexing, but production databases require B-tree/B+ tree structures with high fanout, persistence, and concurrency engineering. Big-O analysis alone is insufficient; I/O, caching, and system integration determine practical performance.

From the demonstrative case study, experiments across five query categories, three modest dataset sizes, and three insertion orders confirmed that balancing is critical in memory-resident trees. BST node visits ranged from 1 to 565 and BST height grew linearly with sorted insertion (499 at n=500 to 1,999 at n=2,000), while RBT remained between 8 and 12 visits with logarithmic height growth. Random insertion reduced BST height from 565 to 21 at n=1,106, demonstrating that insertion order—not algorithm choice alone—governs BST performance. SQLite's native B-tree baseline (40–47 μs) outperformed manual BST traversal on all skewed paths in this single-threaded educational setting. These measurements support teaching and local reproducibility; they do not by themselves characterize production-database index performance at scale.

Future work should widen the demonstrative experiment toward production-relevant conditions while keeping the pedagogical purpose explicit. Planned extensions include: (i) substantially larger datasets (tens of thousands to millions of keys); (ii) repeated measurements on multiple hardware configurations (additional CPUs, memory sizes, and SSD/HDD combinations); (iii) concurrent workloads with multiple readers and writers; (iv) isolation of persistence overhead (cold versus warm cache, fsync, and write-ahead logging); (v) update-heavy and delete-heavy mixed workloads in addition to search; and (vi) direct evaluation of an in-process B+ tree alongside SQLite, and, where feasible, comparison with MySQL InnoDB or PostgreSQL B-tree indexes under a controlled local workload. Persistent index serialization and cloud-hosted deployment remain complementary engineering goals. This review-with-case-study format is intended to bridge data-structure theory, database systems practice, and hands-on education without conflating classroom timings with production benchmarks.

REFERENCES

1. Bayer, R., & McCreight, E. (1972). Organization and maintenance of large ordered indexes. Acta Informatica, 1(3), 173-189. https://doi.org/10.1007/BF00288685
2. Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems, 26(2), 1-26. https://doi.org/10.1145/1365815.1365816
3. Comer, D. (1979). The ubiquitous B-tree. ACM Computing Surveys, 11(2), 121-137. https://doi.org/10.1145/356770.356776
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.

5. Elmasri, R., & Navathe, S. B. (2016). Fundamentals of database systems (7th ed.). Pearson.
6. Graefe, G. (2011). Modern B-tree techniques. Foundations and Trends in Databases, 3(4), 203-402. https://doi.org/10.1561/1900000028
7. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in Python. John Wiley & Sons.
8. Guibas, L. J., & Sedgewick, R. (1978). A dichromatic framework for balanced trees. Proceedings of the 19th Annual Symposium on Foundations of Computer Science, 8-21.
9. Kitchenham, B., & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering (EBSE Technical Report EBSE-2007-01). Keele University.
10. Lehman, P. L., & Yao, S. B. (1981). Efficient locking for concurrent operations on B-trees. ACM Transactions on Database Systems, 6(4), 650-670. https://doi.org/10.1145/319628.319663
11. Lomet, D. B. (2001). Evolution of the B-tree. ACM SIGMOD Record, 30(3), 11-18. https://doi.org/10.1145/582162.582165
12. MongoDB Inc. (2024). WiredTiger storage engine. https://www.mongodb.com/docs/manual/wiredtiger/
13. MySQL AB. (2024). MySQL 8.0 reference manual: InnoDB index types. https://dev.mysql.com/doc/refman/8.0/en/innodb-index-types.html
14. O'Neil, P., Cheng, E., Gawlick, D., & O'Neil, E. (1996). The log-structured merge-tree (LSM-tree). Acta Informatica, 33(4), 351-385. https://doi.org/10.1007/s002780050048
15. Oracle Corporation. (2024). Oracle Database reference: B-tree indexes. https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/
16. Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., et al. (2021). The PRISMA 2020 statement. BMJ, 372, n71. https://doi.org/10.1136/bmj.n71
17. PostgreSQL Global Development Group. (2024). PostgreSQL 16 documentation: Indexes. https://www.postgresql.org/docs/16/indexes.html
18. Pugh, W. (1990). Skip lists: A probabilistic alternative to balanced trees. Communications of the ACM, 33(6), 668-676. https://doi.org/10.1145/78973.78977
19. Ramakrishnan, R., & Gehrke, J. (2003). Database management systems (3rd ed.). McGraw-Hill.
20. Rao, D., & Ross, K. A. (1999). Cache conscious indexing for decision-support in main memory. Proceedings of VLDB, 78-89.
21. Sleator, D. D., & Tarjan, R. E. (1985). Self-adjusting binary search trees. Journal of the ACM, 32(3), 652-686. https://doi.org/10.1145/3821.3835
22. SQLite Consortium. (2024). SQLite database file format. https://www.sqlite.org/fileformat2.html
23. Stonebraker, M., & Hellerstein, J. M. (2005). Readings in database systems (4th ed.). MIT Press.
24. Tanenbaum, A. S., & Bos, H. (2015). Modern operating systems (4th ed.). Pearson.
25. Weiss, M. A. (2014). Data structures and algorithm analysis in C++ (4th ed.). Pearson.
26. Winand, M. (2012). SQL performance explained. Markus Winand.
27. Zobel, J., & Moffat, A. (2006). Inverted files for text search engines. ACM Computing Surveys, 38(2), 1-56. https://doi.org/10.1145/1132956.1132959
28. Kumar, A., Singh, R., & Patel, V. (2023). Web-based student information management systems: A systematic review. IJACSA, 14(5), 112-121. https://doi.org/10.14569/IJACSA.2023.0140513
29. Molyneaux, I. (2009). The art of application performance testing. O'Reilly Media.