

15 Data Structures that Power Distributed Databases

A Learning Guide

October 26, 2025

Abstract

Distributed databases are the backbone of modern large-scale applications. This document provides a comprehensive overview of 15 key data structures that enable their fast lookups, efficient storage, and high-throughput operations, from Hash Indexes and Bloom Filters to CRDTs and Vector Clocks.

Introduction

Distributed databases are the backbone of modern large-scale applications, powering everything from real-time analytics to global e-commerce platforms. Behind the scenes, these systems rely on specialized data structures to enable fast lookups, efficient storage, and high-throughput operations, even when managing terabytes of data. Here are the 15 key data structures that make distributed databases work.

1 Hash Indexes

A **hash index** efficiently maps keys to values using a hash function that converts a key into an integer, which serves as an index in a hash table. This indexing technique is optimized for fast lookups and insertions, making it ideal for operations like finding a record with a specific ID. Hash indexes provide **$O(1)$ average-time complexity** for insertions, deletions, and lookups. They are commonly used in key-value stores like DynamoDB and caching systems like Redis where quick access to data is crucial.

2 Bloom Filters

A **Bloom filter** is a space-efficient, probabilistic data structure used to test set membership. It starts as a bit array initialized with zeros and relies on multiple independent hash functions. When an element is added, it is passed through k hash functions, each mapping it to an index in the bit array, and those bits are set to 1. To check if an element exists, if all corresponding bits are 1, the element is **probably** in the set (though false positives can occur); if any bit is 0, the element is **definitely not** in the set. Bloom filters allow databases to efficiently check whether a key might exist in a dataset, helping to avoid unnecessary disk lookups. They are widely used in systems like SSTables in LSM trees (Apache Cassandra) and database partitions for fast key lookups.

3 LSM Trees (Log-Structured Merge Trees)

A **Log-Structured Merge Tree** is a write-optimized data structure designed to handle high-throughput workloads efficiently. Unlike B-Trees which modify disk pages directly, LSM Trees

buffer writes sequentially in memory and periodically flush them to disk, reducing random I/O operations. New writes are first stored in an in-memory structure called a **MemTable** (typically a Red-Black Tree or Skip List). Once the MemTable reaches a certain size, it is flushed to disk as an immutable **SSTable (Sorted String Table)**. Reads first check the MemTable, then search recent SSTables using Bloom Filters to quickly determine whether a key exists. Over time, multiple SSTables are merged through **compaction** to remove duplicate, obsolete, or deleted records. LSM Trees are widely used in high-scale NoSQL databases like Apache Cassandra, Google Bigtable, and RocksDB.

4 B-Trees and B+Trees

B-trees are self-balancing tree data structures that maintain sorted data and allow searches, sequential access, insertions, and deletions in $O(\log n)$ time. They are perfectly balanced, meaning every leaf node is at the same depth, and every inner node other than the root is at least half full. **B+trees** are a specialized variant where key-value pairs are stored only at leaf nodes, while non-leaf nodes store only keys and child pointers. All nodes also contain "next" and "previous" pointers, allowing each level to act as a doubly-linked list. Since inner nodes don't store values, more keys can fit per inner node, keeping the tree shallower. B-trees and B+trees are the foundation of most relational database indexes, including MySQL, PostgreSQL, and Oracle.

5 Skip Lists

A **skip list** is a probabilistic data structure that arranges elements in a hierarchical linked list and uses randomized "skips" to enable fast search over ordered data. It consists of multiple levels where each level contains a linked list of nodes. The bottom level contains all data items in sorted order, while higher levels contain a fraction (typically $p=1/2$) of the nodes from the level below. Each node has forward pointers at each level pointing to its successor. Search operations traverse the highest possible layer and drop down only as elements are encountered, allowing $O(\log n)$ time complexity on average. Skip lists provide fast search over dynamic ordered data without complex rebalancing operations. They are used in databases like Redis, distributed systems, and as the underlying structure for MemTables in LSM tree implementations.

6 Merkle Trees

A **Merkle Tree** is a binary tree of cryptographic hashes constructed from the bottom up to efficiently verify data integrity. Leaf nodes store hashes of data blocks (e.g., transactions), while non-leaf nodes store hashes created by combining the hashes of their two children. This process continues until the root, called the **Merkle root**, which serves as a cryptographic fingerprint of all the data. By comparing Merkle roots, distributed systems can quickly verify that two massive datasets are identical without sending gigabytes of data across a network. Merkle trees allow comparison and verification of transactions with viable computational power and bandwidth. They are extensively used in blockchain networks (Bitcoin, Ethereum), distributed databases for synchronizing replicas efficiently (Cassandra, DynamoDB), version control systems (Git), and distributed file systems.

7 Inverted Indexes

An **inverted index** is a data structure that maps each unique term to the documents where it appears, enabling very fast full-text searches. Rather than storing text word by word, it captures unique terms and their frequency across documents. During indexing, text fields are processed by analyzers that break text into tokens, normalize them (e.g., lowercasing), and filter out noise like stopwords. For example, "The quick brown fox" might be tokenized into ["quick", "brown", "fox"], with each term linked to document IDs. When a user searches, the query is tokenized using the same analysis steps, then matched against the inverted index. Results are scored using algorithms like BM25 which consider term frequency and document length to rank relevance. Inverted indexes are the core of search engines like Elasticsearch, Apache Solr, and Lucene.

8 Tries (Prefix Trees)

A **trie** (pronounced "try") is a tree-based data structure that stores strings efficiently by sharing common prefixes. Each node represents a single character, and the path from the root to any node forms a prefix of one or more strings. Words that share prefixes use shared paths, making the structure space-efficient. A special flag marks the end of complete words. Tries enable fast string search, insertion, and deletion operations in $O(L)$ time, where L is the string length. They are particularly effective for tasks such as autocomplete, spell checking, and IP routing. Prefix searching in a trie is extremely efficient as you only need to traverse until the end of the prefix, not the entire word. Tries are used in databases for prefix-based searches, autocomplete features, and can be stored in document stores or key-value databases.

9 Consistent Hashing

Consistent hashing is a distributed hashing technique that operates independently of the number of servers by assigning them positions on an abstract circle or hash ring. Both data objects and server nodes are mapped to positions on this ring using hash functions. Each object key belongs to the server whose key is closest in a clockwise direction on the ring. The key advantage is that when servers are added or removed, only a small fraction of keys need to be remapped, minimizing disruption. To ensure even distribution, servers are assigned multiple virtual nodes (labels) on the ring. Consistent hashing solves the scalability problems of traditional hashing where adding or removing a server requires rehashing most keys. It is widely used in distributed caching systems (Memcached), distributed databases (Cassandra, DynamoDB, Riak), content delivery networks, and load balancing systems.

10 HyperLogLog

HyperLogLog is a probabilistic algorithm designed to estimate the cardinality (number of distinct elements) of very large datasets with high accuracy and low memory usage. The algorithm uses hash functions to map each element to a binary string and focuses on the leading zeros in the hash value. It divides the hash value range into multiple registers (buckets), with each register tracking the maximum number of leading zeros observed for items hashed to that register. After processing all items, HyperLogLog uses a harmonic mean to combine information from all registers to compute the cardinality estimate. Using an auxiliary memory of m units, HyperLogLog produces an estimate with a typical accuracy (standard error) of about $1.04/\sqrt{m}$. For example, with 1.5 kilobytes of memory, it can estimate cardinalities beyond 10^9 with 2% accuracy. HyperLogLog is used in databases like Redis, Spark, Presto, and analytics platforms like

Google Analytics and Adobe Analytics for counting unique visitors, tracking distinct events, and database query optimization.

11 Count-Min Sketch

Count-Min Sketch is a probabilistic data structure that estimates the frequency of elements in a data stream. It uses a 2D array of counters with multiple hash functions, where each row corresponds to a different hash function. When an element appears in the stream, it is passed through all hash functions, and the counters at the corresponding positions in each row are incremented. To estimate the frequency of an element, the same hash functions are applied, and the **minimum value** among the retrieved counts is returned as the estimate. This structure uses sub-linear space at the expense of over-counting some events due to hash collisions. Count-Min Sketch is useful for higher frequency counts, while very low counts should be treated as noise. It's particularly effective for applications like counting sales volume for popular products, tracking item popularity in recommendation systems, network traffic analysis, and detecting anomalies. The data structure is implemented in Redis and used in various big data streaming applications.

12 SSTables (Sorted String Tables)

A **Sorted String Table** is an immutable file format containing sorted key-value pairs. SSTables are the fundamental on-disk representation used by LSM tree-based databases. Data is organized into compressed data blocks, with index blocks providing mappings between key ranges and corresponding data blocks for efficient lookups. Filter blocks containing Bloom filters allow quickly determining if a key might exist in an SSTable file without reading the entire file. Each SSTable is written sequentially to disk and never modified—updates and deletes are handled by writing new SSTables. Over time, multiple SSTables are merged during compaction to consolidate data and remove obsolete entries. Three core operations (seek, next, previous) are used to iterate through data in SSTable files. The sorted nature enables efficient binary search for key lookups and supports fast range scans. SSTables are used in Apache Cassandra, ScyllaDB, Google Bigtable, RocksDB, and LevelDB.

13 R-Trees

An **R-tree** is a balanced tree data structure designed for spatial indexing and range queries over multi-dimensional data. It stores the minimum bounding rectangle (MBR) of each geometry in leaf nodes along with the row ID, while internal nodes store the bounding rectangle enclosing all child nodes. When querying for spatial relationships (e.g., "find all objects within this region"), the tree is traversed from top to bottom, skipping entire sub-trees if their bounding rectangles don't intersect the query region. This dramatically accelerates spatial queries that would otherwise require full table scans. R-trees are commonly used for storing and querying geospatial data such as restaurant locations, street maps, building polygons, and points of interest. They excel at queries like "find all items within 2 km of this location" or "find the nearest gas station". R-trees are implemented in spatial databases like PostGIS (PostgreSQL), MySQL spatial extensions, MongoDB geospatial indexes, and SQLite R*Tree module.

14 CRDTs (Conflict-Free Replicated Data Types)

Conflict-Free Replicated Data Types are data structures replicated across multiple computers in a network that allow independent, concurrent updates without coordination. An algorithm automatically resolves any inconsistencies, and although replicas may have different states at any point, they are guaranteed to eventually converge. There are two main types: **operation-based CRDTs** where updates are represented as commutative and idempotent operations that can be applied in any order, and **state-based CRDTs** where each replica maintains its state independently and periodically exchanges states using merge functions. CRDTs support decentralized operation without requiring a single server, making them ideal for peer-to-peer networks. Common CRDT types include counters, sets, lists, registers, and graphs. They are used in distributed databases (Redis, Riak, Cosmos DB), collaborative editing systems, online chat systems, mobile applications requiring offline capabilities, and distributed caching.

15 Vector Clocks

Vector clocks are a mechanism used in distributed systems to track the causality and ordering of events across multiple nodes without a global clock. Each process maintains a vector (array) of logical clocks, with one counter per process in the system. Initially all counters are zero. When an internal event occurs, the process increments its own counter. When sending a message, the process increments its counter and includes a copy of its vector with the message. When receiving a message, the process increments its counter and updates each element in its vector by taking the maximum of its own value and the received value. By comparing vector clocks, systems can determine if one event causally happened before another, after it, or concurrently. This enables effective conflict resolution and consistency maintenance. Vector clocks are used in distributed databases like Cassandra and DynamoDB to resolve conflicts when multiple replicas are updated independently, collaborative editing applications, distributed version control systems, and event-driven monitoring systems.

Summary

These 15 data structures form the foundation of modern distributed databases, each solving specific challenges in data storage, retrieval, consistency, and scalability. From hash indexes providing $O(1)$ lookups to LSM trees optimizing write-heavy workloads, from Bloom filters reducing disk access to Merkle trees enabling efficient data synchronization, these structures work together to power the large-scale, high-performance systems that drive today's applications. Understanding how these data structures function and when to apply them is essential for building robust, scalable distributed systems.