READ MODE · ANSWER FIRST

How MySQL InnoDB B-tree indexes work

See how InnoDB clustered leaves, B-tree search, page splits, secondary lookups, covering indexes, and index tradeoffs work.

CHEAT SHEET · 5 ESSENTIAL IDEAS

The whole story in 5 lines

InnoDB indexes are ordered page trees that trade extra storage and write maintenance for short, reusable read paths.

1. A clustered-index leaf stores complete row records in primary-key order, so the primary tree is also the table.
2. Point and range searches share the same separator descent, but a range can continue across neighboring leaf pages.
3. A full leaf splits into sorted siblings and adds a parent separator, preserving order by doing extra write work.
4. A secondary leaf stores its indexed columns plus the primary key, which may trigger another clustered-tree search.
5. Indexes speed selective and ordered reads, but every added index consumes cache space and must be maintained by writes.

STAGE 1 OF 7

Setup

Our query is simple: find customer 63 among eight million rows. A full scan would inspect row after row, including millions that cannot possibly match.

InnoDB takes a shorter route through B-tree pages, each holding many sorted entries. The primary-key tree is called the clustered index because the table’s rows are organized around it.

Separator keys divide the possible IDs into ranges, so each comparison can discard whole pages. A secondary index creates a different ordering when the query begins with something other than the primary key.

That gives us a route to investigate rather than an answer to memorize. We will descend by primary key, continue across a range, disturb the order with an insert and then approach the same row by email.

The page tree explains how InnoDB can avoid most of the table. It has not yet told us what the search actually finds at the bottom. Let us send key 63 down the clustered index and open the matching leaf.

STAGE 2 OF 7

Clustered Leaves

02 CLUSTERED LEAVES

To discover what waits at the bottom, key 63 starts in one root page. Its visible separator values, 55 and 84, divide all customer IDs into three ranges.

Since 63 lies between 55 and 84, only the middle branch can contain it. The root does not hold customer records. It spends its space telling searches which page to open next.

That branch reaches the middle of three sorted leaf pages, all at the same depth. The key is here, but what did InnoDB store beside 63?

PAUSE AND PREDICT

What is stored with key 63?

- A pointer
- The row

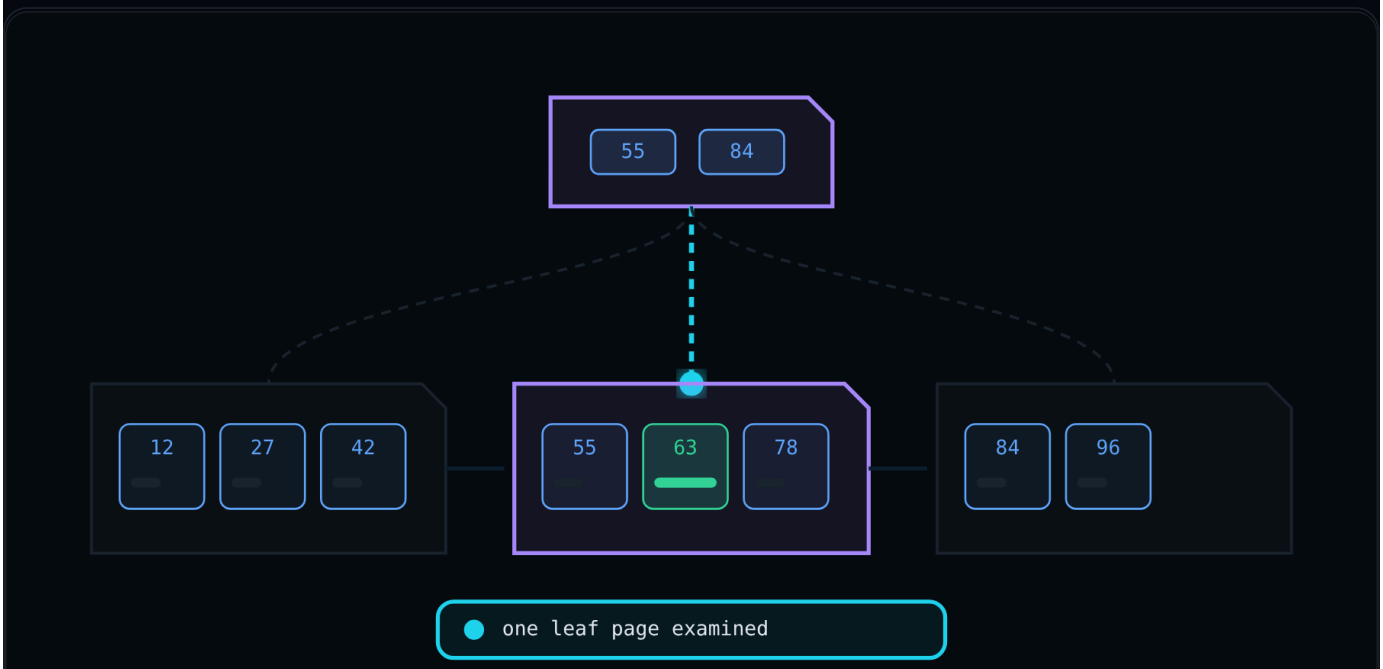
Skip this check

The leaf opens and reveals the complete customer record: ID, email and plan. There is no separate heap address to chase because the clustered index leaf is where InnoDB stores the row.

So the index is not merely a map beside the table. Its upper pages narrow the search, and its primary-key leaves are the table. Because those leaves remain ordered, the same path should also help when a query wants several neighboring IDs.

STAGE 3 OF 7

Search and Range



03 SEARCH AND RANGE

Customer 63 showed us where one search stops. Beside it, consider a second query asking for every ID from 42 through 78. Both can begin in the same primary-key tree.

The point search compares 63 with the root separators. The range search compares its lower bound, 42. Each descends only toward the first leaf that could contain a match.

The range begins at key 42 in the first matching leaf, but its upper bound is 78 in another page. Must it return to the root after 42?

PAUSE AND PREDICT

After key 42, keep scanning?

- Restarts
- Keeps scanning

Skip this check

No restart is needed. The point cursor stops at 63, while the range cursor follows links between neighboring leaves and reads the ordered records until it passes 78.

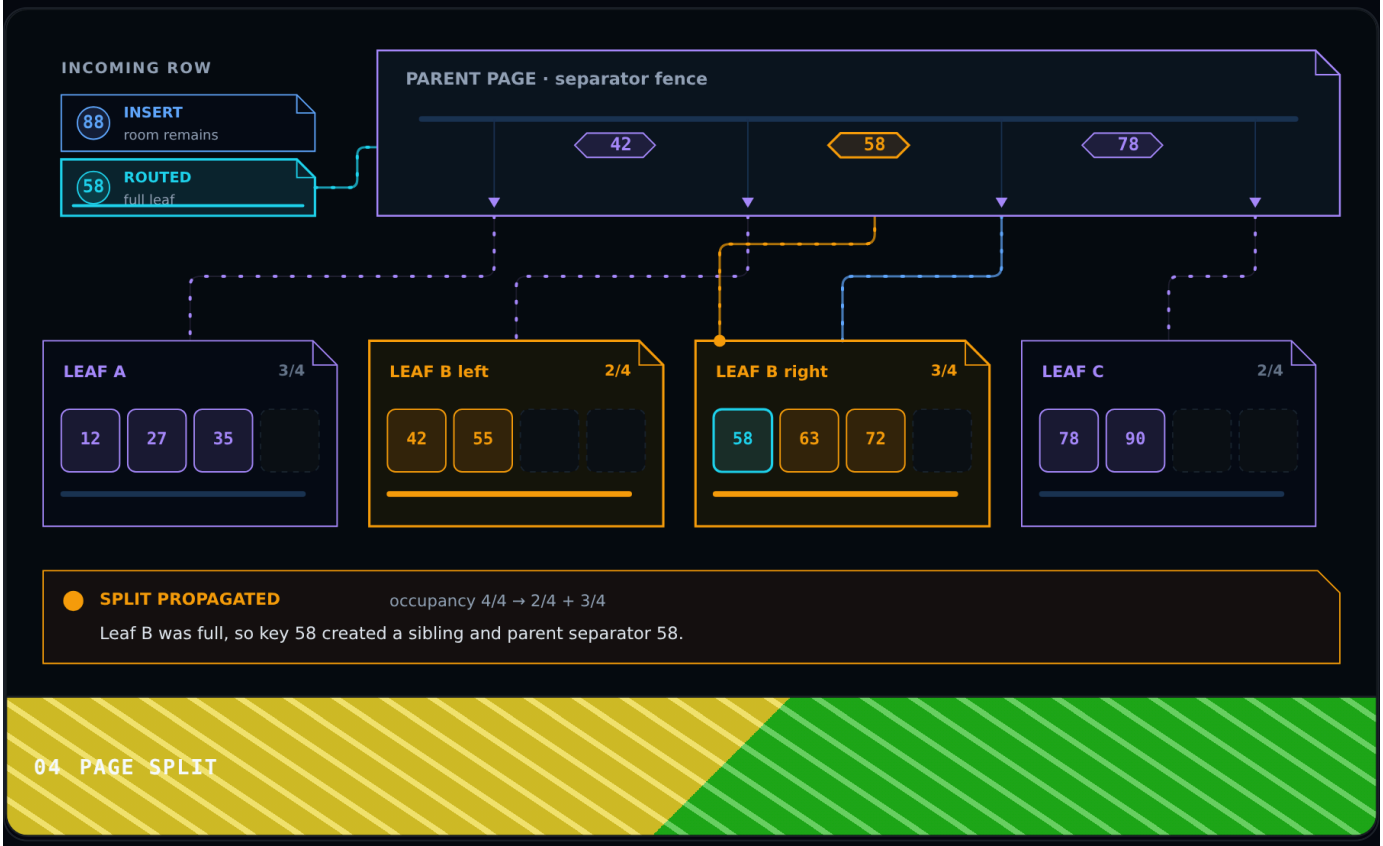
Switch the Query shape control between Point and Range. The first path ends in one leaf, while the second keeps crossing linked leaves until every requested row is collected.

The range stays efficient because every leaf remains in primary-key order and points to its neighbor. That useful order creates the next problem: an inserted record must enter the correct leaf even when

that page is already full.

STAGE 4 OF 7

Page Split



04 PAGE SPLIT

The linked order survives only if each insert lands in its proper page. Choose incoming key 88 or 58: one target has room, while the other is full. Follow the route and see when one leaf becomes two and sends a separator upward.

STAGE 5 OF 7

Secondary Hop

★ If you remember one thing · A covering secondary index ends the query after one tree, while a full-row query descends a second tree.

Try it in Watch mode. Return the requested columns after only one tree descent.

05 SECONDARY HOP

Splitting preserved primary-key order, but many queries do not begin with an ID at all. To find `lin@example.test`, InnoDB needs a second tree ordered by email: a secondary index.

The email search reaches an entry containing `lin@example.test` beside primary key 63. That 63 is not a physical address. It is the key for returning to the clustered tree we already opened.

If the query asks only for email and ID, that secondary entry already contains its answer. A full customer result also needs the plan. Do both SELECT lists stop here?

PAUSE AND PREDICT

Which projection needs two trees?

- Email + id
- Full row

Skip this check

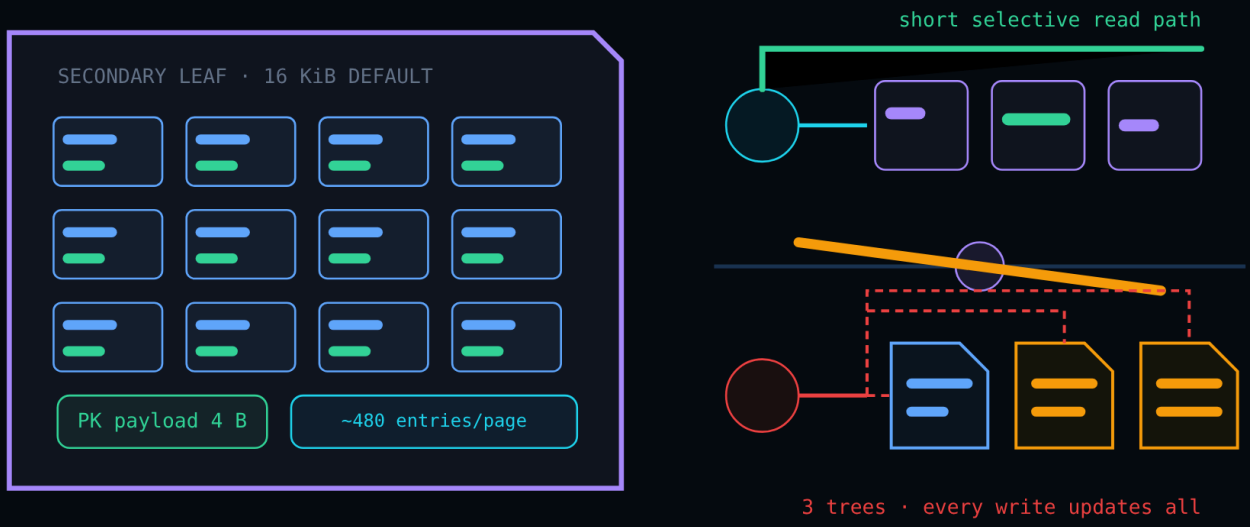
The paths separate. Email + id is a covering result, so one secondary-tree search is enough. Full row carries 63 into the clustered tree and descends again to retrieve the missing plan.

Switch the Projection control between Email + id and Full row. One result fills from the secondary leaf, while the other opens a second cursor and finishes in the clustered leaf.

A secondary index can therefore save a scan and sometimes even the second tree descent. But this shortcut is another ordered structure to store and update, and every one of its entries carries a copy of the primary key.

STAGE 6 OF 7

Tradeoffs



06 TRADEOFFS

That copied primary key gives us a concrete way to price the shortcut. Hold one secondary leaf at InnoDB’s default 16 KiB size, then change only the primary-key payload repeated in every entry.

The benefit is real: selective equality and range queries avoid scanning the table, and a covering index can return every requested column without reopening the clustered tree.

The bill arrives on writes and storage. Each write maintains every affected tree, while each secondary record repeats the primary key. If that copy grows from four bytes to sixteen, what happens inside the same page?

PAUSE AND PREDICT

Do wider keys fit fewer entries?

- Fewer entries
- The same count

Skip this check

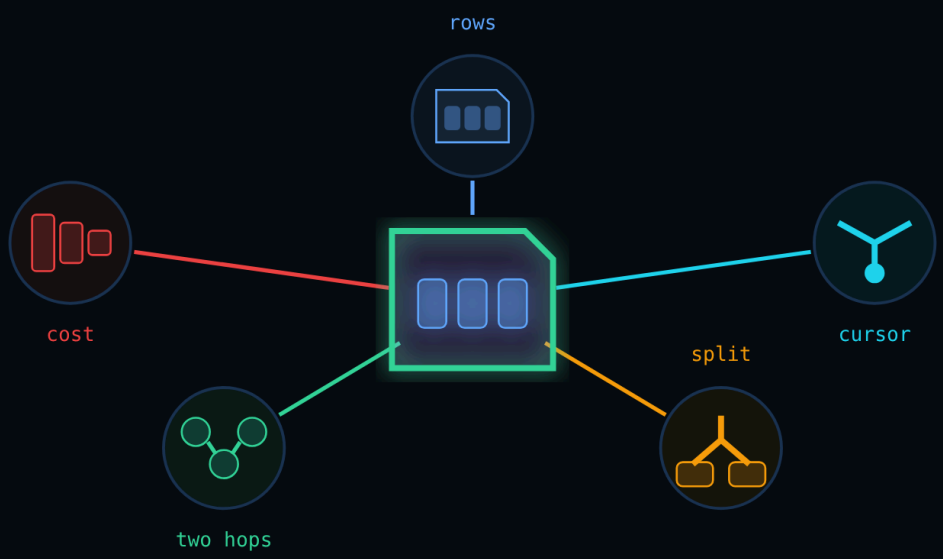
The page does not grow, so wider entries leave room for fewer of them. Across a real index, that lower density can mean more pages, a larger tree, less useful cache and additional I/O.

Switch the Primary key control between INT · 4 B and BINARY(16) · 16 B. The wider key fits fewer entries.

An index is therefore a workload decision, not free speed. A useful tree shortens important reads enough to justify its pages and write maintenance. A short primary key also keeps every secondary copy lighter.

We can now answer the original lookup without treating any of this as magic.

Recap



07 RECAP

Our eight-million-row lookup became a short route because separators discarded whole key ranges. At the end, the clustered leaf did not point elsewhere. It contained customer 63’s complete row.

Keeping those row-bearing leaves sorted gave us more than point lookup. The range query descended once to 42, then followed neighboring leaves until it crossed 78.

That useful order had to survive new rows. When key 58 met a full leaf, InnoDB split the records between siblings and added a separator so later searches could still choose correctly.

Primary-key order could not begin an email search, so the secondary index supplied another route. Its email entry carried key 63, either completing a covering result or reopening the clustered tree.

That copied key also revealed the cost. Wider primary keys packed fewer entries into each secondary page, while every extra tree consumed cache and joined the work of future writes.

So InnoDB’s answer to our original query is a bargain, not a trick. Ordered pages make reads selective, and in return the database stores those pages, splits them when needed and maintains every useful ordering.

The customer values were illustrative, but MySQL’s own documentation supports the mechanisms behind them. You can now reason from a query to its page route, its leaf payload and the storage and write work that made the shortcut possible.