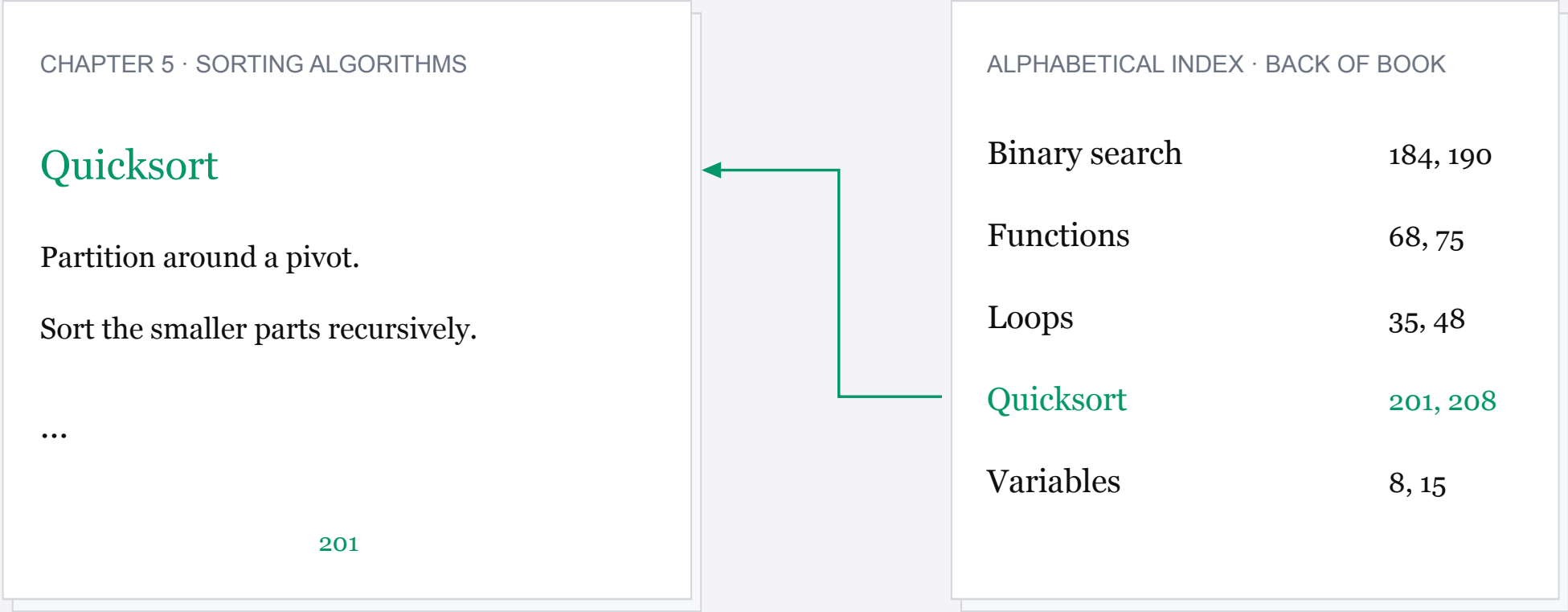


M3

Index Fundamentals

An index maps a search key, such as artist, to the data pages containing matching rows.

Index finds topics quickly

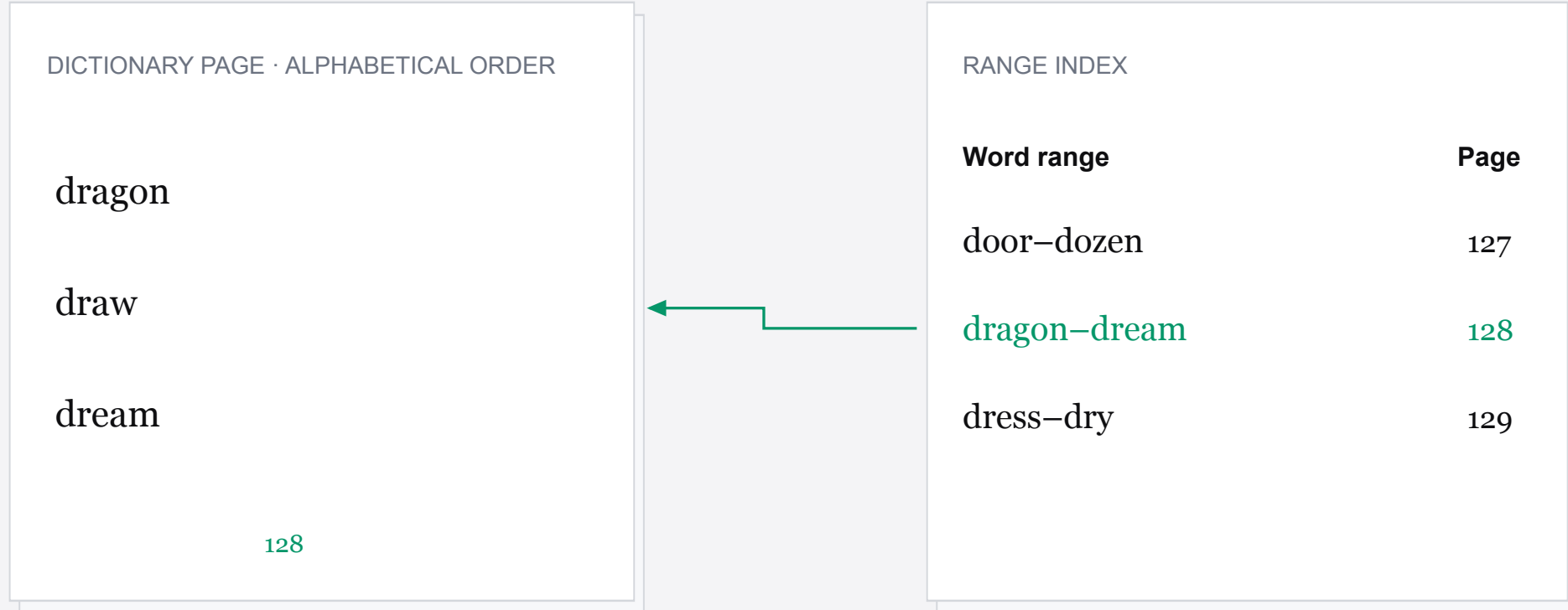


Look up “Quicksort” → turn to pages 201 and 208 in the book.

Python programming book.

A Python programming book organizes its chapters for teaching, not alphabetically by topic. Its back-of-book index sorts topics: “Quicksort” points to pages 201 and 208.

Sorted pages need only a range index



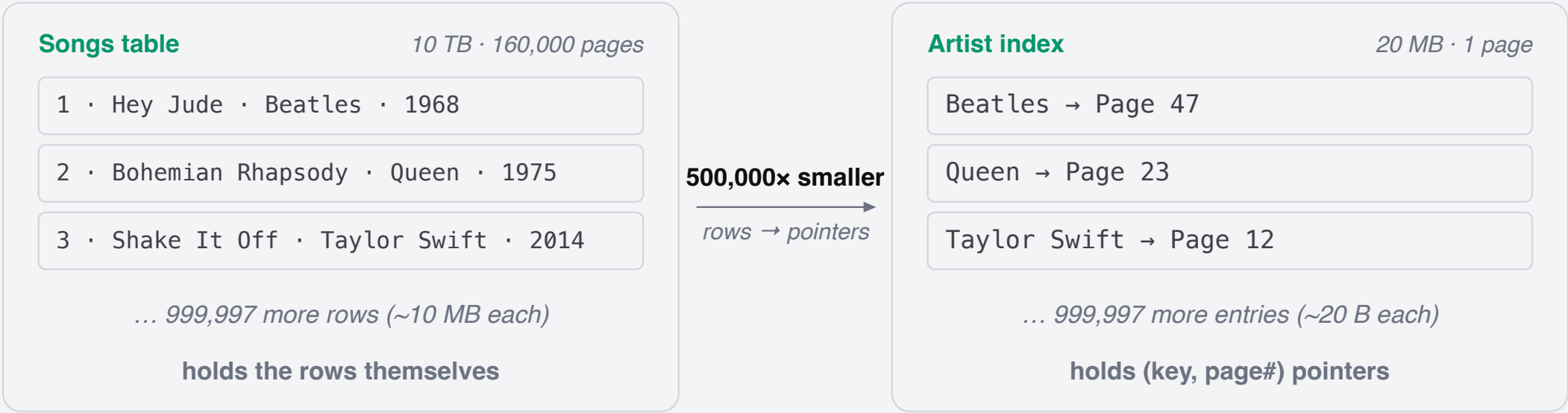
“draw” falls in dragon–dream → read page 128, then find the word.

Oxford English Dictionary analogy.

Think of the Oxford English Dictionary: its words are already alphabetical. An index can store each page’s word range rather than every word. To find “draw,” choose the dragon to dream range, then search page 128.

Index keys and pointers, not whole rows

rows on disk vs. (key, page#) pointers



relative size on disk (same horizontal scale)



10 TB

| 20 MB

(barely visible at this scale)

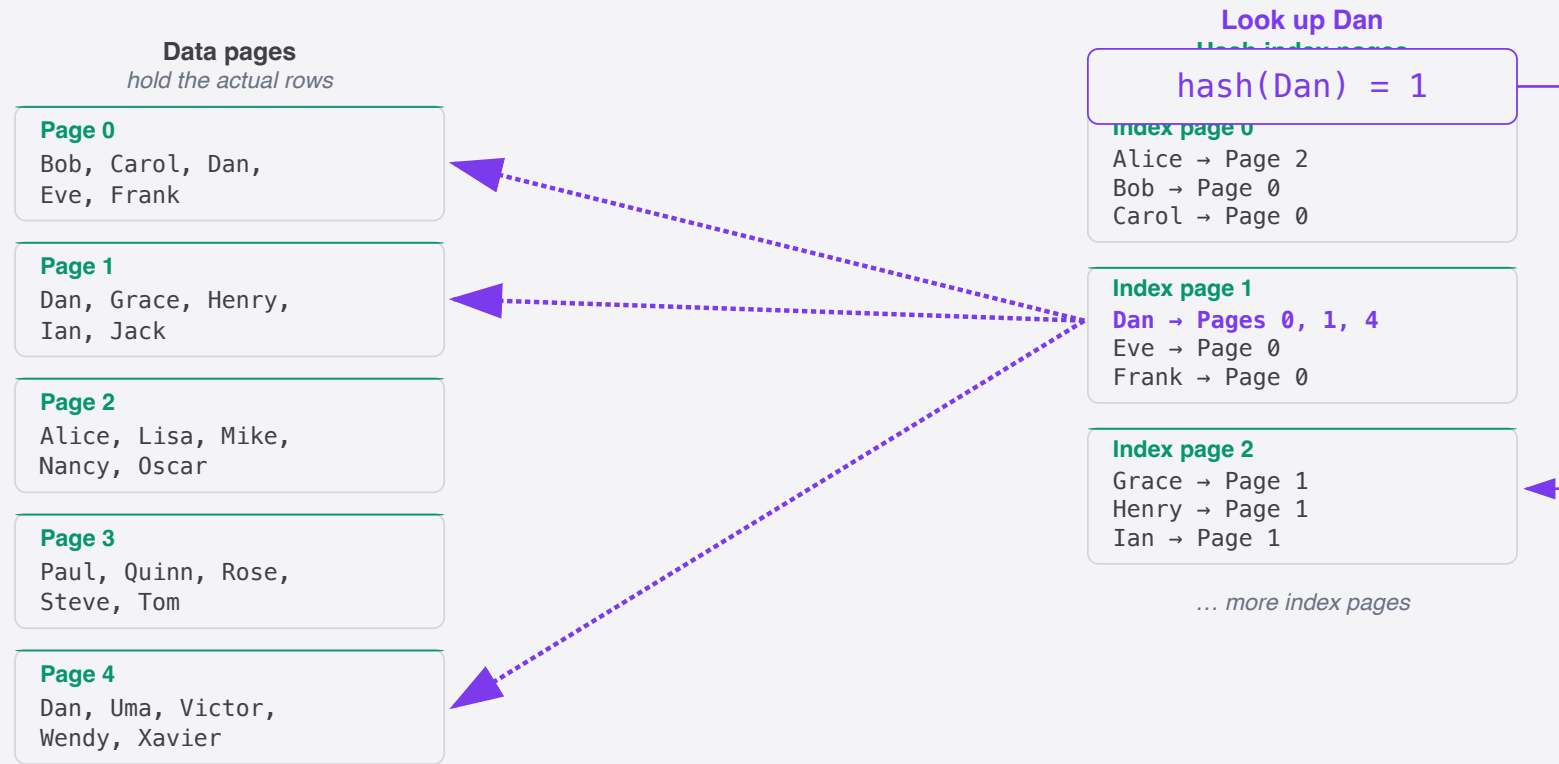
The Songs table stores 10 MB rows (10 TB, 160,000 pages). The artist index stores 20-byte pointers (20 MB, one page), 500,000 times smaller. WHERE artist = 'Beatles' without an index scans 160,000 pages. With an index, the database reads one index page, learns Beatles is on page 47, then reads page 47. The query uses two IOs instead of 160,000, 80,000 times fewer for the same answer.

M3

Hash Indexes

A hash index computes $\text{hash}(\text{key})$ and uses it to choose the page that stores the key's index entry.

hash(key) → index page → entry → data pages



A hash index on `People.name` stores `(search_key, page#)` entries in buckets chosen by `hash(value)`. Here, `hash(Dan) = 1` selects index page 1. Its `Dan` entry points to data pages 0, 1, and 4. The lookup requires 4 IOs: one index-page read and three data-page reads.

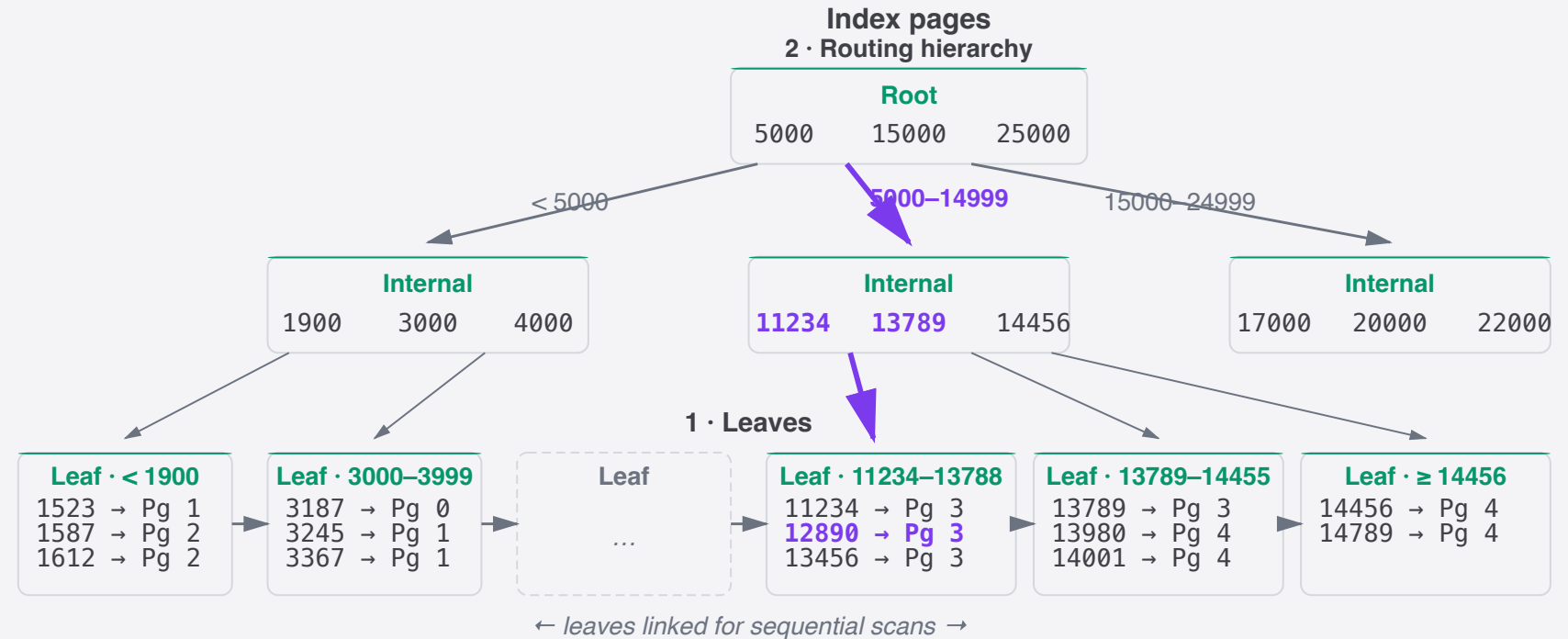
M3

B+Tree Indexes

A B+Tree index stores keys in sorted order in a balanced tree.

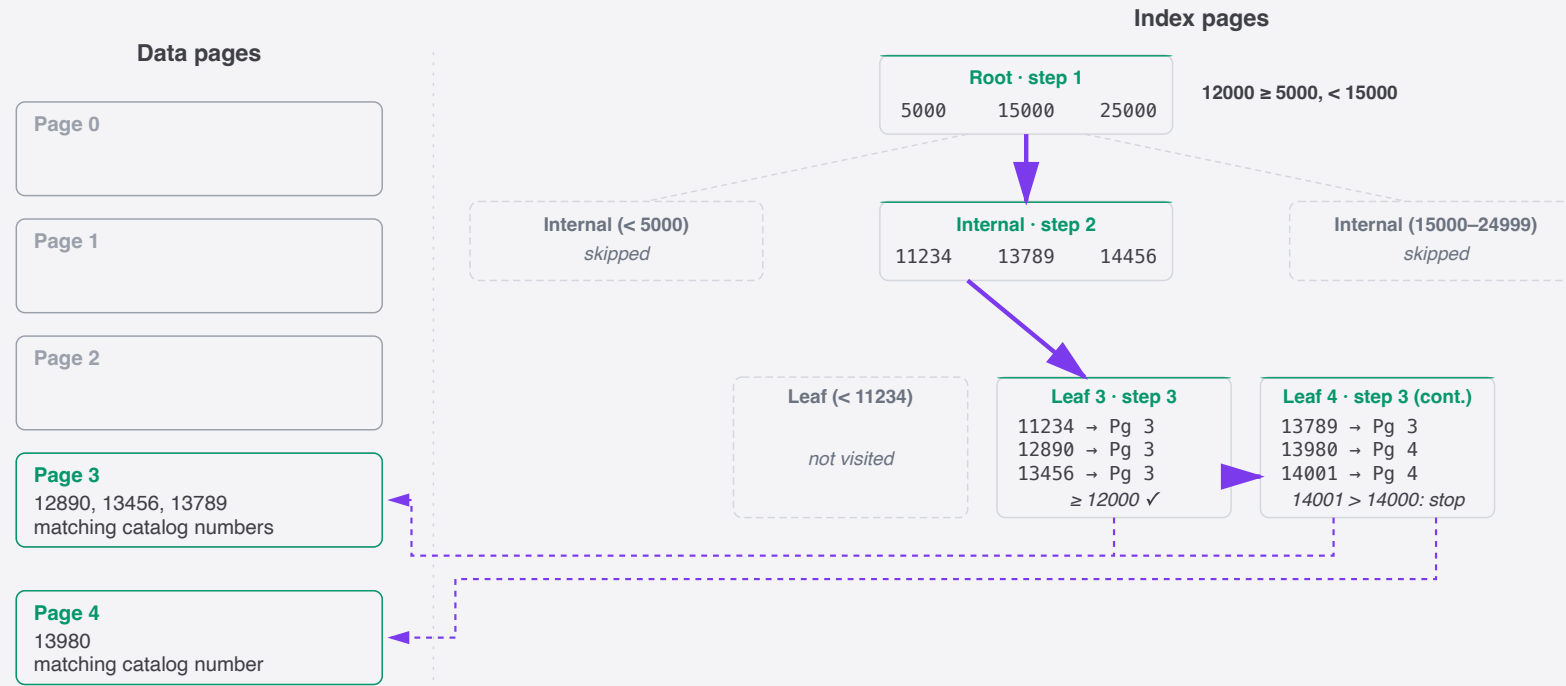
Build a hierarchy over sorted leaves

Data pages		
Song ID · Title · Catalog ID		
Page 0		
1	Evermore	4821
2	Willow	4953
3	Shape of You	3187
Page 1		
4	Photograph	3245
5	Shivers	3367
6	Yesterday	1523
Page 2		
7	Yellow Submarine	1587
8	Hey Jude	1612
9	Bad Blood	1899
Page 3		
10	Flowers	11234
11	Unholy	11567
12	As It Was	12890
13	Heat Waves	13456
Page 4		
15	Blinding Lights	13980
16	Good 4 U	14456
17	Drivers License	14789



Data pages on disk store the Songs rows. The B+Tree on disk stores a root and internal nodes that route lookups to sorted leaves. Leaves store (key, data-page pointer) pairs and link left-to-right. For catalog_no = 12890 ("As It Was"), the root selects 5000..14999, the internal node selects 11234..13788, the leaf resolves 12890 → Page 3, and Page 3 contains the row. The lookup reads four pages.

Find catalog numbers: $12,000 \leq \text{key} \leq 14,000$



For $12000 \leq \text{catalog_no} \leq 14000$, the query visits the solid green nodes, skips the dashed grey ones, and fetches only the data pages that contain matches (Page 3 and Page 4). The descent runs root → middle internal → Leaf 3, then follows leaf links. Leaf 3 yields 12890 and 13456. Leaf 4 yields 13789 and 13980. The next key in Leaf 4 is 14001, which exceeds 14000, so the scan stops without reading another leaf. The query reads 4 index pages and 2 data pages, 6 IOs total, versus roughly 160,000 for a full scan, about 27,000× fewer IOs.

High fanout keeps B+Trees shallow

Fanout is the number of children per internal page. Higher fanout means fewer levels.

$$f \approx \frac{\text{page size}}{\text{key size} + \text{pointer size}}, \quad h \approx \lceil \log_f N \rceil$$

N = number of indexable keys; **h** = approximate number of index levels.

B+Trees: few reads, more work on writes

For **1 trillion keys** and **8-byte pointers**, ignoring headers and unused space:

Page size	Key size	Approx. fanout	Index levels
64 KB	8 B	4,096	4
64 KB	1 KB	63	7
64 MB	8 B	4.2 million	2
64 MB	1 KB	65,027	3

- **Reads:** High fanout keeps the tree shallow. Point queries follow one path; range queries continue through sorted leaves.
- **Writes:** Inserts update pages. Full pages split, and splits can propagate upward. Maintaining balance adds work under heavy writes.

Next: Can we buffer writes and create sorted files in batches instead?

M3

SSTables: Sorted String Tables

An SSTable is an immutable file of key-value pairs sorted by key.

An SSTable is a sorted, immutable file

one immutable file · many files coexist on disk

1 · one SSTable file, three sections

Bloom filter <i>held in RAM</i> fast not-exists check	Index block <i>on disk</i> sparse (every Nth key) key ₁₀₀ → 4096 key ₂₀₀ → 8192 key ₃₀₀ → 12288	Data blocks · sorted by key <i>on disk · compressed · read by offset</i> key ₁₀₀ · v1 · {...} key ₁₀₁ · v1 · {...} key ₁₅₀ · v1 · {...} key ₂₀₀ · v1 · {...}
--	--	--

Analogy: Oxford English Dictionary

Bloom filter <i>held in RAM</i> Does “draw” exist? Does “diagon alley” exist?	Index block <i>first word → page</i> door → Page 201 dragon → Page 202 dress → Page 203 One entry per page. For “draw,” choose the “dragon” entry.	Data pages · sorted strings <i>word → definition</i> Page 201 door an entrance dove a bird dozen twelve Page 202 dragon a mythical creature draw make a picture dream images during sleep Page 203 dress a garment drip fall in drops dry not wet
---	--	---

Each SSTable has its own sparse index and sorted data; multiple files can coexist.

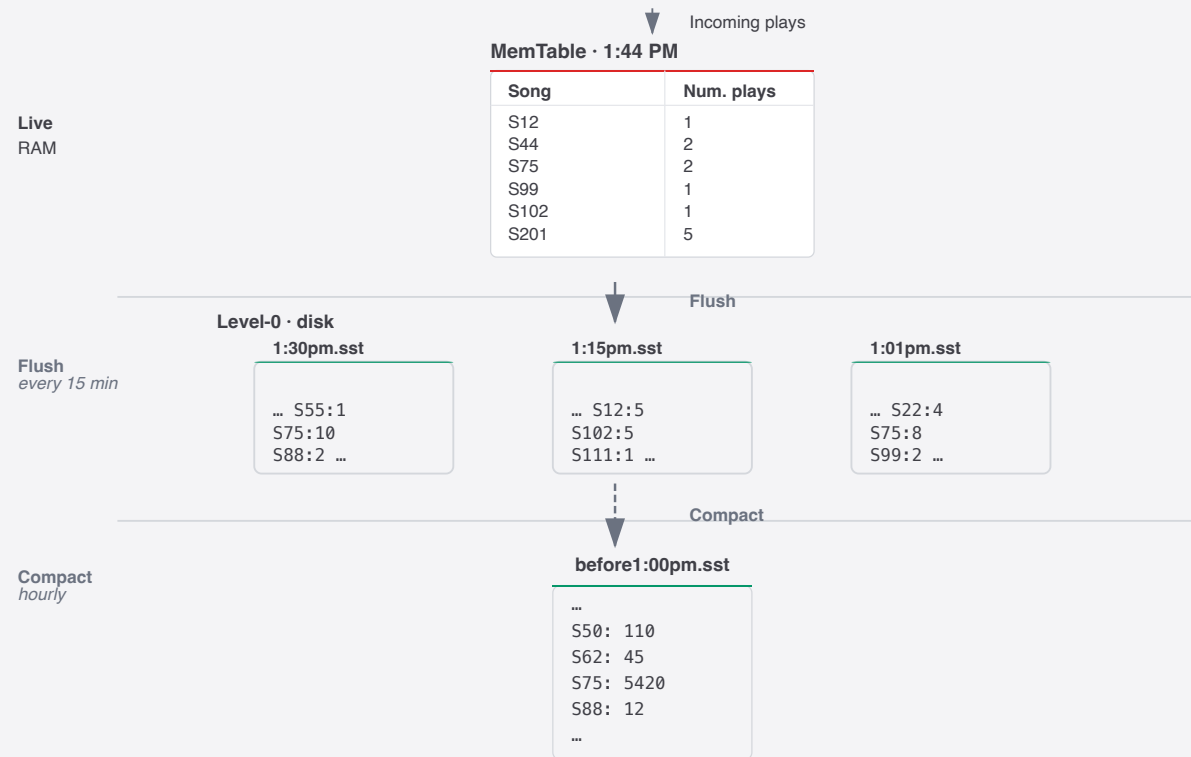
An SSTable is one immutable on-disk file with three sections: a Bloom filter cached in RAM for fast not-exists checks, a sparse index block that binary-searches to the target data block, and sorted, compressed data blocks. Think of an Oxford English Dictionary: the sparse index records the first word on each page, while the pages contain sorted words and their definitions. To find draw, follow the dragon → Page 202 entry. These illustrative page numbers stand for data-block offsets in an SSTable. Multiple SSTables can coexist, each with its own index and sorted data.

M3

LSM Tree Indexes

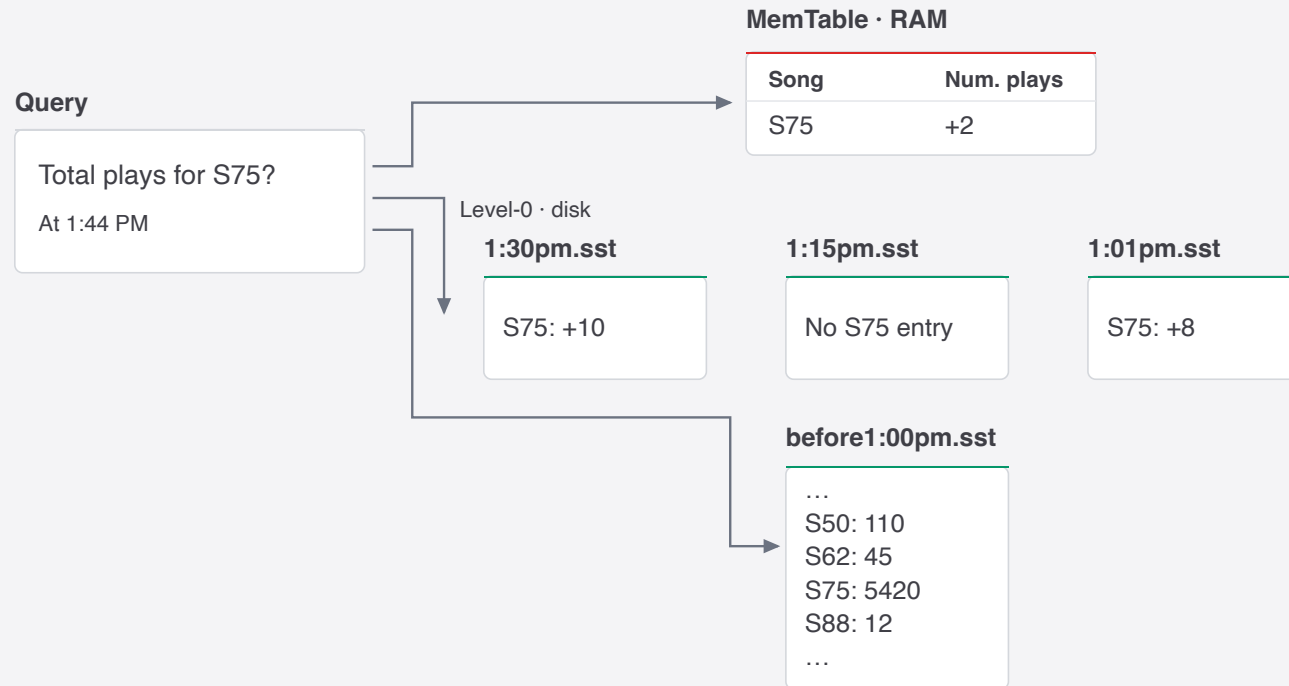
An LSM tree buffers inserts in a RAM MemTable, flushes them to immutable SSTables, and merges SSTables in the background.

Buffer writes, flush files, then compact



LSM write path at 1:44 PM, RAM on top and disk below, time flowing downward. Incoming plays land in the in-RAM MemTable (fast, no seek, lost on crash unless replayed from the WAL). Every 15 minutes the engine flushes it to an immutable Level-0 SSTable on disk (1:01pm.sst, 1:15pm.sst, 1:30pm.sst). Hourly compaction merges those into one larger sorted, de-duplicated Level-1+ file. Total plays for S75 spread across tiers (+2 in the MemTable, +10, no matching entry, and +8 in Level-0, 5,420 in compacted history) and the read path sums them to 5,440 (merge-on-read). The merge operator is configurable: "add" for counters, "newest wins" for last-write-wins keys, "set union" for tags, "max" for high-water-marks. The merge section walks through more examples.

Combine recent writes with stored history



$$2 + 10 + 0 + 8 + 5420 = 5,440 \text{ plays}$$

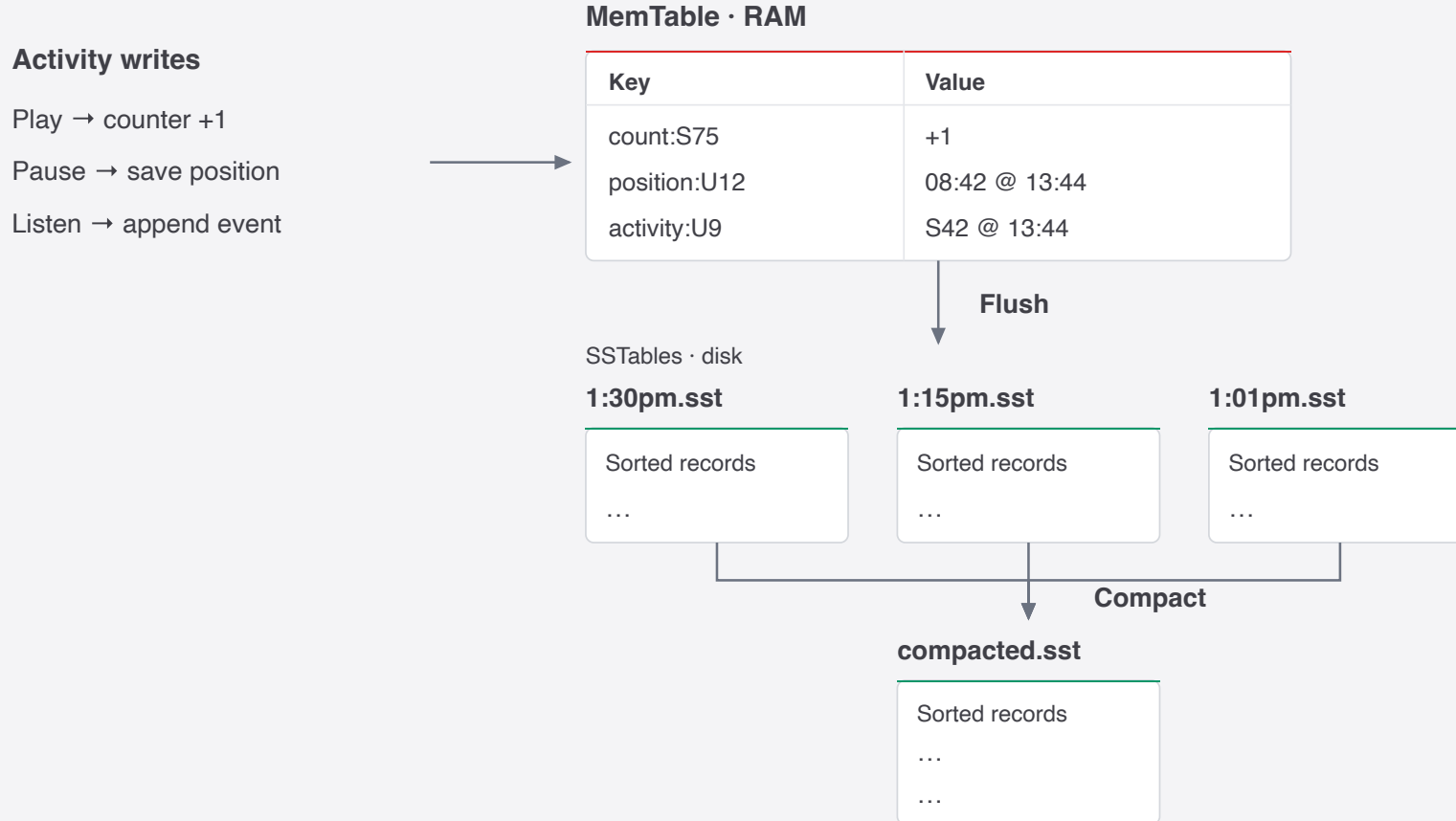
Merge-on-read computes S75's play count at read time instead of storing it, by summing every tier that holds a fragment. The live in-RAM MemTable (red) adds +2, the three flushed Level-0 SSTables on disk (green) add $10 + 0 + 8 = 18$, and the compacted Level-1+ SSTable on disk (green) adds +5,420, for a total of $2 + 18 + 5,420 = 5,440$ (the merge function is "add" because plays are a counter). Each SSTable carries a bloom filter, so the reader skips any tier that cannot hold the key with no disk read, making most reads far cheaper than this worst case.

M3

Case Study 3.3: How LSM trees handle Spotify-scale activity

This example uses an LSM tree to batch writes for play counters, playback positions, and friend activity.

Three workloads share the same write path



In this illustrative architecture, a play writes a counter increment, a pause writes a timestamped position, and a listen appends an activity event. The shared write path buffers records in RAM, flushes sorted SSTables, and compacts them using the appropriate merge rule. The key prefixes distinguish the three example workloads.

Three LSM use cases in Spotify

Use case	What gets written	What the read does
1. Counters	Each play adds +1	Sum the counts
2. Latest position wins	Playback position and timestamp	Return the newest position
3. Friend activity	Timestamped listening events	Merge events in time order

M3

Case Study 3.2: How an inverted index supports Spotify search

This example architecture supports text search across 100 million tracks with an inverted index, which maps each search term to the song_ids that contain it.

Map each word to its matching songs

Find songs containing “Crazy” AND “Heart”

Term index pages

Crazy
Dance
Dream

Heart
Hello
Love
Music
Yesterday

Sorted terms; one row per term

Postings lists · song IDs

Crazy
12, 45, 108, 145, 201, ...

Heart
8, 12, 88, 108, 180, ...

Intersect the lists in RAM

Matching song IDs: 12, 108

Song data pages

Song 12
Title, artist, ...

Song 108
Title, artist, ...

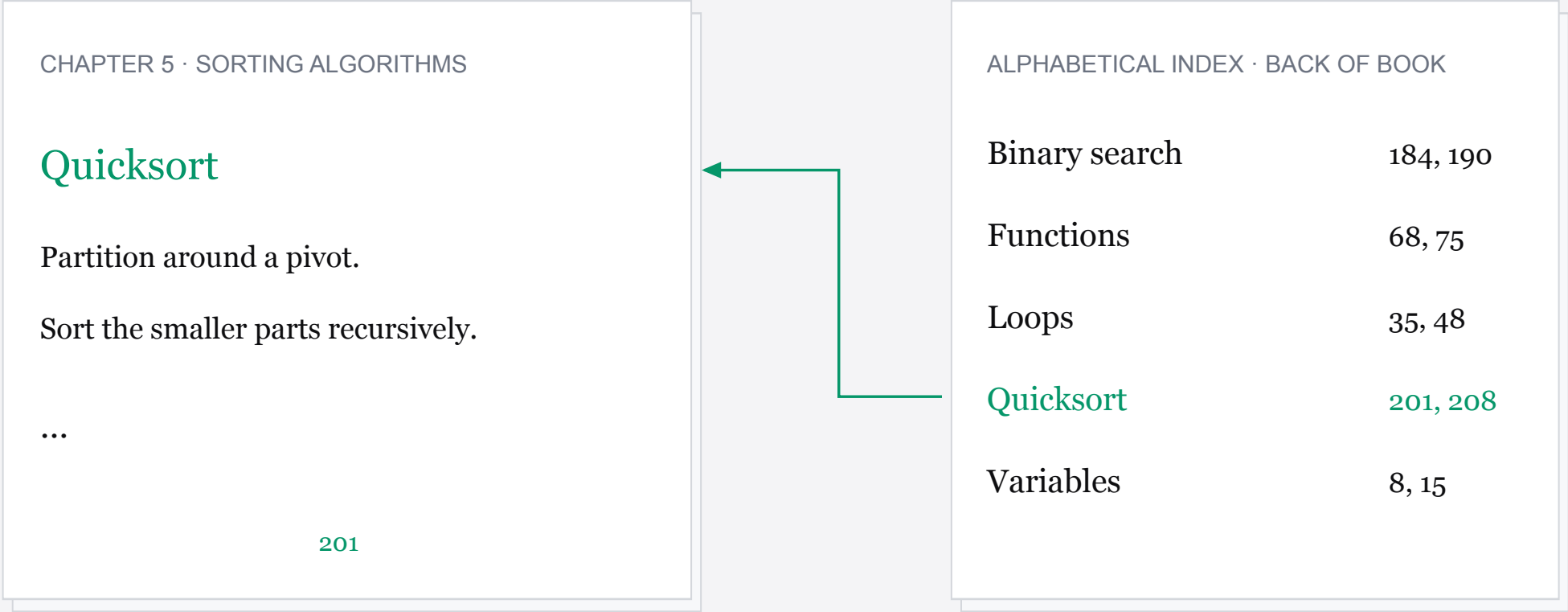
Each term-index page holds several sorted terms; each term points to a postings list of song IDs. For Crazy AND Heart, intersect the two lists in RAM to get {12, 108}, then fetch those songs from the data pages. The postings store IDs; the data pages store rows. The B+Tree routing nodes above the term pages are omitted.

M3

IO Cost & Trade-offs: Choosing Your Index

Each index has costs and benefits.

Index finds topics quickly

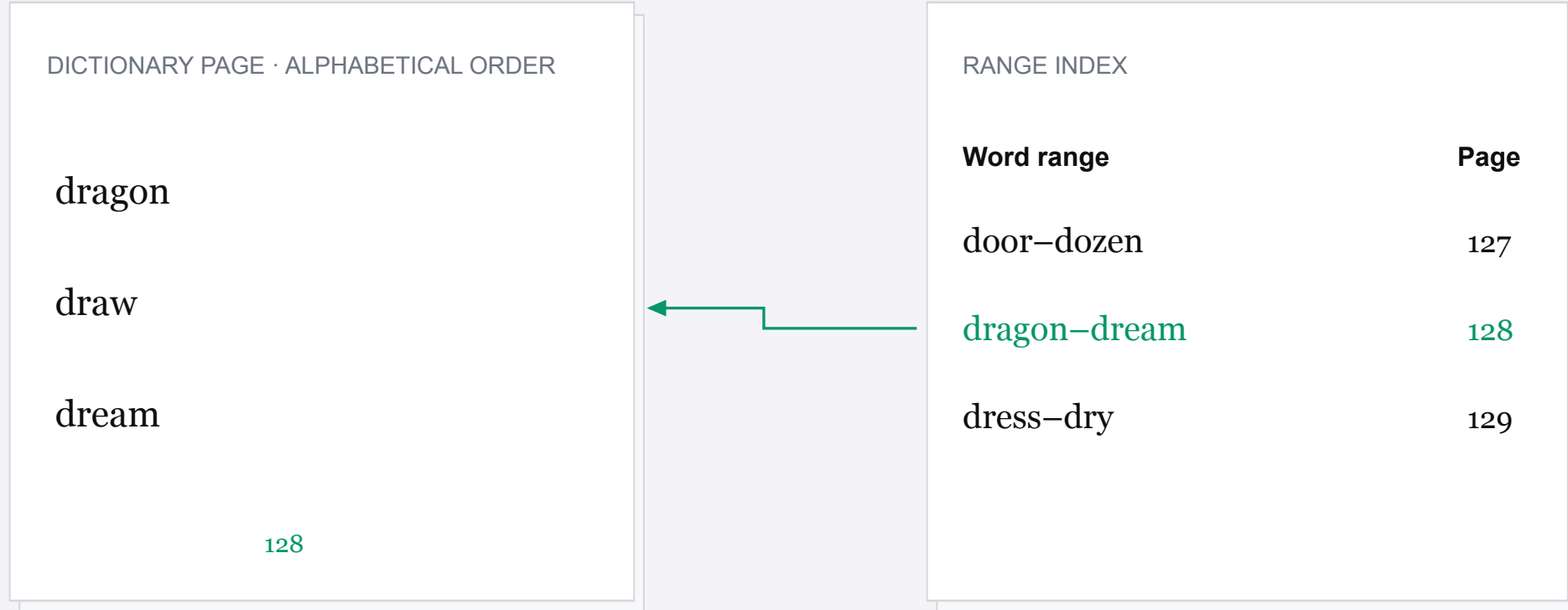


Look up “Quicksort” → turn to pages 201 and 208 in the book.

Python programming book.

A Python programming book organizes its chapters for teaching, not alphabetically by topic. Its back-of-book index sorts topics: “Quicksort” points to pages 201 and 208.

Sorted pages need only a range index



“draw” falls in dragon–dream → read page 128, then find the word.

Oxford English Dictionary analogy.

Think of the Oxford English Dictionary: its words are already alphabetical. An index can store each page’s word range rather than every word. To find “draw,” choose the dragon to dream range, then search page 128.

Clustering puts one user's listens together

Find all listens for user 7.

Unclustered · arrival order

Clustered · (user_id, listen_time)

Rows: user_id · song_id

Page 1	Page 2	Page 3	Page 4
7 · S75	3 · S44	7 · S12	5 · S55
2 · S12	7 · S88	4 · S99	7 · S44

4 data-page reads

Page 1	Page 2	Page 3	Page 4
2 · S12	4 · S99	7 · S75	7 · S12
3 · S44	5 · S55	7 · S88	7 · S44

2 data-page reads

Index lookup cost excluded.

Same matches. Fewer pages.

Find all listens for user_id = 7. Both layouts contain the same eight rows, two per page; the figure shows user_id and song_id. In arrival order, user 7's four listens occupy four pages. Clustering by (user_id, listen_time) groups them onto two neighboring pages, in time order. That is four versus two data-page reads, excluding index lookup cost. The index finds the rows; clustering puts them on fewer pages.

M3

Case Study 3.4: How Spotify Wrapped serves a launch-day spike

Spotify Wrapped computes each user's yearly summary before launch and stores the result by user, so the launch-day spike serves precomputed records.

Precompute summaries, then look them up



The design clusters Listens by (user_id, listen_time), so a batch job reads each user's year sequentially and writes one summary row per user. On December 1, the card fetch uses one hash-index lookup on user_id.

M3

How Three Changes Cut One Query from \$550,000 to \$400

A company runs one weekly aggregation 100 times a day
against a 1 TB table.

Weekly trending genres: October 21-27

```
SELECT s.genre,  
       COUNT(*) AS listens,  
       COUNT(DISTINCT l.user_id) AS listeners,  
       AVG(l.rating) AS avg_rating  
FROM Listens AS l  
JOIN Songs AS s ON s.song_id = l.song_id  
WHERE l.listen_time >= '2026-10-21'  
      AND l.listen_time < '2026-10-28'  
GROUP BY s.genre  
ORDER BY listens DESC;
```

Four problems: reduce the cost of the same query

Songs: **400 pages**. Listens: **16,000 pages**. Buffer: **20 pages**. This week's listens: **150 pages**.

▼ Problem 1 • Baseline join

What does a Hash Partition Join cost? Would a Block Nested Loop Join be cheaper?

▼ Problem 2 • Skip irrelevant dates

Only 150 Listens pages match the week. Which index would help, and which pages must we still read?

▼ Problem 3 • Skip unused columns

The query uses only a few columns. How could we avoid reading the rest of each row?

▼ Problem 4 • Store fewer bytes

Genres and ratings repeat. Which lossless encodings could shrink these columns?

Read fewer pages to cut query cost



Fewer rows × fewer columns × fewer bytes $\approx$ 1400× fewer IOs.

Four stages of the same weekly aggregation. The Hash Partition Join costs 49,200 IOs. A B+Tree index cuts the query to about 554 IOs, columnar storage cuts it to about 60, and compression cuts it to about 35. The endpoint ratio gives roughly 1,400× fewer IOs and lowers annual cost from about \$550,000 to about \$400.