



ADVANCES IN INDEXING TECHNIQUES FOR DATABASE SYSTEMS A SYSTEMATIC LITERATURE REVIEW

Ionut-Adrian TUDOROIU

Valahia University of Targoviste, Faculty of Electrical Engineering, Electronics, and Information Technology
E-mail: adrian.tudoroiu@valahia.ro

Abstract. *Indexing is one of the most important mechanisms for improving query performance in database systems. Without a proper index, even simple queries can become extremely slow when dealing with large amounts of data. This paper reviews the main indexing techniques used in modern databases from classical tree-based and hash-based structures to bitmap indexes and distributed approaches suited for big data environments. For each technique, the underlying principles, performance characteristics, and limitations are analysed. The paper also includes a comparison table and a discussion of recent trends, such as learned index structures and AI-assisted query optimization. The main takeaway is straightforward: there is no universally best index the right choice always depends on the workload, the data, and the system architecture.*

Keywords: database indexing, B-Tree, B+ Tree, hash indexing, bitmap index, inverted index, NoSQL, learned indexes.

1. INTRODUCTION

When working with large databases, one of the first questions that comes up in practice is: how do we find data fast? A full table scan reading every row in a table works fine for small datasets, but it becomes completely impractical at scale. A table with a hundred million rows cannot afford to be scanned for every query. This is exactly the problem that indexes are designed to solve.

A database index is, at its core, an auxiliary data structure that tells the system where to find specific records without having to look at everything. The analogy with a book index holds well: you do not read every page to find a topic, you look it up directly. In databases, the same idea applies but the challenge is that different access patterns require different index structures. A B+ Tree is great for range queries, a hash index is faster for exact lookups, a bitmap index shines in analytical workloads, and inverted indexes are the backbone of any full-text search engine. Choosing the wrong one can hurt performance just as much as having no index at all.

This paper covers the main indexing techniques in use today, compares them across key dimensions, and discusses where the field is heading. For a broader architectural perspective on how indexing fits within the overall database engine, see Hellerstein et al. (2007).

2. FUNDAMENTALS OF DATABASE INDEXING

Before getting into specific structures, it helps to clarify a few basic concepts. A search key is the attribute (or combination of attributes) that an index is built on.

The index maps each key value to the location of the corresponding record, so the system can jump directly to it instead of scanning. The formal definition and theoretical properties of index structures are covered in depth in the classical literature on relational systems (Date, 2003).

Indexes are generally classified as primary or secondary. A primary index is built on the attribute by which the data is physically sorted on disk — since the file is already ordered, the index can be sparse (one entry per disk block is enough). A secondary index is built on any other attribute; since records are not physically ordered by that key, the index must be dense — one entry per record. Secondary indexes add flexibility but come with higher storage and maintenance costs (Ramakrishnan & Gehrke, 2003).

The clustered vs. non-clustered distinction is also important. A clustered index means the data rows are physically stored in the order of the index key, so range scans are fast and efficient. A non-clustered index is a separate structure pointing to the rows, which may be scattered across disk — useful for selective queries, less ideal for large scans (Graefe, 2011). A table can have only one clustered index, but multiple non-clustered ones.

3. TREE-BASED INDEXING TECHNIQUES

3.1. B-TREE

The B-Tree, proposed by Bayer and McCreight (1972), is a self-balancing search tree where all leaf nodes sit at the same depth. This balance ensures that search, insertion, and deletion all run in $O(\log n)$ time regardless of how the keys are distributed. What makes it practical for databases is its high branching factor — each node can hold many keys, so the tree stays shallow even for very large datasets. A B-Tree of order 100 can cover over a million records in just three levels, meaning at most three disk reads per lookup (Knuth, 1998).

3.2. B+ TREE

The B+ Tree improves on the B-Tree by moving all data pointers to the leaf level only. Internal nodes store only routing keys, which increases the branching factor further and reduces tree height. More importantly, the leaf nodes are linked in a doubly linked list, making range scans extremely efficient: find the first matching leaf, then follow the chain. This is why virtually every major relational database Oracle, PostgreSQL, MySQL, SQL Server uses the B+ Tree as its primary index

structure (Comer, 1979). It handles mixed workloads well, which is the norm in OLTP systems.

4. HASH-BASED INDEXING TECHNIQUES

Hash indexes work by applying a hash function to the search key to compute directly the address of the bucket where the record is stored. When the hash function distributes keys evenly, exact-match queries run in $O(1)$ average time faster than any tree-based approach. The tradeoff is that hash functions destroy key ordering, so range queries are simply not supported (Ramakrishnan & Gehrke, 2003).

Static hashing is simple and fast, but inflexible if the data grows significantly, overflow chains build up and performance degrades. A detailed treatment of static hashing and its reorganization strategies can be found in Elmasri & Navathe (2010). Dynamic hashing (Litwin, 1980) solves this by splitting buckets incrementally as data is added, without requiring a full rebuild. Extendible hashing (Fagin et al., 1979) takes a similar approach using an in-memory directory of bucket pointers, doubling the directory only when a bucket overflows. PostgreSQL uses an extendible hash implementation for its hash index type. These dynamic variants are well suited for growing datasets where exact-match performance matters.

5. BITMAP INDEXING

A bitmap index represents each distinct value of an attribute as a bit vector: for a table of n rows, bit i is set to 1 if row i has that value, 0 otherwise. To answer a multi-condition query, the system simply applies bitwise AND/OR/NOT operations on the relevant bitmaps operations that modern CPUs execute extremely fast using SIMD instructions (O'Neil & Quass, 1997). Bitmap indexes work best for low-cardinality attributes things like gender, region, status, or product category in analytical (OLAP) environments. Oracle, Apache Hive, and Apache Druid all use them heavily. The downside is that they are storage-intensive for high-cardinality attributes and expensive to maintain under frequent writes, since each update may touch multiple bitmaps. In transactional (OLTP) systems, they are generally avoided for this reason.

6. MODERN INDEXING TECHNIQUES

6.1. INVERTED INDEXES

An inverted index maps each distinct term in a document collection to the list of documents containing it called a posting list. It is the core data structure behind full-text search engines like Elasticsearch and Apache Solr, built on top of Apache Lucene. Construction is parallelizable and can be done with MapReduce at very large scale. Compression techniques like delta-encoding and variable-byte encoding reduce storage significantly while keeping decompression fast (Manning et al., 2008). For text search at any scale, there is no practical alternative.

6.2. DISTRIBUTED INDEXING IN BIG DATA SYSTEMS

When data is spread across many nodes as in Hadoop, Spark, or cloud storage traditional indexes no longer apply. Distributed indexes must be partitioned, fault-tolerant, and concurrently accessible. Google Bigtable and Apache HBase use a distributed B+ Tree variant where the index is itself stored as a sorted table across nodes. A more lightweight approach is data skipping: columnar formats like Parquet and ORC embed per-block statistics (min/max, dictionaries, Bloom filters) that let query engines skip large chunks of data without reading them at all a technique sometimes called predicate pushdown (Chang et al., 2008; Melnik et al., 2010).

6.3. INDEXING IN NOSQL DATABASES

NoSQL systems each handle indexing differently, shaped by their data model. MongoDB supports secondary indexes on any JSON field, including arrays and geospatial data. Neo4j uses index-free adjacency nodes directly reference their neighbors in storage, so graph traversals require no index lookups. Cassandra relies on consistent hashing for partition-level access and offers secondary indexes via materialized views. A recurring challenge across all these systems is maintaining consistency between primary data and secondary indexes without ACID transactions many NoSQL databases accept eventual consistency on index reads, which developers must account for (Sadalage & Fowler, 2012).

7. COMPARATIVE ANALYSIS

Table 1 summarizes the main characteristics of each indexing technique across four dimensions: query performance, storage overhead, scalability, and the workloads they are best suited for.

Table 1. Comparative analysis of database indexing techniques

Technique	Query Perf.	Storage OH	Scalability	Best Workload
B-Tree	$O(\log n)$	Medium	Medium	Range queries, general OLTP
B+ Tree	$O(\log n)$	Medium	High	Relational DBs, file systems
Static Hashing	$O(1)$ avg	Low	Low	Exact-match lookups (static data)
Dynamic Hashing	$O(1)$ avg	Medium	Medium	Growing datasets, exact-match
Extendible Hash	$O(1)$ avg	Medium	High	Dynamic workloads, key-value
Bitmap Index	Very High (AND/OR)	High	Medium	Data warehouses, low-cardinality
Inverted Index	Very High	High	Very High	Full-text search, IR systems
Distributed Idx	High	Very High	Very High	Big data, distributed analytics

NoSQL Index	High	Variable	Very High	Document, graph, key-value stores
-------------	------	----------	-----------	-----------------------------------

The B+ Tree stands out for its versatility it handles both point and range queries reasonably well, scales within a single node, and fits the typical OLTP workload. Hash-based indexes win on exact-match speed but cannot do range queries at all. Bitmap indexes have a very specific niche: multi-predicate analytics on low-cardinality columns. For text search and large-scale distributed systems, inverted and distributed indexes are the only viable options. The key practical insight is that none of these structures dominates across all dimension's workload profiling is essential before making a choice.

8. CURRENT TRENDS AND FUTURE RESEARCH

The most notable recent development in indexing is the learned index, introduced by Kraska et al. (2018). The idea is to replace a B-Tree with a neural network that predicts the position of a key in a sorted array. On static read-heavy datasets, learned indexes can be 1.5–3× faster and up to 100× smaller than B-Trees. Follow-up work, ALEX (Ding et al., 2020) and PGM-Index (Ferragina & Vinciguerra, 2020) extended this to support insertions and deletions, making learned indexes more practical for real workloads.

Cloud-native databases like Amazon Aurora and Google Spanner use distributed B+ Tree variants stored in disaggregated storage layers, allowing indexes to scale independently of compute. LSM-tree-based engines (RocksDB, Cassandra) use Bloom filters as probabilistic pre-filters to avoid unnecessary disk reads for absent keys. On the query optimization side, systems like Bao (Marcus et al., 2021) use reinforcement learning to guide join order selection based on past execution feedback. Commercial tools (SQL Server's Tuning Advisor, IBM Db2's Design Advisor) already automate index recommendations. Looking ahead, the most promising directions are online self-adapting indexes and multi-dimensional learned structures for geospatial and scientific data (Idreos et al., 2019).

9. CONCLUSIONS

This review covered the major indexing techniques used in database systems from classical tree-based and hash-based structures to bitmap indexes, inverted indexes, and modern distributed approaches. Looking at them together, a few clear patterns emerge that are worth summarizing.

The B+ Tree remains the most widely deployed index structure in relational systems, and its dominance is not accidental. It handles both point queries and range queries well, integrates naturally with buffer pool management, and has decades of engineering refinement behind it. Hash-based indexes win on raw speed for equality lookups, but the moment range queries enter the picture they become useless the hash function destroys key ordering and nothing can recover it. Bitmap indexes are powerful in data warehouse settings with low-cardinality attributes and complex multi-predicate filters, but they are a maintenance liability in write-intensive

transactional systems. Inverted and distributed indexes are now essential infrastructure for text search and big data query engines respectively not optional extras.

The most interesting current direction is the convergence of machine learning and index structures. Learned indexes, adaptive query optimizers, and AI-driven index recommendation tools all point toward the same shift: moving from statically designed structures toward systems that observe their workload and adjust accordingly. Early results are promising, but open problems around update handling, distribution shifts, and multi-dimensional data mean this is still an active research area rather than a solved one.

For practitioners, the conclusion is straightforward: there is no universal index. Every structure reviewed here represents a set of trade-offs between query speed and update cost, between storage efficiency and flexibility, between consistency and scalability. Understanding your query patterns and data distribution is the prerequisite for any meaningful indexing decision and getting that decision right is one of the highest-leverage choices in database design.

10. REFERENCES

- [1] Bayer, R., & McCreight, E. (1972). Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3), 173–189.
- [2] Chang, F., Dean, J., Ghemawat, S., et al. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), Article 4.
- [3] Comer, D. (1979). Ubiquitous B-tree. *ACM Computing Surveys*, 11(2), 121–137.
- [4] Date, C. J. (2003). *An Introduction to Database Systems* (8th ed.). Addison-Wesley.
- [5] Ding, J., Minhas, U. F., Yu, J., et al. (2020). ALEX: An updatable adaptive learned index. *Proc. ACM SIGMOD*, 969–984.
- [6] Elmasri, R., & Navathe, S. B. (2010). *Fundamentals of Database Systems* (6th ed.). Addison-Wesley.
- [7] Fagin, R., Nievergelt, J., Pippenger, N., & Strong, H. R. (1979). Extendible hashing. *ACM TODS*, 4(3), 315–344.
- [8] Ferragina, P., & Vinciguerra, G. (2020). The PGM-index: A fully-dynamic compressed learned index. *PVLDB*, 13(8), 1162–1175.
- [9] Graefe, G. (2011). Modern B-tree techniques. *Foundations and Trends in Databases*, 3(4), 203–402.
- [10] Hellerstein, J. M., Stonebraker, M., & Hamilton, J. (2007). Architecture of a database system. *Foundations and Trends in Databases*, 1(2), 141–259.
- [11] Idreos, S., Dayan, N., Qin, W., et al. (2019). Learning data structure alchemy. *IEEE TCDE Bulletin*, 42(2), 4–14.
- [12] Knuth, D. E. (1998). *The Art of Computer Programming*, Vol. 3 (2nd ed.). Addison-Wesley.
- [13] Kraska, T., Beutel, A., Chi, E. H., Dean, J., & Polyzotis, N. (2018). The case for learned index structures. *Proc. ACM SIGMOD*, 489–504.
- [14] Litwin, W. (1980). Linear hashing: A new tool for file and table addressing. *Proc. 6th VLDB*, 212–223.
- [15] Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
- [16] Marcus, R., Negi, P., Mao, H., et al. (2021). Bao: Making learned query optimization practical. *Proc. ACM SIGMOD*, 1275–1288.

- [17] Melnik, S., Gubarev, A., Long, J. J., et al. (2010). Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1–2), 330–339.
- [18] O'Neil, P., & Quass, D. (1997). Improved query performance with variant indexes. *Proc. ACM SIGMOD*, 38–49.
- [19] O'Neil, P., Cheng, E., Gawlick, D., & O'Neil, E. (1996). The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4), 351–385.
- [20] Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3rd ed.). McGraw-Hill.
- [21] Sadalage, P. J., & Fowler, M. (2012). *NoSQL Distilled*. Addison-Wesley.