

Indexes — Database Ki Firasat (Speed Ka Raaz)

Why big tables answer slowly and how an index fixes it: CREATE INDEX, reading EXPLAIN, and the honest cost every index charges.

Introduction

Our `students` table has five rows; every query feels instant. Grow it to a **million** students and a simple question — *who lives in Sahiwal?* — turns slow. Without help, the database answers by reading the table from the first row to the last, checking every single one. That full read is called a **table scan**, and it is the database equivalent of reading a whole book to find one name.

A book solves this with an **index** at the back: names in order, with page numbers. A database index is the same idea — a separate, sorted lookup structure the database maintains, so a search jumps straight to the few rows that match. This lesson watches a query before and after its index, learns to read the database's own report card (EXPLAIN), and ends with the honest price every index charges.

(Roman Urdu: *Index = kitab ki fehris. Poori table parhne ke bajaye, seedha sahi row par chhalang.*)

Key points

- **An index is a sorted copy of key values + row pointers:** `CREATE INDEX idx_city ON students(city);` builds a lookup ordered by city. Searches by city now jump; the table itself is unchanged.
- **Indexes speed reads, they do not change results:** with or without an index, `WHERE city = 'Sahiwal'` returns the same rows. Only the *time* changes. An index can never make an answer wrong — just fast or slow.
- **EXPLAIN shows the plan:** put `EXPLAIN` before any SELECT and the database reports *how* it will answer — before running it. Two columns matter first: **type** (`ALL` = full table scan, slow; `ref` / `range` / `const` = index used) and **rows** (how many rows it expects to examine).
(Roman Urdu: *EXPLAIN database ka report card hai — query chalane se pehle pata chal jata hai ke raasta sust hai ya tez.*)
- **The primary key is already an index:** lookups by `id` were fast from day one — that is why DB.6's key matters beyond identity. You add *extra* indexes for the other columns you search and join by: cities, emails, foreign keys.
- **Every index charges rent:** each index must be updated on every INSERT, UPDATE and DELETE, and it takes storage. An index on a column nobody searches is pure cost. Index the columns your real questions use — no more.
- **Leftmost rule of thumb:** an index on `(city, class)` helps searches on city alone, or city + class together — like a phone book sorted by surname, then first name. It does *not* help a search on class alone.

Code examples

Imagine `students` has grown to **1,000,000 rows** for these examples (same columns; Sahiwal still has only a handful of students).

Example 1 — The slow search, and its report card

```
EXPLAIN SELECT * FROM students WHERE city = 'Sahiwal';
```

What it does: Before running the search, we ask how it *would* run. There is no index on `city`, so the plan is a full scan.

What you will see:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | students | ALL | NULL | NULL | NULL | NULL | 1000000 | Using where |
```

Two cells tell the story: **type = ALL** (every row will be read) and **rows = 1000000** (a million rows examined to find a handful). On a real server this query takes seconds, not milliseconds — and it gets worse with every new student.

Example 2 — CREATE INDEX, then read the new report card

```
CREATE INDEX idx_city ON students (city);

EXPLAIN SELECT * FROM students WHERE city = 'Sahiwal';

SELECT name, city FROM students WHERE city = 'Sahiwal';
```

What it does: One command builds the sorted lookup for `city`. The same EXPLAIN now reports a completely different plan, and the real search returns exactly the rows it always did — Ali and Hira in our original five, a handful in the million.

What you will see:

```

+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| 1 | SIMPLE | students | ref | idx_city | idx_city | 202 | const | 2 |
Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+

+-----+-----+
| name | city |
+-----+-----+
| Ali Raza | Sahiwal |
| Hira Shah | Sahiwal |
+-----+-----+
2 rows in set

```

type = ref and **rows = 2**: the database now examines about two rows instead of a million. Same question, same answer — the difference is the road taken. This before/after EXPLAIN is how professionals prove a slowness is fixed, not just hope it is.

Example 3 — The rent: what your indexes cost

```
SHOW INDEX FROM students;
```

What it does: Lists every index the table carries. After DB.6's primary key and Example 2's city index, our table has exactly two.

What you will see:

```

+-----+-----+-----+-----+-----+-----+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Cardinality |
+-----+-----+-----+-----+-----+-----+
| students | 0 | PRIMARY | 1 | id | 999998 |
| students | 1 | idx_city | 1 | city | 120 |
+-----+-----+-----+-----+-----+-----+

```

Each of these structures is re-sorted and updated on **every write**. Two well-chosen indexes are cheap rent for fast searches. Twenty hopeful indexes — on columns nobody queries — make every INSERT pay twenty tolls. SHOW INDEX is your audit: if an index cannot name the real question it serves, it is a candidate for removal (`DROP INDEX idx_city ON students;`).

Quick recap

- Without an index, a search is a **table scan** — every row read (EXPLAIN: type ALL, rows = table size).
- **CREATE INDEX** builds a sorted lookup: searches jump (type ref, rows ≈ matches), results never change.
- **EXPLAIN** before a slow query is the professional habit — measure, index, re-measure.
- Indexes charge rent on every write and in storage: index the columns real questions use — foreign keys, search and sort columns — and audit with SHOW INDEX.

Common mistakes

- **Indexing everything "for speed"**: each extra index slows every INSERT/UPDATE/DELETE and eats storage. Ten indexes do not make a table ten times faster; they make writes ten times heavier.
- **Judging by the small table**: with five rows, everything is instant, so beginners conclude indexes are pointless. Judge at realistic size — or trust the EXPLAIN rows column, which tells the truth at any size.
(Roman Urdu: Chhoti table par har cheez tez lagti hai — EXPLAIN ka rows wala khana asli kahani sunata hai.)
- **A missing index on the foreign key**: joins and child lookups search the linking column constantly. (MySQL creates this index automatically for InnoDB foreign keys — but on tables without the declared key, DB.6's lesson, you must add it yourself.)
- **Index on the wrong column order**: an index on (class, city) does not speed a search by city alone — the sort is class-first. Match the index's leading column to how the questions actually arrive.

Practice

1. Your products table (100,000 rows) is searched by name every day and by category every day, and never by supplier_phone. Write the CREATE INDEX commands you would add — and name the one column you refuse to index, with the reason.
2. Read this EXPLAIN line and write two sentences: what will the database do, and what will you change?
type: ALL, key: NULL, rows: 450000 for the query SELECT * FROM orders WHERE customer_id = 42.
3. Explain the rent in your own words: a table gets 10,000 new rows a day and has 6 indexes. What extra work happens on every single insert, and when is that price worth paying?

Watch the Code Scope channel: <https://www.youtube.com/@CodeScopePK> — this lesson's own video joins the channel when it is recorded.

Read online: <https://toolscope.netlify.app/notes/sql-indexes> · More free tools and notes at <https://toolscope.netlify.app>