16 SQL Indexes

SQL Indexes: A Comprehensive Guide (MySQL/InnoDB Focus)

Indexes are critical for efficient data retrieval, but their design and application require careful consideration to maximize benefits and avoid common pitfalls.

Why Indexes Matter

What is an Index?

An **index** is a specialized data structure that accelerates data retrieval by minimizing disk **Input/Output (IO)** operations. Think of it as the index in a book: instead of flipping through every page to find a topic, you consult the index to locate the exact page instantly.

- **Structure:** An index is separate from the table's data (stored in the **Heap**) and contains a subset of columns (the indexed columns) along with **pointers** to the actual data's location in the Heap.
- **Underlying Mechanism:** In MySQL's InnoDB, indexes typically use a **B+Tree** data structure, optimized for fast lookups, range queries, and sequential access due to its balanced tree structure and linked leaf nodes.

The Cost of Searching Without Indexes

Without an index, MySQL must perform a **Full Heap Scan**, reading every page of the table's data to find the desired rows. This is computationally expensive due to the high cost of disk IO operations.

- **Heap:** The Heap is where table data is stored, organized into fixed-size **pages** (e.g., 16KB in InnoDB).
- **IO Operation:** Each IO operation reads one or more pages from disk, a slow process compared to CPU operations.
- **Example:** For a query like `SELECT * FROM employees WHERE emp_id = 10000;`, without an index, MySQL may need to scan hundreds of pages (e.g., 333 pages), resulting in significant IO overhead.

Key Benefit: Indexes reduce IO operations by pointing directly to the relevant data pages, avoiding full scans.

How Indexes Work

Consider the query `SELECT * FROM employees WHERE emp_id = 10000;`. With an index on `emp_id`, the process is:

1. **Index Lookup (1 IO):** MySQL searches the B+Tree index for `emp_id = 10000`. The index quickly identifies the entry and its pointer to the data's location (e.g., page 333 in the Heap).

2. **Heap Fetch (1 IO):** MySQL performs a targeted IO to retrieve the full row from page 333.

Performance Impact:

- **Without Index:** ~333 IOs (scanning every page).
- **With Index:** ~2 IOs (index lookup + heap fetch).

Note: If the index is **covering** (contains all columns needed by the query), the heap fetch may be skipped, further reducing IOs.

Clustered vs. Non-Clustered Indexes

MySQL's InnoDB uses two types of indexes: **clustered** and **non-clustered (secondary)**. Their differences are critical for understanding performance.

Feature	Clustered Index	Non-Clustered Index (Secondary)
Data Storage	Stores the entire table data, sorted by the index key.	Stores indexed column(s) + a pointer to the data (usually the PRIMARY KEY).
Pointers	Leaf nodes contain all columns of the row.	Leaf nodes contain the indexed column(s) and a reference to the PRIMARY KEY.
Quantity	Only **one** per table (data can only be sorted one way).	Multiple per table.
InnoDB Role	The **PRIMARY KEY** is the clustered index. If no PRIMARY KEY exists, InnoDB uses the first unique non-nullable index or generates an internal `GEN_CLUST_INDEX`.	All other indexes are non-clustered.
Performance	Fastest for lookups and range queries, as data is physically stored with the index.	Requires a **bookmark lookup** to fetch additional columns via the clustered index, adding an extra step.

Bookmark Lookup: When querying a non-clustered index, MySQL retrieves the PRIMARY KEY from the index, then uses it to fetch the full row from the clustered index. This extra step can impact performance if many columns are needed.

Optimizing Index Design

Integer vs. String Columns

The choice of column type affects index efficiency:

Column Type	Advantages	Disadvantages
Integer (e.g., `emp_id`)	Small size, faster searches, more entries per page, fewer IOs. Ideal for PRIMARY KEYs and frequent lookups.	None significant.
String (e.g., `emp_name`)	Useful for name-based searches.	Larger index size, slower searches, and higher memory usage. Prefix indexes (e.g., `INDEX (name(10))`) reduce size but may cause collisions if the prefix is too short.

Tip: Use prefix indexes for strings (e.g., `CREATE INDEX idx_name ON employees (name(10));`) to save space, but ensure the prefix length maintains selectivity.

Composite Indexes

A **composite index** includes multiple columns, e.g., `CREATE INDEX idx_id_name ON employees (emp_id, name);`.

- **Sorting:** The index is sorted first by `emp_id`, then by `name` for rows with the same `emp_id`.
- **Leftmost Prefix Rule:** MySQL can use the index efficiently for queries filtering on:
  - `emp_id` alone.
  - Both `emp_id` and `name`.
- **Limitation:** Queries filtering only on `name` cannot use this index efficiently unless MySQL performs a full index scan.
- **Key Insight:** The order of columns in a composite index matters. Place frequently filtered columns first.

Index Usage and Common Pitfalls

The `LIKE` Operator

The `LIKE` operator's pattern determines whether an index is used:

Pattern	Index Usage	Explanation
`WHERE name LIKE 'E%'`	Index used.	The constant prefix ('E') allows MySQL to locate the starting point in the index and scan forward.
`WHERE name LIKE '%e'`	Index not used (Full Heap Scan).	A leading wildcard prevents MySQL from identifying a starting point in the index.
`WHERE name LIKE '%ed%'`	Index not used (Full Heap Scan).	Leading wildcards require scanning all rows.

Index Blockers

Indexes are most effective with equality (`=`), range (`>`, `<`, `BETWEEN`), and certain `LIKE` patterns. The following prevent index usage:

- **Functions on Columns:** `WHERE YEAR(emp_dob) = 2000;`
  - **Problem:** MySQL must apply `YEAR()` to every row, bypassing the index on `emp_dob`.
  - **Solution:** Rewrite as `WHERE emp_dob BETWEEN '2000-01-01' AND '2000-12-31';` to use the index.
- **Calculations on Columns:** `WHERE emp_salary * 1.1 > 100000;`
  - **Problem:** The calculation prevents index usage on `emp_salary`.
  - **Solution:** Rewrite as `WHERE emp_salary > 100000 / 1.1;` to enable index usage.
- **Negations:** `WHERE emp_id ≠ 10000;` or `WHERE emp_id NOT IN (10000, 20000);`
  - **Problem:** Negations often lead to full scans or inefficient index usage.

Practical Tip: Use MySQL's `EXPLAIN` or `EXPLAIN ANALYZE` to verify whether an index is used and identify potential full scans.

Key Takeaways

- **Indexes** reduce IO operations by pointing directly to data pages, avoiding costly Full Heap Scans.
- **Clustered Indexes** (PRIMARY KEY in InnoDB) store table data; **non-clustered indexes** require a bookmark lookup.
- **Integer columns** are ideal for indexing due to their compact size and speed.
- **Composite indexes** must align with query patterns, respecting the leftmost prefix rule.
- **Avoid index blockers** like functions, calculations, or leading wildcards in `LIKE`. Rewrite queries to leverage indexes.
- **Use** `EXPLAIN` to optimize queries and confirm index usage.

By designing indexes thoughtfully and aligning queries with index structures, you can significantly enhance MySQL query performance while avoiding common inefficiencies.