Module 13 of 16

Indexes and performance

B+tree indexes and InnoDB's clustered primary key, selectivity and the cost of every index, composite indexes and the leftmost prefix rule, reading MySQL's EXPLAIN and EXPLAIN ANALYZE, sargable predicates and their rewrites, and covering indexes.

Every slow query conversation ends at the same place: is there an index the query can use, and is the query written so that it can use it? This module explains indexes from the storage up, as MySQL 8's InnoDB engine builds them — pages, B+trees, a table stored as its primary key, secondary indexes that point to it — and then turns that picture into the working rules an engineer applies every day: which columns to index and in what order, how to read the plan the optimizer chose, how to write predicates an index can serve, and how to answer a query from the index alone.

How indexes work covers the B+tree and why it is shallow, the clustered primary key, what a secondary index stores, selectivity and what every index costs. Composite indexes covers the sort order of a multi-column index, the leftmost prefix rule, how a range stops a seek, sorting from an index and choosing the column order. Reading a query plan covers `EXPLAIN`'s columns, the access-type ladder from `const` to `ALL`, the `Extra` notes and `EXPLAIN ANALYZE`. Sargable queries covers functions, arithmetic, wildcards and type conversions on indexed columns, and functional indexes. Covering indexes covers index-only reads, the primary key carried in every secondary index, and deferred joins.

The exercise engine is SQLite behind a MySQL layer, so plans are read rather than printed; the exercises judge the rows. They create indexes and run the queries those indexes serve, measure selectivity, compute the leftmost prefix and index coverage from catalog-style tables, audit stored `EXPLAIN` rows, and rewrite non-sargable predicates without losing an edge row. The checkpoint removes the duplicates that block a `UNIQUE` index, writes an indexed monthly report and finds the indexes nobody reads.

CONTENTS

1	How indexes work — B+trees, clustered and secondary	15 min
2	Composite indexes and the leftmost prefix	15 min
3	Reading a query plan with EXPLAIN	16 min

4	Sargable queries — predicates an index can use	15 min
5	Covering indexes and index-only reads	14 min
✓	Checkpoint — Indexes and performance	25 min

Exercises in this module compile on MySQL 8 syntax on SQLite 3.49. Reference solutions and the answer key follow each lesson.

LESSON 1 · 15 MIN

How indexes work — B+trees, clustered and secondary

A table without an index is a heap of pages. To answer `WHERE email = 'asha@example.com'` the server has to read every page and test every row — a **full table scan**. On ten rows nobody notices; on ten million rows it is the difference between a millisecond and several seconds, and it is the first thing an interviewer asks about when a query is slow. This lesson explains what an index physically is in MySQL 8's storage engine, InnoDB, how the primary key shapes the whole table, what a secondary index stores, and what every index costs.

The book index analogy, made precise

An index is a **second copy of some columns, kept sorted**, with a way back to the full row. Because it is sorted, finding a value is a search, not a scan — the same reason you find "B+tree" in a textbook's index in seconds without reading the book.

```
CREATE TABLE customers (
  id          INT PRIMARY KEY,
  email       VARCHAR(120) NOT NULL,
  city        VARCHAR(60),
  signup_date DATE
);

CREATE INDEX idx_customers_city ON customers (city);
-- MySQL also accepts the same index inside ALTER TABLE:
-- ALTER TABLE customers ADD INDEX idx_customers_city (city);
```

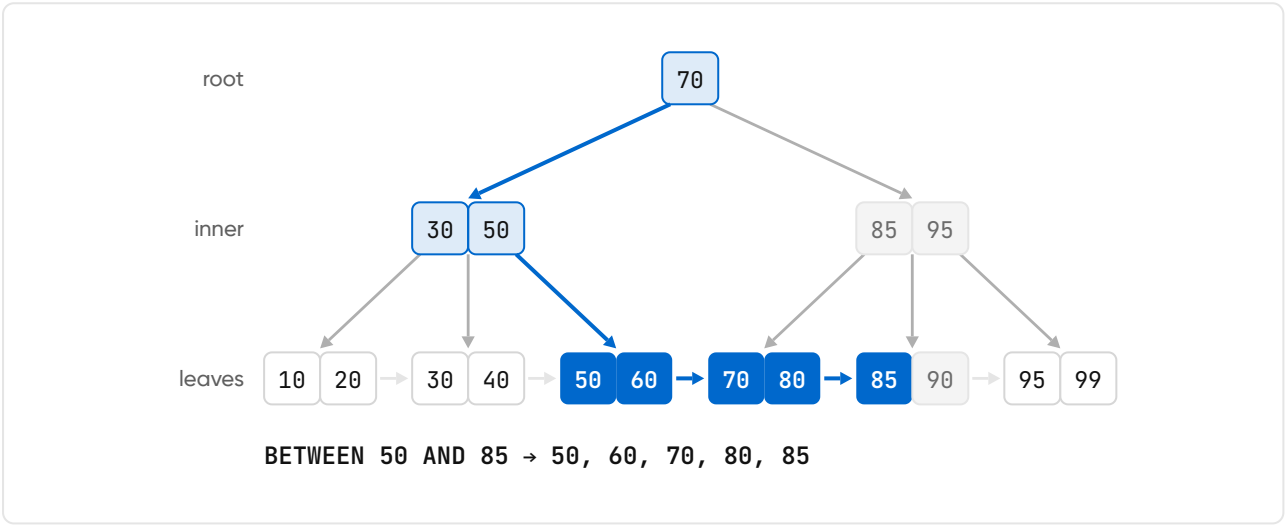
After this, a query that filters on `city` can search `idx_customers_city` for `'Pune'`, read the handful of matching entries, and fetch just those rows. Nothing in the query text changes — `SELECT … WHERE city = 'Pune'` is the same statement with or without the index. Choosing whether to use it is the **optimizer's** job, and it only uses an index when it estimates that is cheaper than scanning.

Why a B+tree

Storage is read in **pages** — InnoDB's default page is 16 KB — and a page read is the unit of cost, whether it comes from disk or from the buffer pool. A B+tree is built for that:

- every **node is one page** and holds hundreds of keys, so the tree is very wide and very shallow;
- the **inner nodes** hold only keys and child pointers, used to steer the search;
- the **leaf nodes** hold the entries themselves, in sorted order, and each leaf is **linked to the next**.

A B+tree: descend once, then walk the leaves



1. The index keeps the keys sorted in leaf pages linked left to right. Each inner page holds only separators that steer a search: here 6 leaves, 2 inner pages and one root.
2. Find 60. The root compares it with its separator: 60 < 70, so the search goes to child 1. One page read.
3. The inner page compares again: 60 ≥ 50, child 3. A second page read, and only one page per level is ever touched.
4. The leaf holds 60: found in 3 page reads. With hundreds of keys per real page, three or four levels cover tens of millions of rows.
5. A range, 50 to 85, descends once to the leaf holding 50, then follows the leaf links sideways, reading 3 leaves and stopping at the first key past 85.

With a few hundred keys per page, three levels address tens of millions of rows. An equality lookup is therefore **three or four page reads**, and the upper levels are almost always cached in memory. A binary tree over the same rows would be about twenty-four levels deep — twenty-four dependent reads. The linked leaves are what make a **range** cheap: `WHERE signup_date BETWEEN '2024-01-01' AND '2024-01-31'` searches down once to the first date and then walks the leaves sideways until it passes the last.

Search cost grows with the logarithm of the table, so doubling the table adds a fraction of a level. That is why "add an index" turns an O(n) scan into roughly O(log n) work.

The clustered index: the table *is* the primary key

In InnoDB the table is not a heap with an index beside it. **The table is stored as a B+tree on the primary key** — the clustered index — and its leaf pages hold the complete rows.

Clustered index (PRIMARY) on `customers.id` — its leaves are the rows:

id	email	city	signup_date
1	asha@…	Pune	2024-01-04
2	ravi@…	Delhi	2024-01-09
3	meera@…	Pune	2024-02-11

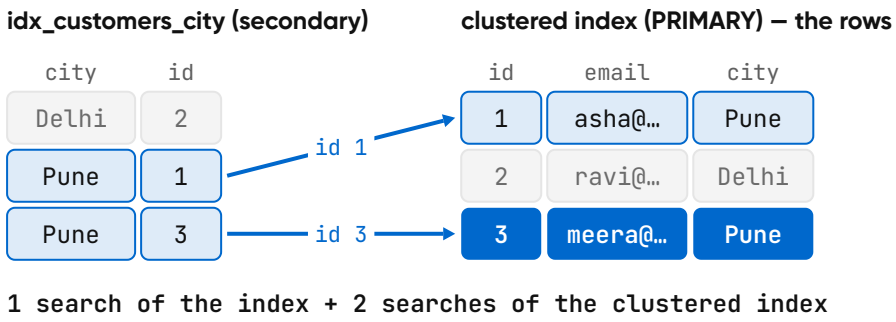
Consequences worth saying out loud in an interview:

- a table has **exactly one** clustered index, because rows can be physically ordered only one way;
- a lookup or range on the primary key reads the rows directly — the fastest access there is;
- if you declare no primary key, InnoDB uses the first `UNIQUE` index whose columns are all `NOT NULL`, and failing that it invents a **hidden 6-byte row id** — so always declare one;
- a short, ever-increasing key (an `INT AUTO_INCREMENT`) appends new rows at the right edge of the tree; a random key such as a UUID string inserts into the middle, splits pages and leaves them half full.

Secondary indexes store the primary key

Every other index is a **secondary** (non-clustered) index. Its leaves hold the indexed column values **plus the primary key** of the row — not a physical address.

A secondary index stores the primary key, not the row



1. InnoDB keeps two trees here. The clustered index is the table itself, rows sorted by id. The secondary index on city holds only each city with its row's id, sorted by city.
2. The search goes down the secondary index to 'Pune' and reads the matching entries side by side: ids 1 and 3. It has the ids, not yet the rows.
3. For the full row it searches the clustered index by id. First id 1: its leaf holds the whole row, city Pune included.
4. Then id 3: one clustered lookup per matching row. Cheap for a few rows; for a third of the table those jumps cost more than a scan, which is why the optimizer may skip the index.

So `SELECT * FROM customers WHERE city = 'Pune'` is two searches: one down `idx_customers_city` to collect ids 1 and 3, then one down the clustered index **per id** to fetch each full row. If the filter matches a few rows, that is cheap. If it matches a third of the table, those per-row lookups jump all over the clustered index and a plain scan wins — which is why the optimizer sometimes ignores an index you created. Two useful corollaries:

- a long primary key makes **every** secondary index bigger, because each entry repeats it;
- a query that needs only the indexed columns and the primary key never visits the clustered index at all — a **covering** index, the subject of a later lesson.

PostgreSQL differs here: its tables are heaps, every index (including the primary key's) points at a physical tuple location, and there is no clustered index maintained automatically (`CLUSTER` reorders once).

Selectivity decides whether an index helps

An index pays off when a lookup returns a **small fraction** of the table. The measure is **selectivity**: distinct values divided by rows.

```sql
SELECT COUNT(DISTINCT email)   / COUNT(*) AS email_selectivity,    -- ≈ 1.0
       COUNT(DISTINCT city)    / COUNT(*) AS city_selectivity,     -- maybe 0.
       COUNT(DISTINCT is_active) / COUNT(*) AS active_selectivity  -- 2 / rows
FROM customers;
```

`email` is ideal. `city` helps when the cities are many and evenly spread. `is_active` (two values) rarely helps on its own: half the table matches, and the optimizer will scan. The server keeps estimates of these counts — the **cardinality** shown by `SHOW INDEX`, refreshed by `ANALYZE TABLE` — and plans from them, not from your intentions.

What an index costs

Indexes are not free, and "why not index every column?" is a classic question. Each index:

- is **written on every change**: an `INSERT` adds an entry to every index, a `DELETE` removes one from each, an `UPDATE` of an indexed column moves its entry;
- takes **disk and buffer pool memory**, crowding out data pages you would rather keep cached;
- gives the optimizer **more plans to weigh**, and occasionally a worse one;
- can be **redundant**: an index on `(customer_id)` is covered by one on `(customer_id, placed_on)`.

A `UNIQUE` index is also a **constraint**: the server refuses a duplicate, so it must check the index before every insert. And it cannot be created while duplicates exist — finding them first is a practical skill you will use in the checkpoint.

Other kinds of index in MySQL

- **B-tree** — the default for InnoDB, and what "index" means unless stated otherwise.
- **FULLTEXT** — word search with `MATCH … AGAINST`, for prose columns where `LIKE '%word%'` cannot use a B-tree.
- **SPATIAL** — R-tree indexes on geometry columns.
- **Hash** — only for `MEMORY` tables; InnoDB builds an internal *adaptive hash index* over hot B-tree pages by itself.
- **Functional indexes** (MySQL 8.0.13+) — an index on an expression, written with double parentheses: `CREATE INDEX idx_year ON orders ((YEAR(placed_on)));`.

- **Invisible indexes** (8.0) — `ALTER TABLE t ALTER INDEX idx INVISIBLE` hides one from the optimizer while it is still maintained, so you can test dropping it safely.

Practise

The exercises run on SQLite behind a MySQL layer, so they judge the rows your queries return, not the plan. You will create an index and run the lookup it serves, and then measure the selectivity of candidate columns — the number you should look at before creating any index.

EXERCISE 1

Index the lookup column

Customer support looks up one customer's orders many times a minute. Create an index named `idx_orders_customer` on `orders (customer_id)`, then return customer **7**'s orders.

`orders`:

id	customer_id	placed_on	amount
1	7	2024-03-02	450
2	3	2024-03-02	120
3	7	2024-01-15	980
4	5	2024-02-01	300
5	7	2024-03-02	75
6	3	2024-04-10	610

Output columns: `id`, `placed_on`, `amount`, ordered by `placed_on`, then `id`.

```
id | placed_on | amount
3 | 2024-01-15 | 980
1 | 2024-03-02 | 450
5 | 2024-03-02 | 75
```

INPUT 1

```
CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT NOT NULL,
  placed_on DATE NOT NULL,
  amount INT NOT NULL
);
INSERT INTO orders VALUES
  (1, 7, '2024-03-02', 450),
  (2, 3, '2024-03-02', 120),
  (3, 7, '2024-01-15', 980),
  (4, 5, '2024-02-01', 300),
  (5, 7, '2024-03-02', 75),
  (6, 3, '2024-04-10', 610);
```

OUTPUT 1

```
id | placed_on | amount
3 | 2024-01-15 | 980
1 | 2024-03-02 | 450
5 | 2024-03-02 | 75
```

REFERENCE SOLUTION

```sql
CREATE INDEX idx_orders_customer ON orders (customer_id);

SELECT id, placed_on, amount
FROM orders
WHERE customer_id = 7
ORDER BY placed_on, id;
```

EXERCISE 2

Measure selectivity before indexing

Before adding indexes to `users`, measure how selective each candidate column is. For each of `email`, `country`, `plan` and `is_active`, return its name, its number of distinct non-NULL values, and its selectivity — distinct values divided by the number of rows in the table, rounded to 4 places.

`users`:

id	email	country	plan	is_active
1	asha@example.com	IN	free	1
2	ravi@example.com	IN	pro	1
3	lee@example.com	US	free	0
4	meera@example.com	IN	free	1
5	tom@example.com	UK	team	1
6	ana@example.com	US	free	1
7	kiran@example.com	IN	pro	0
8	zoya@example.com	IN	free	1

Output columns: `column_name`, `distinct_values`, `selectivity` — one row per column (build them with `UNION ALL`), ordered by `selectivity` descending, then `column_name`.

```
column_name | distinct_values | selectivity
email | 8 | 1
country | 3 | 0.375
plan | 3 | 0.375
is_active | 2 | 0.25
```

INPUT 1

```
CREATE TABLE users (id INT PRIMARY KEY,
email VARCHAR(120), country VARCHAR(2),
plan VARCHAR(10), is_active INT);
INSERT INTO users VALUES
  (1, 'asha@example.com', 'IN', 'free',
1),
  (2, 'ravi@example.com', 'IN', 'pro',
1),
  (3, 'lee@example.com', 'US', 'free',
0),
  (4, 'meera@example.com', 'IN', 'free',
1),
  (5, 'tom@example.com', 'UK', 'team',
1),
  (6, 'ana@example.com', 'US', 'free',
1),
  (7, 'kiran@example.com', 'IN', 'pro',
0),
  (8, 'zoya@example.com', 'IN', 'free',
1);
```

OUTPUT 1

```
column_name | distinct_values |
selectivity
email | 8 | 1
country | 3 | 0.375
plan | 3 | 0.375
is_active | 2 | 0.25
```

REFERENCE SOLUTION

```sql
SELECT 'email' AS column_name,
       COUNT(DISTINCT email) AS distinct_values,
       ROUND(COUNT(DISTINCT email) / COUNT(*), 4) AS selectivity
FROM users
UNION ALL
SELECT 'country', COUNT(DISTINCT country), ROUND(COUNT(DISTINCT country) / COUNT(*),
FROM users
UNION ALL
SELECT 'plan', COUNT(DISTINCT plan), ROUND(COUNT(DISTINCT plan) / COUNT(*), 4)
FROM users
UNION ALL
SELECT 'is_active', COUNT(DISTINCT is_active), ROUND(COUNT(DISTINCT is_active) / COU
FROM users
ORDER BY selectivity DESC, column_name;
```

QUIZ

1. In InnoDB, what is stored in the leaf pages of the clustered index?

A. Only the primary key values

B. The complete rows, in primary key order

C. Physical row addresses

D. The secondary index entries

2. What does an InnoDB secondary index entry use to point back to its row?

A. A physical page and slot number

B. The row's primary key value

C. A hash of the row

D. The hidden row id, always

3. Why is a B+tree only three or four levels deep for millions of rows?

A. It is rebalanced into a binary tree on every insert

B. Each node is a whole page holding hundreds of keys, so the fan-out is huge

C. It stores only distinct values

D. MySQL caps tables at four levels

4. Which column is the **worst** candidate for a single-column index on a 10-million-row `users` table?

A. `email` (unique)

B. `phone` (almost unique)

C. `is_active` (0 or 1, mostly 1)

D. `username` (unique)

5. Why not create an index on every column of a table?

A. MySQL allows only one index per table

B. Every index must be updated on each write and takes disk and buffer-pool memory, while unused ones speed up nothing

C. Indexes make SELECT slower

D. Indexes change query results

ANSWER KEY · HOW INDEXES WORK — B+TREES, CLUSTERED AND SECONDARY

1. B InnoDB stores the table **as** the B+tree on the primary key, so its leaves hold the full rows. That is why a table has exactly one clustered index and why a primary key range scan reads rows directly.

2. B Secondary leaves store the indexed columns plus the primary key, so a lookup searches the clustered index by that key. The hidden 6-byte row id is used only when the table has no primary key and no unique NOT NULL index. PostgreSQL, by contrast, points at a physical tuple location.

3. B With hundreds of keys per 16 KB page, each level multiplies the reachable entries by hundreds: three levels reach tens of millions. A binary tree has a fan-out of two and would need about twenty-four levels.

4. C `is_active` has a selectivity near zero: a lookup on the common value returns most of the table, so the optimizer will scan anyway, while the index still costs every write. The other three return one row per lookup.

5. B Indexes speed up the reads that use them and tax every INSERT, UPDATE and DELETE. Index the columns real queries filter, join and sort on. A table may have up to 64 secondary indexes in InnoDB, so the limit is cost, not a rule.

LESSON 2 · 15 MIN

Composite indexes and the leftmost prefix

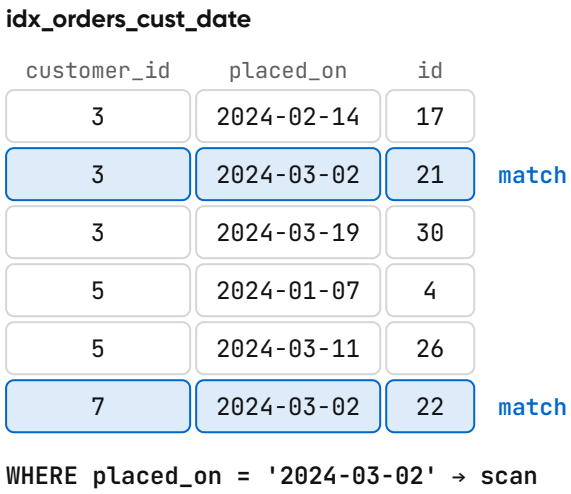
Real queries filter on more than one column: a customer's orders in a date range, a city's users by signup date, a product's reviews by rating. One index per column is not the answer — MySQL generally searches one index per table, and intersecting two (`index_merge`) is a fallback, not a plan you design for. The tool is a **composite index**, one B+tree over several columns. This lesson explains how its entries are ordered, the leftmost prefix rule that follows from it, how a range condition changes things, and how to choose the column order.

How the entries are sorted

A composite index sorts by its first column; among equal first values, by the second; among equal pairs, by the third — exactly like a phone book sorted by surname, then first name.

```
CREATE INDEX idx_orders_cust_date ON orders (customer_id, placed_on);
```

A composite index is sorted by its first column, then its second



1. The entries are sorted by customer_id, and among equal customer_ids by placed_on — a phone book by surname, then first name. Each customer's dates are in order; across customers they are not.
2. WHERE customer_id = 3: customer 3's entries sit together, so the search descends once to the first and reads 3 in a row. The leftmost column can always seek.
3. Add placed_on >= '2024-03-01': within customer 3 the dates are sorted, so the run just starts later — 2 entries. Equality on the first column, then a range on the second, both narrow the seek.
4. WHERE placed_on = '2024-03-02' alone: the two matches are rows 2 and 6, under different customers, with others between them. No single run holds them, so the index cannot seek on its second column.

Everything else follows from this picture. Customer 3's orders sit together, and within them the dates are in order. But the dates **across** customers are not in order — 2024-03-02 appears twice, far apart.

The leftmost prefix rule

An index on `(a, b, c)` can be **searched** through any leftmost prefix of its columns:

Filter	Uses the index to seek?
`a = 1`	yes — prefix (a)
`a = 1 AND b = 2`	yes — prefix (a, b)
`a = 1 AND b = 2 AND c = 3`	yes — all three
`b = 2`	no — the b values are scattered
`b = 2 AND c = 3`	no
`a = 1 AND c = 3`	partly — seeks on a, then filters c

The last row is the subtle one: with `b` missing, the server seeks to `a = 1` and then has to examine every entry for that `a`, testing `c`. MySQL does that test inside the index rather than after fetching each row — **index condition pushdown**, shown as `Using index condition` in a plan — but it is still reading all of `a = 1`'s entries.

The order of conditions in the `WHERE` clause is irrelevant: `WHERE b = 2 AND a = 1` matches `(a, b)` just as well. Only the order **inside the index definition** matters.

Two corollaries interviewers like:

- an index on `(a)` is **redundant** next to one on `(a, b)` — drop it;
- an index on `(b)` is **not** redundant next to `(a, b)` — it serves queries the composite cannot.

A range stops the seek

Once a column is compared with a **range** (`<`, `>`, `BETWEEN`, `LIKE 'abc%'`), the columns after it can no longer narrow the seek, because within a range of `a` values the `b` values are not

sorted.

```
-- index (customer_id, placed_on, status)

WHERE customer_id = 3 AND placed_on ⩾ '2024-03-01' AND status = 'paid'
-- seek: customer_id = 3, placed_on from 2024-03-01 onwards (two columns)
-- status: tested on each entry in that range, not used to seek
```

The rule of thumb that falls out: **equality columns first, then the one range column.** If the query above is common, `(customer_id, status, placed_on)` serves it fully — two equalities, then the range — and reads only the matching entries.

Sorting comes for free — when the prefix fits

Because the entries are already in order, a composite index can also satisfy `ORDER BY` without a sort step:

```
SELECT id, placed_on, amount
FROM orders
WHERE customer_id = 3
ORDER BY placed_on;          -- index (customer_id, placed_on): no filesort
```

The equality on `customer_id` fixes the first column, and the remaining entries come out ordered by `placed_on`. `ORDER BY placed_on DESC` works too: InnoDB reads the leaves backwards (`Backward index scan` in the plan). MySQL 8 also supports **descending index columns**, `(customer_id, placed_on DESC)`, which matter when one query sorts the columns in mixed directions, such as `ORDER BY a ASC, b DESC`.

Remove the equality and the order breaks: `ORDER BY placed_on` alone across all customers cannot read this index in order, so the server sorts (`Using filesort`).

Choosing the column order

There is no single rule, but these questions settle almost every case:

1. **Which columns are compared with equality?** They go first.
2. **Which column is the range or the sort?** It goes last among the useful columns.

3. **Which prefixes do other queries need?** Order the equality columns so that the index's prefixes serve as many queries as possible — one well-ordered index often replaces three.
4. **Only then, selectivity.** Putting the most selective column first is common advice, but prefix reuse matters more: an index on `(country, city)` serves "all of India" and "Pune, India"; `(city, country)` serves only the second.

```
-- queries:  WHERE tenant_id = ? AND created_at ⩾ ?
--           WHERE tenant_id = ? AND user_id = ?
--           WHERE tenant_id = ?
-- one index for the first and third, one for the second:
CREATE INDEX idx_events_tenant_time ON events (tenant_id, created_at);
CREATE INDEX idx_events_tenant_user ON events (tenant_id, user_id);
```

Skip scan: the exception, not the plan

MySQL 8.0.13 added the **index skip scan**: with an index on `(gender, birth_date)` and a filter on `birth_date` alone, the optimizer may seek once for each distinct `gender` value and range-scan inside each. It only pays off when the skipped leading column has very few distinct values, and it shows as `Using index for skip scan`. Design for the leftmost prefix; treat a skip scan as a happy accident. PostgreSQL 18 added a similar skip scan for B-tree indexes; older PostgreSQL versions could scan a multicolumn index for a non-leading column, but much less efficiently.

Limits worth knowing

- An InnoDB index can have at most **16 columns**, and its key size is capped (3,072 bytes with the default row format) — long `VARCHAR`s can need a **prefix index**, `INDEX (title(40))`, which cannot cover a query or serve `ORDER BY title`.
- Every secondary entry also carries the **primary key**, so `(a, b)` on a table keyed by `id` is effectively `(a, b, id)` — which is why `WHERE a = 1 ORDER BY id` can avoid a sort.
- More columns mean a bigger index and more write work. Add a column because a query needs it, not in case one might.

Practise

The first exercise creates a composite index and runs the customer-and-date-range query it is built for. The second works the leftmost prefix rule out in SQL: given each index's columns in order and each query's equality filters, compute how many leading columns each index can seek on.

EXERCISE 1

A customer's month, from one index

The account page shows one customer's orders for one month. Create an index named `idx_orders_cust_date` on `orders (customer_id, placed_on)` — equality column first, range column second — then return customer **3**'s orders placed in **March 2024**. Write the month as a range on the bare `placed_on` column.

`orders`:

id	customer_id	placed_on	amount
1	3	2024-02-14	250
2	3	2024-03-02	180
3	5	2024-03-11	900
4	3	2024-03-19	420
5	7	2024-03-02	60
6	3	2024-04-01	310
7	3	2024-03-02	95

Output columns: `id`, `placed_on`, `amount`, ordered by `placed_on`, then `id`.

```
id | placed_on | amount
2 | 2024-03-02 | 180
7 | 2024-03-02 | 95
4 | 2024-03-19 | 420
```

INPUT 1

```
CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT NOT NULL,
  placed_on DATE NOT NULL,
  amount INT NOT NULL
);
INSERT INTO orders VALUES
  (1, 3, '2024-02-14', 250),
  (2, 3, '2024-03-02', 180),
  (3, 5, '2024-03-11', 900),
  (4, 3, '2024-03-19', 420),
  (5, 7, '2024-03-02', 60),
  (6, 3, '2024-04-01', 310),
  (7, 3, '2024-03-02', 95);
```

OUTPUT 1

```
id | placed_on | amount
2 | 2024-03-02 | 180
7 | 2024-03-02 | 95
4 | 2024-03-19 | 420
```

REFERENCE SOLUTION

```sql
CREATE INDEX idx_orders_cust_date ON orders (customer_id, placed_on);

SELECT id, placed_on, amount
FROM orders
WHERE customer_id = 3
  AND placed_on ⩾ '2024-03-01'
  AND placed_on < '2024-04-01'
ORDER BY placed_on, id;
```

EXERCISE 2

Work out the leftmost prefix

A catalog table lists each index's columns in order (`seq` starts at 1), and another lists the columns each query compares with **equality**. For every pair of a query and an index, compute how many leading columns of the index the query can seek on: the count of consecutive index columns, from `seq` 1, that all appear among the query's filter columns (0 when the first column is missing).

`index_columns` :

index_name	seq	column_name
idx_cust_date	1	customer_id
idx_cust_date	2	placed_on
idx_status_cust_date	1	status
idx_status_cust_date	2	customer_id
idx_status_cust_date	3	placed_on
idx_date	1	placed_on

`query_filters`:

query_id	column_name
1	customer_id
2	customer_id
2	placed_on
3	status
3	placed_on
4	placed_on

Output columns: `query_id`, `index_name`, `usable_columns` — one row for every query in `query_filters` and every index in `index_columns`, ordered by `query_id`, then `index_name`.

```
query_id | index_name | usable_columns
1 | idx_cust_date | 1
1 | idx_date | 0
1 | idx_status_cust_date | 0
2 | idx_cust_date | 2
2 | idx_date | 1
2 | idx_status_cust_date | 0
3 | idx_cust_date | 0
3 | idx_date | 1
3 | idx_status_cust_date | 1
4 | idx_cust_date | 0
4 | idx_date | 1
4 | idx_status_cust_date | 0
```

INPUT 1

```
CREATE TABLE index_columns (index_name
VARCHAR(40), seq INT, column_name
VARCHAR(40));
INSERT INTO index_columns VALUES
  ('idx_cust_date', 1, 'customer_id'),
('idx_cust_date', 2, 'placed_on'),
  ('idx_status_cust_date', 1, 'status'),
('idx_status_cust_date', 2,
'customer_id'), ('idx_status_cust_date',
3, 'placed_on'),
  ('idx_date', 1, 'placed_on');
CREATE TABLE query_filters (query_id
INT, column_name VARCHAR(40));
INSERT INTO query_filters VALUES
  (1, 'customer_id'), (2,
'customer_id'), (2, 'placed_on'),
  (3, 'status'), (3, 'placed_on'), (4,
'placed_on');
```

OUTPUT 1

```
query_id | index_name | usable_columns
1 | idx_cust_date | 1
1 | idx_date | 0
1 | idx_status_cust_date | 0
2 | idx_cust_date | 2
2 | idx_date | 1
2 | idx_status_cust_date | 0
3 | idx_cust_date | 0
3 | idx_date | 1
3 | idx_status_cust_date | 1
4 | idx_cust_date | 0
4 | idx_date | 1
4 | idx_status_cust_date | 0
```

REFERENCE SOLUTION

```sql
SELECT q.query_id,
       i.index_name,
       COALESCE(
         (SELECT MIN(ic.seq)
          FROM index_columns ic
          WHERE ic.index_name = i.index_name
            AND ic.column_name NOT IN (SELECT f.column_name
                                       FROM query_filters f
                                       WHERE f.query_id = q.query_id)) - 1,
         (SELECT MAX(ic.seq) FROM index_columns ic WHERE ic.index_name = i.index_nam
       ) AS usable_columns
FROM (SELECT DISTINCT query_id FROM query_filters) AS q
CROSS JOIN (SELECT DISTINCT index_name FROM index_columns) AS i
ORDER BY q.query_id, i.index_name;
```

QUIZ

1. With an index on `(a, b, c)`, which `WHERE` clause can **not** seek on the index?

A. `a = 1`

B. `a = 1 AND b = 2`

C. `b = 2 AND c = 3`

D. `c = 3 AND b = 2 AND a = 1`

2. Index `(customer_id, placed_on, status)`. For `WHERE customer_id = 3 AND placed_on ⩾ '2024-03-01' AND status = 'paid'`, how many columns narrow the seek?

A. One

B. Two — the range on `placed_on` stops `status` being used to seek

C. Three

D. None, because of the range

3. A table has indexes on `(user_id)` and on `(user_id, created_at)`. What should you do?

A. Keep both: each serves different queries

B. Drop `(user_id)`: the composite's leading column serves every query it could

C. Drop the composite: single-column indexes are faster

D. Add an index on `(created_at, user_id)` too

4. Index `(customer_id, placed_on)`. Which query avoids a sort step?

A. `SELECT … ORDER BY placed_on`

B. `SELECT … WHERE customer_id = 3 ORDER BY placed_on`

C. `SELECT … WHERE placed_on > '2024-01-01' ORDER BY customer_id, amount`

D. `SELECT … ORDER BY amount`

ANSWER KEY · COMPOSITE INDEXES AND THE LEFTMOST PREFIX

1. C Without `a`, the `b` and `c` values are scattered across the index, so there is no leftmost prefix to seek on. The order of conditions in the text is irrelevant, which is why the last option uses all three columns.

2. B The seek uses `customer_id` (equality) and the range on `placed_on`; within that range `status` is not sorted, so it is only tested on each entry (index condition pushdown). Reordering to `(customer_id, status, placed_on)` lets all three narrow it.

3. B Any query that can use `(user_id)` can use the leftmost prefix of `(user_id, created_at)`, so the single-column index is redundant and only costs writes and memory.

4. B With `customer_id` fixed by equality, the remaining entries are already in `placed_on` order, so they stream out sorted. Across all customers the dates are not in order, and `amount` is not in the index at all.

LESSON 3 · 16 MIN

Reading a query plan with EXPLAIN

An index you created is a hope; a **query plan** is the server telling you what it will actually do. Every performance conversation — in a code review, an incident or an interview — ends with "what does `EXPLAIN` say?". This lesson teaches MySQL 8's plan output column by column, the access-type ladder you should know by heart, the `Extra` notes that signal trouble, and `EXPLAIN ANALYZE`, which runs the query and shows the real numbers.

The exercise engine here is SQLite, whose `EXPLAIN QUERY PLAN` looks nothing like MySQL's, so this lesson is reading: the plans below are MySQL's, and the exercises analyse plan rows stored in a table.

Asking for a plan

Put `EXPLAIN` in front of a `SELECT` (or an `UPDATE`, `DELETE`, `INSERT … SELECT`). The query is planned, not run.

```
EXPLAIN
SELECT o.id, o.amount
FROM orders o
JOIN customers c ON c.id = o.customer_id
WHERE c.city = 'Pune'
ORDER BY o.amount DESC;
```

id	select_type	table	type	possible_keys	key	key_len	ref
1	SIMPLE	c	ref	PRIMARY,idx_city	idx_city	243	co
1	SIMPLE	o	ref	idx_orders_customer	idx_orders_customer	4	sh

One row per table access, listed in the **join order** the optimizer chose: here it starts from the Pune customers and, for each, looks up their orders.

The columns

- **id** and **select_type** — which `SELECT` the row belongs to: `SIMPLE` for a query without subqueries or unions, `PRIMARY` for the outer one, `SUBQUERY`, `DERIVED` (a derived table in `FROM`), `UNION`.
- **table** — the table or alias accessed in this step.
- **type** — *how* rows are found. The most important column; see the ladder below.
- **possible_keys** — indexes that could apply. `NULL` here means no index matches the predicates at all.
- **key** — the index actually chosen, or `NULL` for none.
- **key_len** — how many bytes of the index key are used: on a composite index, a short `key_len` tells you only the first column was used to seek.
- **ref** — what the key is compared with: `const` for a literal, `db.table.column` for a join column.
- **rows** — the **estimated** number of rows examined in this step, per lookup. For a nested-loop join, multiply the steps' `rows` (and `filtered`) to estimate the total work.
- **filtered** — the estimated percentage of those rows that survive the remaining conditions.
- **Extra** — everything else, and often the most revealing part.

The access-type ladder

From best to worst — memorise the order:

type	Meaning
`system` / `const`	at most one matching row, found by a primary key or unique index compared with a constant; read once
`eq_ref`	in a join, exactly one row per row of the previous table, through a primary key or unique `NOT NULL` index
`ref`	all rows matching one value of a non-unique index (or a prefix of a unique one)
`ref_or_null`	like `ref`, plus the `IS NULL` rows
`index_merge`	several indexes searched and their results combined
`range`	a range of one index: `<`, `>`, `BETWEEN`, `IN (…)`, `LIKE 'abc%'`
`index`	the **whole index** read from start to end — a full scan, but of the smaller structure
`ALL`	the **whole table** read — a full table scan

`ALL` on a big table is the usual culprit. `index` is better only because the index is smaller than the table; it is still reading everything. Seeing `ALL` on the **inner** table of a join is the worst case: the whole table read once per outer row. MySQL 8.0.18 and later replace that with a hash join, shown as `Using join buffer (hash join)` — better, but still a sign an index on the join column is missing.

Extra: what to look for

Good signs:

- `Using index` — a **covering index**: the step is answered from the index alone, never touching the rows.
- `Using index condition` — **index condition pushdown**: part of the `WHERE` is tested on index entries before rows are fetched.
- `Using index for skip scan`, `Backward index scan` — the index is used in a less obvious way.

Warning signs:

- `Using filesort` — the rows are sorted for `ORDER BY` / `GROUP BY` because no index delivers that order. Not a file necessarily; it is any sort the server performs. Cheap on fifty rows, expensive on five million.
- `Using temporary` — an internal temporary table is built, typically for a `GROUP BY` or `DISTINCT` on columns from different tables, or a `UNION`.
- `Using where` with `type = ALL` — every row is read and most are thrown away.
- `Using join buffer` — a join without a usable index on the inner table.

Neutral: `Using where` on its own just means a condition is checked after the access method.

A slow query, diagnosed

```sql
EXPLAIN SELECT * FROM orders WHERE YEAR(placed_on) = 2024 ORDER BY amount;
```

table	type	possible_keys	key	rows	filtered	Extra
orders	ALL	NULL	NULL	498211	100.00	Using where; Using filesort

Read it left to right: full table scan (`ALL`), no candidate index (`possible_keys` `NULL` — even though `placed_on` is indexed, because the column is wrapped in a function), half a million rows examined, and then a sort. The fix is a predicate the index can use — `placed_on ⩾ '2024-01-01' AND placed_on < '2025-01-01'` — after which the plan shows `type: range`, `key: idx_orders_placed_on` and far fewer `rows`. The next lesson is about exactly that rewrite.

EXPLAIN ANALYZE: the real numbers

`EXPLAIN` shows estimates. `EXPLAIN ANALYZE` (MySQL 8.0.18+) **runs** the query and prints the plan as a tree, each node with the estimate *and* what happened:

```
→ Sort: o.amount DESC  (actual time=4.21..4.30 rows=1083 loops=1)
    → Nested loop inner join  (cost=520.4 rows=1080) (actual time=0.09..3.62
        → Covering index lookup on c using idx_city (city='Pune')  (cost=12.3
        → Index lookup on o using idx_orders_customer (customer_id=c.id)  (co
```

Read it from the most indented line outwards. `actual time=first..last` is milliseconds to the first and last row, `rows` is per loop, and `loops` is how many times the node ran — `rows=9.2 loops=118` is about 1,083 rows in total. The biggest gap between estimated and actual rows is where the optimizer was misled, usually by stale statistics (`ANALYZE TABLE orders;` refreshes them). Since `EXPLAIN ANALYZE` really executes the statement, never point it at a slow `DELETE` on production. `EXPLAIN FORMAT=TREE` gives the same tree without running it, and `FORMAT=JSON` adds the cost model's figures.

PostgreSQL's equivalents are `EXPLAIN` and `EXPLAIN (ANALYZE, BUFFERS)`, with node names such as `Seq Scan`, `Index Scan`, `Index Only Scan` and `Bitmap Heap Scan` in place of MySQL's `type` column.

Practise

Both exercises work on plan rows stored as data — the columns of MySQL's `EXPLAIN` in a table. The first lists the full table scans, biggest first. The second ranks each step's access type on the ladder and reports each query's weakest step and whether it sorts.

EXERCISE 1

Find the full table scans

`plan_rows` stores the `EXPLAIN` output of several slow queries, one row per table access. List every **full table scan** (`type = 'ALL'`), with a hint for the fix: `'no candidate index'` when `possible_keys` is NULL, otherwise `'index not chosen'`.

`plan_rows`:

query_id	step	table_name	type	possible_keys	key_name	rows_examined	ex
1	1	customers	ref	idx_city	idx_city	120	U in
1	2	orders	ALL	NULL	NULL	498211	U w U b (h jo
2	1	orders	range	idx_placed_on	idx_placed_on	3100	U in c U fil
3	1	products	ALL	idx_category	NULL	2400	U w
3	2	reviews	ref	idx_product	idx_product	14	N
4	1	users	const	PRIMARY	PRIMARY	1	N
5	1	events	index	idx_tenant_time	idx_tenant_time	75000	U in U fil

Output columns: `query_id`, `table_name`, `rows_examined`, `fix_hint`, ordered by `rows_examined` descending, then `query_id`, then `table_name`.

```
query_id | table_name | rows_examined | fix_hint
1 | orders | 498211 | no candidate index
3 | products | 2400 | index not chosen
```

INPUT 1

```
CREATE TABLE plan_rows (
  query_id INT,
  step INT,
  table_name VARCHAR(30),
  type VARCHAR(20),
  possible_keys VARCHAR(80),
  key_name VARCHAR(40),
  rows_examined INT,
  extra VARCHAR(120)
);
INSERT INTO plan_rows VALUES
  (1, 1, 'customers', 'ref', 'idx_city',
'idx_city', 120, 'Using index'),
  (1, 2, 'orders', 'ALL', NULL, NULL,
498211, 'Using where; Using join buffer
(hash join)'),
  (2, 1, 'orders', 'range',
'idx_placed_on', 'idx_placed_on', 3100,
'Using index condition; Using
filesort'),
  (3, 1, 'products', 'ALL',
'idx_category', NULL, 2400, 'Using
where'),
  (3, 2, 'reviews', 'ref',
'idx_product', 'idx_product', 14, NULL),
  (4, 1, 'users', 'const', 'PRIMARY',
'PRIMARY', 1, NULL),
  (5, 1, 'events', 'index',
'idx_tenant_time', 'idx_tenant_time',
75000, 'Using index; Using filesort');
```

OUTPUT 1

```
query_id | table_name | rows_examined |
fix_hint
1 | orders | 498211 | no candidate index
3 | products | 2400 | index not chosen
```

REFERENCE SOLUTION

```sql
SELECT query_id,
       table_name,
       rows_examined,
       CASE WHEN possible_keys IS NULL THEN 'no candidate index'
            ELSE 'index not chosen' END AS fix_hint
FROM plan_rows
WHERE type = 'ALL'
ORDER BY rows_examined DESC, query_id, table_name;
```

EXERCISE 2

Each query's weakest step

Rank every step of `plan_rows` on MySQL's access-type ladder — `system` 1, `const` 2, `eq_ref` 3, `ref` 4, `ref_or_null` 5, `index_merge` 6, `range` 7, `index` 8, `ALL` 9 — and report, for each query, the type of its **worst** step (highest rank) and whether any of its steps sorts (`extra` contains `Using filesort`). The same `plan_rows` table as the previous exercise; `extra` can be NULL.

Output columns: `query_id`, `worst_type`, `needs_sort` (`'yes'` or `'no'`), ordered by `query_id`.

```
query_id | worst_type | needs_sort
1 | ALL | no
2 | range | yes
3 | ALL | no
4 | const | no
5 | index | yes
```

INPUT 1

```
CREATE TABLE plan_rows (
  query_id INT,
  step INT,
  table_name VARCHAR(30),
  type VARCHAR(20),
  possible_keys VARCHAR(80),
  key_name VARCHAR(40),
  rows_examined INT,
  extra VARCHAR(120)
);
INSERT INTO plan_rows VALUES
  (1, 1, 'customers', 'ref', 'idx_city',
'idx_city', 120, 'Using index'),
  (1, 2, 'orders', 'ALL', NULL, NULL,
498211, 'Using where; Using join buffer
(hash join)'),
  (2, 1, 'orders', 'range',
'idx_placed_on', 'idx_placed_on', 3100,
'Using index condition; Using
filesort'),
  (3, 1, 'products', 'ALL',
'idx_category', NULL, 2400, 'Using
where'),
  (3, 2, 'reviews', 'ref',
'idx_product', 'idx_product', 14, NULL),
  (4, 1, 'users', 'const', 'PRIMARY',
'PRIMARY', 1, NULL),
  (5, 1, 'events', 'index',
'idx_tenant_time', 'idx_tenant_time',
75000, 'Using index; Using filesort');
```

OUTPUT 1

```
query_id | worst_type | needs_sort
1 | ALL | no
2 | range | yes
3 | ALL | no
4 | const | no
5 | index | yes
```

REFERENCE SOLUTION

```sql
WITH ranked AS (
  SELECT query_id,
         type,
         CASE type
           WHEN 'system' THEN 1 WHEN 'const' THEN 2 WHEN 'eq_ref' THEN 3
           WHEN 'ref' THEN 4 WHEN 'ref_or_null' THEN 5 WHEN 'index_merge' THEN 6
           WHEN 'range' THEN 7 WHEN 'index' THEN 8 WHEN 'ALL' THEN 9
         END AS rnk,
         CASE WHEN extra LIKE '%Using filesort%' THEN 1 ELSE 0 END AS sorts
  FROM plan_rows
),
per_query AS (
  SELECT query_id, MAX(rnk) AS worst_rank, MAX(sorts) AS any_sort
  FROM ranked
  GROUP BY query_id
)
SELECT p.query_id,
       (SELECT MIN(r.type) FROM ranked r
        WHERE r.query_id = p.query_id AND r.rnk = p.worst_rank) AS worst_type,
       IF(p.any_sort = 1, 'yes', 'no') AS needs_sort
FROM per_query p
ORDER BY p.query_id;
```

QUIZ

1. Which `type` values are listed from **best** to **worst**?

A. `ALL`, `index`, `range`, `ref`, `const`

B. `const`, `eq_ref`, `ref`, `range`, `index`, `ALL`

C. `index`, `const`, `ref`, `ALL`, `range`

D. `eq_ref`, `const`, `ALL`, `ref`, `index`

2. A plan row shows `possible_keys: NULL`, `key: NULL`, `type: ALL`. What does it tell you?

A. An index exists but the optimizer found the scan cheaper

B. No index matches the predicates on this table at all

C. The query is covered

D. Statistics are stale

3. What does `Using index` in `Extra` mean?

A. An index was used to find rows, then rows were fetched

B. The step is answered from the index alone — a covering index

C. Index condition pushdown

D. The index was used only for sorting

4. How does `EXPLAIN ANALYZE` differ from `EXPLAIN`?

A. It only formats the output as JSON

B. It actually executes the query and reports real times, row counts and loops beside the estimates

C. It suggests indexes to create

D. It works only on PostgreSQL

5. Which `Extra` note means the server sorted the rows itself because no index provided the order?

A. `Using where`

B. `Using filesort`

C. `Using index`

D. `Backward index scan`

ANSWER KEY · READING A QUERY PLAN WITH EXPLAIN

1. B `const` reads at most one row by a unique key; `eq_ref` one row per outer row in a join; `ref` all rows for one value; `range` a slice of an index; `index` the whole index; `ALL` the whole table.

2. B `possible_keys` NULL means no index is even a candidate — either none exists on the filtered columns or the predicate hides the column, for example `YEAR(placed_on) = 2024`. When an index is listed in `possible_keys` but `key` is NULL, the optimizer considered it and preferred the scan.

3. B `Using index` means no row lookups were needed. `Using index condition` (pushdown) is a different note: it filters index entries before fetching rows, which still fetches the survivors.

4. B MySQL 8.0.18 added `EXPLAIN ANALYZE`, which runs the statement and prints a tree with `actual time`, `rows` and `loops`. Because it executes, never use it on a slow data-changing statement in production. `FORMAT=TREE` shows the tree without running.

5. B `Using filesort` is any sort the server performs for `ORDER BY` or `GROUP BY`, in memory or on disk. `Backward index scan` is the good case: the index delivers a descending order by reading leaves backwards.

LESSON 4 · 15 MIN

Sargable queries — predicates an index can use

You created the right index and the query still scans the whole table. Nine times out of ten the index is fine and the **predicate** is the problem: something in the `WHERE` clause hides the column from the index. A predicate an index can search on is called **sargable** — from *Search ARGument ABLE* — and writing sargable conditions is the cheapest performance fix there is: same rows, same index, no schema change. This lesson lists the patterns that break sargability, shows the rewrite for each, and explains the rule underneath them all.

The rule

An index is sorted by the column's **stored values**. The server can seek only if it can turn your condition into "find this value, or this range of values, in that order". So:

> Leave the indexed column **bare** on one side of the comparison, and put all the computation on the other side.

`placed_on ⩾ '2024-01-01'` — bare column, constant: sargable. `YEAR(placed_on) = 2024` — the index is not sorted by `YEAR(placed_on)`, so the server must compute it for every row: not sargable.

Functions on the column

```
-- non-sargable: YEAR() on every row, type ALL
SELECT id, amount FROM orders WHERE YEAR(placed_on) = 2024;

-- sargable: a range on the bare column, type range
SELECT id, amount FROM orders
WHERE placed_on ⩾ '2024-01-01' AND placed_on < '2025-01-01';
```

The same shape covers months and days: `MONTH(d) = 3 AND YEAR(d) = 2024` becomes `d ⩾ '2024-03-01' AND d < '2024-04-01'`; `DATE(created_at) = '2024-06-30'` becomes `created_at ⩾ '2024-06-30' AND created_at < '2024-07-01'`.

Prefer the **half-open** range (`>=` start, `<` next start) to `BETWEEN`. On a `DATETIME`, `BETWEEN '2024-01-01' AND '2024-12-31'` means up to `2024-12-31 00:00:00` and silently drops every order placed later that day. The half-open form is correct for `DATE` and `DATETIME` alike and never overlaps the next period.

Other common functions with sargable twins:

Non-sargable	Sargable rewrite
`LEFT(sku, 3) = 'ABC'`	`sku LIKE 'ABC%'`
`SUBSTRING(code, 1, 2) = 'IN'`	`code LIKE 'IN%'`
`DATEDIFF('2024-06-30', d) <= 30`	`d >= DATE_SUB('2024-06-30', INTERVAL 30 DAY)`
`IFNULL(discount, 0) > 0`	`discount > 0` (NULL is never > 0 anyway)
`CONCAT(first, ' ', last) = 'Asha Rao'`	`first = 'Asha' AND last = 'Rao'`

Arithmetic on the column

Move the arithmetic to the constant side, as in algebra:

```sql
WHERE price * 1.18 > 1180          -- non-sargable
WHERE price > 1180 / 1.18          -- sargable: price > 1000

WHERE created_at + INTERVAL 7 DAY > '2024-06-30'      -- non-sargable
WHERE created_at > '2024-06-30' - INTERVAL 7 DAY      -- sargable
```

Watch the sign when you divide by a negative number, and watch the boundary: `DATEDIFF(a, d) <= 30` and `d >= a - 30 days` agree only when `d` is not after `a`. A rewrite must return **the same rows** — test it on the edge values, as the exercises do.

LIKE and wildcards

A B-tree can serve a `LIKE` whose pattern starts with literal characters: `name LIKE 'ravi%'` is a range — every entry from `'ravi'` up to just before `'ravj'`. A **leading wildcard** cannot:

`'%ravi'` and `'%ravi%'` match entries scattered all over the index.

```
WHERE email LIKE 'ravi%'        -- range scan on idx_email
WHERE email LIKE '%@gmail.com'  -- full scan
```

For suffix searches, store the reversed string in an indexed column, or a separate indexed `domain` column. For words inside prose, use a `FULLTEXT` index with `MATCH(body) AGAINST('index')`.

Implicit type conversions

Comparing a column with a value of a **different type** makes MySQL convert, and if it is the column that gets converted, the index is lost:

```
-- phone is VARCHAR(15) with an index
WHERE phone = 9876543210        -- number: every phone converted to a number,
WHERE phone = '9876543210'      -- string: index lookup
```

The same trap hides in joins: joining a `VARCHAR` column to an `INT` column, or two string columns with **different character sets or collations**, can stop the index on the inner table being used. Keep join columns the same type, length, charset and collation.

The opposite direction is harmless: an `INT` column compared with the string `'42'` converts the constant once and still uses the index.

OR, NOT and negative conditions

- `WHERE a = 1 OR b = 2` on two different columns cannot be one seek. MySQL may use `index_merge` (union of two index scans) or scan; rewriting as `UNION` of two sargable queries is a classic fix for a hot query.
- `WHERE a = 1 OR a = 2` is fine — it is `a IN (1, 2)`, a set of seeks.
- `≠`, `NOT IN` and `NOT LIKE` usually match most of the table, so the optimizer scans regardless; that is selectivity, not syntax.
- `IS NULL` **is** sargable in MySQL — the nulls are stored in the index (`ref_or_null` when combined with `= value`).

When the expression cannot move: functional indexes

Sometimes the expression is the real key — case-insensitive matching on a case-sensitive collation, or JSON fields. MySQL 8.0.13+ can index an expression directly:

```
CREATE INDEX idx_users_email_lower ON users ((LOWER(email)));
SELECT id FROM users WHERE LOWER(email) = 'asha@example.com';    -- now sargabl
```

The query must use the **same expression** as the index. Before 8.0.13 the equivalent was a generated column with an index on it. PostgreSQL has had expression indexes for a long time: `CREATE INDEX ON users (lower(email));`.

Sargable is necessary, not sufficient

A sargable predicate lets the optimizer *consider* the index. It still chooses by cost: if the range covers a large share of the table, a scan wins, and that is correct. Sargability is about not taking the choice away from it.

Practise

Both exercises hand you a non-sargable query and ask for a sargable one that returns exactly the same rows — including the edge rows a careless rewrite gets wrong, such as an order at 23:30 on the last day of the year.

EXERCISE 1

Rewrite YEAR() as a range

The yearly report runs `SELECT id, created_at, amount FROM orders WHERE YEAR(created_at) = 2024`, and the index on `created_at` is never used. Rewrite it so the bare column is compared with a range and **exactly the same rows** come back — `created_at` is a `DATETIME`, so watch the last day of the year.

`orders`:

id	created_at	amount
1	2023-12-31 23:59:59	400
2	2024-01-01 00:00:00	150
3	2024-07-14 12:05:00	820
4	2024-12-31 23:30:00	99
5	2025-01-01 00:00:00	310
6	2024-12-31 09:00:00	45

Output columns: `id`, `created_at`, `amount`, ordered by `created_at`, then `id`.

```
id | created_at | amount
2 | 2024-01-01 00:00:00 | 150
3 | 2024-07-14 12:05:00 | 820
6 | 2024-12-31 09:00:00 | 45
4 | 2024-12-31 23:30:00 | 99
```

INPUT 1

```
CREATE TABLE orders (
  id INT PRIMARY KEY,
  created_at DATETIME NOT NULL,
  amount INT NOT NULL
);
INSERT INTO orders VALUES
  (1, '2023-12-31 23:59:59', 400),
  (2, '2024-01-01 00:00:00', 150),
  (3, '2024-07-14 12:05:00', 820),
  (4, '2024-12-31 23:30:00', 99),
  (5, '2025-01-01 00:00:00', 310),
  (6, '2024-12-31 09:00:00', 45);
```

OUTPUT 1

```
id | created_at | amount
2 | 2024-01-01 00:00:00 | 150
3 | 2024-07-14 12:05:00 | 820
6 | 2024-12-31 09:00:00 | 45
4 | 2024-12-31 23:30:00 | 99
```

REFERENCE SOLUTION

```sql
SELECT id, created_at, amount
FROM orders
WHERE created_at ⩾ '2024-01-01'
  AND created_at < '2025-01-01'
ORDER BY created_at, id;
```

EXERCISE 2

A prefix and a date window, sargable

The admin audit runs this query, which can use neither the index on `username` nor the one on `signup_date`:

```
SELECT id, username, signup_date
FROM users
WHERE LEFT(username, 3) = 'adm'
  AND DATEDIFF('2024-06-30', signup_date) ⩽ 30
ORDER BY signup_date, id;
```

Rewrite both predicates so each compares the bare column, returning **the same rows**. No user signed up after 2024-06-30, and usernames are lower case.

`users`:

id	username	signup_date
1	admin	2024-06-20
2	adm_ops	2024-05-31
3	adm_billing	2024-05-30
4	ravi	2024-06-25
5	ad	2024-06-29
6	xadmin	2024-06-10
7	administrator	2024-06-30

Output columns: `id`, `username`, `signup_date`, ordered by `signup_date`, then `id`.

```
id | username | signup_date
2 | adm_ops | 2024-05-31
1 | admin | 2024-06-20
7 | administrator | 2024-06-30
```

INPUT 1

```
CREATE TABLE users (
  id INT PRIMARY KEY,
  username VARCHAR(40) NOT NULL,
  signup_date DATE NOT NULL
);
INSERT INTO users VALUES
  (1, 'admin', '2024-06-20'),
  (2, 'adm_ops', '2024-05-31'),
  (3, 'adm_billing', '2024-05-30'),
  (4, 'ravi', '2024-06-25'),
  (5, 'ad', '2024-06-29'),
  (6, 'xadmin', '2024-06-10'),
  (7, 'administrator', '2024-06-30');
```

OUTPUT 1

```
id | username | signup_date
2 | adm_ops | 2024-05-31
1 | admin | 2024-06-20
7 | administrator | 2024-06-30
```

REFERENCE SOLUTION

```sql
SELECT id, username, signup_date
FROM users
WHERE username LIKE 'adm%'
  AND signup_date ⩾ DATE_SUB('2024-06-30', INTERVAL 30 DAY)
ORDER BY signup_date, id;
```

QUIZ

1. `orders` has an index on `placed_on`. Which predicate can use it to seek?

A. `YEAR(placed_on) = 2024`

B. `DATE_FORMAT(placed_on, '%Y-%m') = '2024-03'`

C. `placed_on ⩾ '2024-03-01' AND placed_on < '2024-04-01'`

D. `placed_on + INTERVAL 1 DAY > '2024-03-01'`

2. `phone` is `VARCHAR(15)` with an index. Why might `WHERE phone = 9876543210` scan the table?

A. Numbers are never allowed in WHERE

B. MySQL converts each stored string to a number to compare, so the index on the strings cannot be searched

C. VARCHAR columns cannot be indexed

D. The number is too large

3. Which `LIKE` pattern can use a B-tree index on `email`?

A. `'%@gmail.com'`

B. `'%ravi%'`

C. `'ravi%'`

D. `'_avi%'`

4. You must keep `WHERE LOWER(email) = ?` (the column's collation is case-sensitive). How can MySQL 8 still seek?

A. It cannot

B. Create a functional index on `((LOWER(email)))`

C. Add `FORCE INDEX`

D. Use `LIKE` instead of `=`

ANSWER KEY · SARGABLE QUERIES — PREDICATES AN INDEX CAN USE

1. C Only the third leaves the column bare. The others compute an expression on every row, and the index is sorted by the stored dates, not by that expression.

2. B Comparing a string column with a number converts the column side, row by row. Write `phone = '9876543210'`. An `INT` column compared with the string `'42'` is harmless: the constant is converted once.

3. C A fixed prefix marks a contiguous range of the sorted index. A leading `%` or `_` puts the unknown part first, so the matches are scattered everywhere.

4. B MySQL 8.0.13+ indexes expressions written in double parentheses; a query using the same expression can then seek. `FORCE INDEX` cannot make a non-sargable predicate searchable — at best it forces a full index scan.

LESSON 5 · 14 MIN

Covering indexes and index-only reads

The last lesson made the server *able* to use an index. This one makes it use the index **instead of** the table. Recall from the first lesson that an InnoDB secondary index lookup is two searches: down the secondary index to find primary keys, then down the clustered index once per key to fetch the row. If the index already holds every column the query needs, the second search is pointless — the server can answer from the index alone. Such an index **covers** the query, and the plan says so with `Using index`. This is one of the highest-leverage tricks in query tuning, and a favourite follow-up question after "what is a clustered index?".

Two lookups, or one

```
CREATE TABLE orders (
  id          INT PRIMARY KEY,
  customer_id INT NOT NULL,
  placed_on   DATE NOT NULL,
  status      VARCHAR(12) NOT NULL,
  amount      INT NOT NULL,
  notes       TEXT
);
CREATE INDEX idx_orders_customer ON orders (customer_id);
```

```
SELECT placed_on, amount FROM orders WHERE customer_id = 7;
```

With `idx_orders_customer`, the server finds customer 7's entries — each holding `customer_id` and `id` — and then, for **each** one, searches the clustered index to read `placed_on` and `amount`. Forty orders, forty extra searches, landing on pages scattered around the table.

Now extend the index:

```
CREATE INDEX idx_orders_cust_date_amt ON orders (customer_id, placed_on, amour
```

Every column the query touches — `customer_id` in the `WHERE`, `placed_on` and `amount` in the `SELECT` — is in the index. The server reads customer 7's entries, which sit next to each other, and is done:

table	type	key	rows	Extra
orders	ref	idx_orders_cust_date_amt	40	Using index

The primary key comes along for free

Because InnoDB secondary entries store the primary key, the key is **always covered**:

```
CREATE INDEX idx_users_email ON users (email);

SELECT id FROM users WHERE email = 'asha@example.com';          -- covered
SELECT id, email FROM users WHERE email LIKE 'asha%';           -- covered
SELECT id, email, name FROM users WHERE email = 'asha@…';       -- not: name
```

That also explains why `COUNT(*)` on an InnoDB table often scans the **smallest secondary index** rather than the table: every index has one entry per row, and a narrow index has the fewest pages.

What "every column" means

A query is covered only if the index contains the columns used **anywhere** in it for that table:

- the `SELECT` list (so `SELECT *` defeats it — list the columns you need);
- the `WHERE` clause;
- `JOIN … ON` conditions;
- `GROUP BY`, `ORDER BY`, `HAVING`, and the arguments of aggregates.

```
-- index (customer_id, placed_on, amount)
SELECT customer_id, COUNT(*) AS orders, SUM(amount) AS total
FROM orders
WHERE placed_on ⩾ '2024-01-01'
GROUP BY customer_id;
```

Every column is in the index, so the query is covered. Note that `placed_on` is the second column and the filter has no `customer_id` equality, so this is not a seek: the plan is `type: index` — a full scan of the index — but still `Using index`. Scanning a compact index is much cheaper than scanning the table, especially when the table has wide columns like `notes`. If

the date filter is the common case, an index starting with `placed_on` would make it a range instead.

Covering helps sorting too

When the index order matches the `ORDER BY` *and* covers the columns, the server streams rows in order straight from the leaves — no lookups and no filesort:

```
-- index (customer_id, placed_on, amount)
SELECT placed_on, amount
FROM orders
WHERE customer_id = 7
ORDER BY placed_on DESC
LIMIT 5;
```

With `LIMIT 5`, it reads five index entries and stops. This "latest N for one key" pattern — a user's recent orders, a thread's newest posts — is exactly where a covering composite index shines.

Deferred joins: covering the expensive part

When the final rows need wide columns, a covering index can still do the hard work of choosing *which* rows:

```
SELECT o.*
FROM orders o
JOIN (
  SELECT id FROM orders
  WHERE customer_id = 7
  ORDER BY placed_on DESC
  LIMIT 20 OFFSET 2000
) page ON page.id = o.id
ORDER BY o.placed_on DESC;
```

The inner query is covered by `(customer_id, placed_on)` (plus the implicit `id`), skips 2,000 entries in the index cheaply, and only 20 full rows are fetched. This **deferred join** is the standard fix for deep `OFFSET` pagination when keyset pagination is not possible.

The trade-offs

A covering index is a wider index, with all the costs of the first lesson multiplied:

- **bigger** — more pages, more memory, more I/O on writes;
- **more write work** — updating any of its columns moves the entry, so covering a frequently updated column is expensive;
- **limits** — at most 16 columns and a key-size cap; `TEXT` / `BLOB` columns can only be prefix-indexed, and a prefix never covers.

Cover the few hot queries that matter, not every query. MySQL has no `INCLUDE` clause: where PostgreSQL 11+ writes `CREATE INDEX … (customer_id) INCLUDE (amount)`, MySQL appends `amount` to the key. PostgreSQL also needs its visibility map to be current for an `Index Only Scan` to skip the heap — one reason `VACUUM` matters there.

Practise

The first exercise builds a covering index and runs the per-customer report it serves. The second decides coverage in SQL: given each index's columns and the columns each query touches, list which indexes cover each query — remembering that InnoDB adds the primary key to every secondary index.

EXERCISE 1

Cover the yearly customer report

The finance dashboard sums each customer's 2024 orders. Create the covering index `idx_orders_cover` on `orders (customer_id, placed_on, amount)` — every column the report touches — then write the report with a sargable date range.

`orders`:

id	customer_id	placed_on	status	amount	notes
1	4	2024-01-09	paid	300	gift wrap
2	2	2024-02-11	paid	1200	NULL
3	4	2024-03-30	refunded	450	damaged box
4	9	2023-12-31	paid	5000	NULL
5	2	2024-05-05	paid	300	NULL
6	9	2024-06-18	paid	750	call first
7	4	2025-01-02	paid	80	NULL

Output columns: `customer_id`, `orders` (the count), `total` (the sum of `amount`) — 2024 orders only, every status — ordered by `total` descending, then `customer_id`.

```
customer_id | orders | total
2 | 2 | 1500
4 | 2 | 750
9 | 1 | 750
```

INPUT 1

```
CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT NOT NULL,
  placed_on DATE NOT NULL,
  status VARCHAR(12) NOT NULL,
  amount INT NOT NULL,
  notes TEXT
);
INSERT INTO orders VALUES
  (1, 4, '2024-01-09', 'paid', 300,
'gift wrap'),
  (2, 2, '2024-02-11', 'paid', 1200,
NULL),
  (3, 4, '2024-03-30', 'refunded', 450,
'damaged box'),
  (4, 9, '2023-12-31', 'paid', 5000,
NULL),
  (5, 2, '2024-05-05', 'paid', 300,
NULL),
  (6, 9, '2024-06-18', 'paid', 750,
'call first'),
  (7, 4, '2025-01-02', 'paid', 80,
NULL);
```

OUTPUT 1

```
customer_id | orders | total
2 | 2 | 1500
4 | 2 | 750
9 | 1 | 750
```

REFERENCE SOLUTION

```sql
CREATE INDEX idx_orders_cover ON orders (customer_id, placed_on, amount);

SELECT customer_id,
       COUNT(*) AS orders,
       SUM(amount) AS total
FROM orders
WHERE placed_on ⩾ '2024-01-01'
  AND placed_on < '2025-01-01'
GROUP BY customer_id
ORDER BY total DESC, customer_id;
```

EXERCISE 2

Which indexes cover each query?

`index_columns` lists the columns of each secondary index on `users`, whose primary key is `id`. `query_columns` lists every column each query touches. An index covers a query when every column the query touches is in the index — and, because InnoDB stores the primary key in every secondary index, `id` is always covered. List each query with every index that covers it, or `'none'`.

`index_columns`:

index_name	seq	column_name
idx_email	1	email
idx_city_signup	1	city
idx_city_signup	2	signup_date
idx_city_signup_name	1	city
idx_city_signup_name	2	signup_date
idx_city_signup_name	3	name

`query_columns`:

query_id	column_name
1	id
1	email
2	city
2	signup_date
3	id
3	name
3	city
4	email
4	name

Output columns: `query_id`, `covered_by` — one row per covering index, or a single `'none'` row — ordered by `query_id`, then `covered_by`.

```
query_id | covered_by
1 | idx_email
2 | idx_city_signup
2 | idx_city_signup_name
3 | idx_city_signup_name
4 | none
```

INPUT 1

```
CREATE TABLE index_columns (index_name
VARCHAR(40), seq INT, column_name
VARCHAR(40));
INSERT INTO index_columns VALUES
  ('idx_email', 1, 'email'),
  ('idx_city_signup', 1, 'city'),
('idx_city_signup', 2, 'signup_date'),
  ('idx_city_signup_name', 1, 'city'),
('idx_city_signup_name', 2,
'signup_date'), ('idx_city_signup_name',
3, 'name');
CREATE TABLE query_columns (query_id
INT, column_name VARCHAR(40));
INSERT INTO query_columns VALUES
  (1, 'id'), (1, 'email'), (2, 'city'),
(2, 'signup_date'),
  (3, 'id'), (3, 'name'), (3, 'city'),
(4, 'email'), (4, 'name');
```

OUTPUT 1

```
query_id | covered_by
1 | idx_email
2 | idx_city_signup
2 | idx_city_signup_name
3 | idx_city_signup_name
4 | none
```

REFERENCE SOLUTION

```sql
WITH queries AS (
  SELECT DISTINCT query_id FROM query_columns
),
covering AS (
  SELECT q.query_id, i.index_name
  FROM queries q
  CROSS JOIN (SELECT DISTINCT index_name FROM index_columns) AS i
  WHERE NOT EXISTS (
    SELECT 1
    FROM query_columns c
    WHERE c.query_id = q.query_id
      AND c.column_name <> 'id'
      AND c.column_name NOT IN (SELECT ic.column_name
                                FROM index_columns ic
                                WHERE ic.index_name = i.index_name)
  )
)
SELECT q.query_id, COALESCE(cv.index_name, 'none') AS covered_by
FROM queries q
LEFT JOIN covering cv ON cv.query_id = q.query_id
ORDER BY q.query_id, covered_by;
```

QUIZ

1. Index `idx_email` on `users (email)`, primary key `id`. Which query is covered?

A. `SELECT * FROM users WHERE email = ?`

B. `SELECT id, email FROM users WHERE email = ?`

C. `SELECT name FROM users WHERE email = ?`

D. `SELECT email, city FROM users WHERE email = ?`

2. What makes a covering index faster than an ordinary secondary index lookup?

A. It is stored in memory only

B. It skips the per-row search in the clustered index and reads compact, adjacent entries

C. It is a hash index

D. It locks fewer rows

3. Why does `SELECT COUNT(*) FROM orders` in InnoDB often read a secondary index rather than the table?

A. Secondary indexes store a precomputed count

B. Every index has one entry per row, and the narrowest one has the fewest pages to read

C. The clustered index cannot be scanned

D. It is a bug

4. How do you get the effect of PostgreSQL's `INCLUDE (amount)` in MySQL?

A. `CREATE INDEX … INCLUDE (amount)` works in MySQL 8 too

B. Append `amount` to the end of the index key

C. Create a second index on `amount`

D. It is impossible in MySQL

ANSWER KEY · COVERING INDEXES AND INDEX-ONLY READS

1. B InnoDB secondary entries store the primary key, so `id` and `email` are both in `idx_email`. `name`, `city` and `*` need the row from the clustered index.

2. B A non-covering lookup searches the clustered index once per match, landing on scattered pages. A covering read stays in the index, whose entries are smaller than rows and sit together.

3. B Counting needs one entry per row, not the rows' contents, so the optimizer scans the smallest index. InnoDB keeps no exact stored row count, because each transaction may see a different number of rows under MVCC.

4. B MySQL has no `INCLUDE` clause. Adding the column as the last key column carries it in the leaves, at the cost of making it part of the sort order and the key size limit.

CHECKPOINT · 25 MIN

Checkpoint — Indexes and performance

This checkpoint covers the whole module: what an index is and why it is a B+tree, InnoDB's clustered primary key and the secondary indexes that point to it, selectivity and the cost of every index, composite indexes and the leftmost prefix rule, how a range stops a seek, reading MySQL's `EXPLAIN` and `EXPLAIN ANALYZE`, sargable predicates and their rewrites, and covering indexes.

How it works. Twelve questions and three SQL tasks. You need 70% on the questions and every task accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- Why is a B+tree three or four levels deep for millions of rows, and what do its linked leaves make cheap?
- What does an InnoDB secondary index store as its pointer to the row, and what follows from that?
- Which of `WHERE a = 1`, `WHERE b = 2` and `WHERE a = 1 AND c = 3` can seek on an index on `(a, b, c)`, and how far?
- Why do equality columns go before the range column in a composite index?
- Put `ALL`, `const`, `range`, `ref`, `index` and `eq_ref` in order from best to worst.
- What do `Using index`, `Using index condition`, `Using filesort` and `Using temporary` mean?
- Rewrite `WHERE YEAR(d) = 2024` and `WHERE LEFT(code, 2) = 'IN'` so an index can be used.
- Why does `SELECT *` almost never use a covering index?

The three tasks are practical maintenance work: remove the duplicate rows that block a `UNIQUE` index, create it and show what remains; build an index for a monthly payments report and write the report with a sargable date range; and find the indexes a usage log says are never read, the ones a DBA would consider dropping.

EXERCISE 1

Clear the way for a UNIQUE index

Sign-up allowed duplicate e-mail addresses until today, and `CREATE UNIQUE INDEX` fails while they exist. Write a script that:

1. lists the duplicated addresses — `email`, `copies` — ordered by `email`;
2. deletes every duplicate row except the one with the **lowest** `id` per address;
3. creates the unique index `uq_users_email` on `users (email)`;
4. returns the remaining users — `id`, `email` — ordered by `id` (this SELECT is in the starter).

MySQL refuses a `DELETE` whose subquery reads the table being deleted from (error 1093) unless the subquery is wrapped in a derived table — write it that way.

`users`:

id	email	name
1	asha@example.com	Asha
2	ravi@example.com	Ravi
3	asha@example.com	Asha R
4	lee@example.com	Lee
5	ravi@example.com	Ravi K
6	asha@example.com	A. Rao

```
email | copies
asha@example.com | 3
ravi@example.com | 2

id | email
1 | asha@example.com
2 | ravi@example.com
4 | lee@example.com
```

INPUT 1

```
CREATE TABLE users (id INT PRIMARY KEY,
email VARCHAR(120) NOT NULL, name
VARCHAR(60));
INSERT INTO users VALUES
  (1, 'asha@example.com', 'Asha'),
  (2, 'ravi@example.com', 'Ravi'),
  (3, 'asha@example.com', 'Asha R'),
  (4, 'lee@example.com', 'Lee'),
  (5, 'ravi@example.com', 'Ravi K'),
  (6, 'asha@example.com', 'A. Rao');
```

OUTPUT 1

```
email | copies
asha@example.com | 3
ravi@example.com | 2

id | email
1 | asha@example.com
2 | ravi@example.com
4 | lee@example.com
```

REFERENCE SOLUTION

```sql
SELECT email, COUNT(*) AS copies
FROM users
GROUP BY email
HAVING COUNT(*) > 1
ORDER BY email;

DELETE FROM users
WHERE id NOT IN (
  SELECT keep_id
  FROM (SELECT MIN(id) AS keep_id FROM users GROUP BY email) AS keepers
);

CREATE UNIQUE INDEX uq_users_email ON users (email);

SELECT id, email FROM users ORDER BY id;
```

EXERCISE 2

An indexed monthly payments report

Create an index named `idx_payments_paid_at` on `payments (paid_at)`, then report **April 2024** by payment method with a sargable range on `paid_at` (a `DATETIME`).

`payments`:

id	paid_at	method	amount
1	2024-03-31 23:59:59	upi	500
2	2024-04-01 00:00:00	upi	250
3	2024-04-12 14:20:00	card	1800
4	2024-04-30 22:15:00	upi	900
5	2024-05-01 00:00:00	card	700
6	2024-04-18 09:00:00	netbanking	1150
7	2024-04-18 09:05:00	card	200

Output columns: `method`, `payments` (count), `total` (sum of `amount`), ordered by `total` descending, then `method`.

```
method | payments | total
card | 2 | 2000
netbanking | 1 | 1150
upi | 2 | 1150
```

INPUT 1

```
CREATE TABLE payments (id INT PRIMARY
KEY, paid_at DATETIME NOT NULL, method
VARCHAR(12) NOT NULL, amount INT NOT
NULL);
INSERT INTO payments VALUES
  (1, '2024-03-31 23:59:59', 'upi',
500),
  (2, '2024-04-01 00:00:00', 'upi',
250),
  (3, '2024-04-12 14:20:00', 'card',
1800),
  (4, '2024-04-30 22:15:00', 'upi',
900),
  (5, '2024-05-01 00:00:00', 'card',
700),
  (6, '2024-04-18 09:00:00',
'netbanking', 1150),
  (7, '2024-04-18 09:05:00', 'card',
200);
```

OUTPUT 1

```
method | payments | total
card | 2 | 2000
netbanking | 1 | 1150
upi | 2 | 1150
```

REFERENCE SOLUTION

```sql
CREATE INDEX idx_payments_paid_at ON payments (paid_at);

SELECT method,
       COUNT(*) AS payments,
       SUM(amount) AS total
FROM payments
WHERE paid_at ⩾ '2024-04-01'
  AND paid_at < '2024-05-01'
GROUP BY method
ORDER BY total DESC, method;
```

EXERCISE 3

Indexes nobody reads

Before a release, the DBA wants the indexes that cost writes and serve no reads. `table_indexes` lists every index; `index_usage` holds the read counters gathered since the last restart (an index absent from it was never read). Return every index that was never read (no usage row, or `read_count` 0) **and** is neither the `PRIMARY` key nor unique — those enforce constraints and stay.

`table_indexes`:

table_name	index_name	is_unique
orders	PRIMARY	1
orders	idx_orders_customer	0
orders	idx_orders_status	0
orders	idx_orders_customer_old	0
users	PRIMARY	1
users	uq_users_email	1
users	idx_users_city	0

`index_usage`:

table_name	index_name	read_count
orders	PRIMARY	98000
orders	idx_orders_customer	4300
orders	idx_orders_status	0
users	idx_users_city	12

Output columns: `table_name`, `index_name`, ordered by `table_name`, then `index_name`.

```
table_name | index_name
orders | idx_orders_customer_old
orders | idx_orders_status
```

INPUT 1

```
CREATE TABLE table_indexes (table_name
VARCHAR(30), index_name VARCHAR(40),
is_unique INT);
INSERT INTO table_indexes VALUES
  ('orders', 'PRIMARY', 1), ('orders',
'idx_orders_customer', 0), ('orders',
'idx_orders_status', 0),
  ('orders', 'idx_orders_customer_old',
0), ('users', 'PRIMARY', 1), ('users',
'uq_users_email', 1),
  ('users', 'idx_users_city', 0);
CREATE TABLE index_usage (table_name
VARCHAR(30), index_name VARCHAR(40),
read_count INT);
INSERT INTO index_usage VALUES
  ('orders', 'PRIMARY', 98000),
('orders', 'idx_orders_customer', 4300),
  ('orders', 'idx_orders_status', 0),
('users', 'idx_users_city', 12);
```

OUTPUT 1

```
table_name | index_name
orders | idx_orders_customer_old
orders | idx_orders_status
```

REFERENCE SOLUTION

```sql
SELECT i.table_name, i.index_name
FROM table_indexes i
LEFT JOIN index_usage u
  ON u.table_name = i.table_name
 AND u.index_name = i.index_name
WHERE i.index_name <> 'PRIMARY'
  AND i.is_unique = 0
  AND COALESCE(u.read_count, 0) = 0
ORDER BY i.table_name, i.index_name;
```

CHECKPOINT QUESTIONS

1. How many clustered indexes can an InnoDB table have?

A. None — InnoDB tables are heaps

B. Exactly one, on the primary key (or its substitute)

C. One per unique column

D. Up to 64

2. A table is keyed by a random UUID string. What is the main cost compared with an `INT AUTO_INCREMENT` key?

A. UUIDs cannot be indexed

B. Inserts land all over the clustered index, splitting pages, and every secondary index repeats the long key

C. Queries return rows in random order

D. Nothing — UUIDs are faster

3. Index `(country, city)`. Which query can seek on it?

A. `WHERE city = 'Pune'`

B. `WHERE country = 'IN'`

C. `WHERE city LIKE 'P%'`

D. `WHERE UPPER(country) = 'IN'`

4. Which composite index serves `WHERE user_id = ? AND status = ? AND created_at ⩾ ?` best?

A. `(created_at, user_id, status)`

B. `(user_id, created_at, status)`

C. `(user_id, status, created_at)`

D. `(status, created_at)`

5. In `EXPLAIN`, `type: index` means…

A. An index lookup for one value

B. A full scan of an index, start to end

C. The query uses an index hint

D. A unique index was used

6. In a two-table nested-loop plan, step 1 shows `rows: 200` and step 2 shows `rows: 50`. Roughly how many rows does the join examine in step 2?

A. 50

B. 250

C. About 10,000 — 50 per row of step 1

D. 200

7. Which `Extra` note is a reason to look for a better index?

A. `Using index`

B. `Using index condition`

C. `Using temporary; Using filesort`

D. `Backward index scan`

8. Which predicate is **not** sargable for an index on `created_at` (a `DATETIME`)?

A. `created_at ⩾ '2024-06-01'`

B. `created_at < '2024-07-01'`

C. `DATE(created_at) = '2024-06-30'`

D. `created_at ⩾ '2024-06-30' AND created_at < '2024-07-01'`

9. Why may MySQL ignore an index on `status` for `WHERE status = 'delivered'`?

A. Indexes are ignored on VARCHAR columns

B. 90% of rows are delivered, so a full scan is cheaper than a lookup per row

C. The index must be rebuilt first

D. Equality is not sargable

10. Index `(customer_id, placed_on)` on `orders` with primary key `id`. Is `SELECT id, placed_on FROM orders WHERE customer_id = 7` covered?

A. No — `id` is not in the index

B. Yes — `id` is carried in every secondary entry

C. Only if `id` is added to the index

D. Only with `FORCE INDEX`

11. What does `CREATE UNIQUE INDEX uq ON users (email)` do when two rows share an email?

A. Keeps the first row and deletes the rest

B. Fails with a duplicate-entry error; the duplicates must be removed first

C. Creates the index and ignores duplicates

D. Creates a non-unique index instead

12. Which tool makes an index invisible to the optimizer while still maintaining it, so you can test dropping it?

A. `DROP INDEX … TEMPORARY`

B. `ALTER TABLE t ALTER INDEX idx INVISIBLE`

C. `IGNORE INDEX` in every query

D. `ANALYZE TABLE`

ANSWER KEY · CHECKPOINT — INDEXES AND PERFORMANCE

1. B The rows themselves are stored in the clustered index, so they can be ordered only one way. Without a primary key InnoDB clusters on the first unique NOT NULL index, or a hidden row id.

2. B An increasing key appends at the right edge of the B+tree; a random key inserts into the middle and splits pages. And because each secondary entry carries the primary key, a 36-character key inflates every index.

3. B Only a filter on the leading column `country` follows the leftmost prefix. `city` alone is not a prefix, and wrapping `country` in a function hides it.

4. C Equality columns first, the range last: `(user_id, status, created_at)` seeks on all three. In the second option the range on `created_at` stops `status` from narrowing the seek.

5. B `index` reads the whole index — cheaper than `ALL` when the index is narrower than the table, and often paired with `Using index` when it also covers the query, but still a full scan.

6. C `rows` is per lookup. Step 2 runs once for each of step 1's 200 rows, so about 200 × 50 rows are examined (before `filtered` is applied).

7. C A temporary table plus a sort means grouping or ordering is done the expensive way. The other three are signs an index is being used well.

8. C `DATE()` is computed on every row. The last option returns the same rows as a range on the bare column.

9. B The optimizer chooses by cost. When a value matches most of the table, the per-row lookups into the clustered index cost more than reading the table in order.

10. B InnoDB secondary indexes store the primary key, so `customer_id`, `placed_on` and `id` are all in the index and the plan shows `Using index`.

11. B A unique index is a constraint, and building it checks the existing rows. Find duplicates with `GROUP BY email HAVING COUNT(*) > 1` and resolve them first.

12. B MySQL 8 invisible indexes keep the index up to date but hidden from plans; if nothing slows down, drop it, otherwise make it `VISIBLE` again instantly. `IGNORE INDEX` affects only the queries that carry the hint.
