□ 1. INDEXING — The Real Deep Dive

Most people think indexes are “like the index of a book.”

That is **wrong and shallow**.

Indexes in MySQL (InnoDB specifically) are **B+ Trees**, and their behavior decides your entire system’s performance.

□ What is a B+ Tree?

Think of it like:

- A sorted lookup structure
- With “nodes” that contain ranges of values
- Balanced so that any lookup is **O(log n)** time
- Designed for disk access, not memory

Why disk?

Because disk is slow.
RAM is fast.
B+ Trees minimize disk reads by:

- Grouping many values into one node
- Keeping the tree height small (usually 3–5 levels)
- Allowing range scans

□ Why Indexes Speed Up Queries

Example without index

SELECT * FROM bookings WHERE room_id = 5;

MySQL must:

- Scan **every row** (FULL TABLE SCAN)
- Check room_id one by one

If table has **1 million rows**, you read 1 million rows.

With index on (room_id):

MySQL jumps into the B+ tree:

- Find the node containing room_id = 5
- Jump directly to matching rows

No scanning.
No wasted reads.

□ Composite Indexing (The real power)

Example:

INDEX (room_id, checkin_date, checkout_date)

This is *one index* with multiple columns, not three indexes.

Important rule:

Leftmost-prefix rule

In the above index, MySQL can use:

- room_id
- room_id + checkin_date
- room_id + checkin_date + checkout_date

But NOT:

- checkin_date alone
- checkout_date alone
- checkin_date + checkout_date without room_id

This catches developers off guard.

□ Index Misconception to Challenge

“Adding indexes always improves performance.”

No.

Excessive indexes:

- Slow down writes (INSERT/UPDATE/DELETE must update every index)
- Increase storage
- Cause locking contentions

Indexes are **not free**.
They are a trade-off.

□ 2. Transactions and ROLLBACK

A **transaction** is a group of SQL operations that behave as **one atomic operation**.

Example:

```
START TRANSACTION;

INSERT booking
INSERT payment
UPDATE room status

COMMIT;
```

If ANY step fails, you do:

```
ROLLBACK;
```

Everything reverts exactly as if nothing happened.

ACID Properties (deep explanation)

A — Atomicity

All or nothing.
If you book a room but payment fails → booking must not exist.

C — Consistency

Database must always remain valid.
Foreign keys, triggers, constraints must hold true.

I — Isolation

One user’s transaction should not *interfere* with another’s.

D — Durability

After COMMIT, data must survive:

- server crash
- power loss
- OS kill

InnoDB achieves this with:

- Redo logs
- Write-ahead logs
- Doublewrite buffer

□ 3. ENGINE=InnoDB — The Real Architecture

Here’s what InnoDB actually gives you:

✔ Row-Level Locking

Only the rows being changed are locked.

Compare:

MyISAM

- Locks entire TABLE
- Bad for concurrency
- Ancient

InnoDB

- Locks rows individually
- Multiple users can update different rows at the same time
- Critical for high-performance booking systems

✔ Clustered Index

The **PRIMARY KEY** determines physical row order.

This is why:

- Primary key should be short, sequential, never-changing
- BIGINT AUTO_INCREMENT is ideal

If you choose a bad primary key (e.g., UUID):

- Inserts become random I/O
- Fragmentation increases
- Performance tanks

✔ MVCC (Multi-Version Concurrency Control)

This allows **consistent reads without locking**.

Example:

- Transaction A reads row
- Transaction B writes row
- Transaction A still sees the “old version” of the row
- B sees the updated version

This avoids blocking and deadlocks.

Implemented via:

- Undo logs
- Snapshot reads

✔ Doublewrite Buffer

Guarantees durability:

- Writes pages to buffer
- Writes buffer to disk
- Even if system crashes halfway, data is recoverable

✔ Redo Log

Ensures changes are recoverable even after power loss.

✔ Foreign Key Enforcement

MyISAM completely lacks this.

□ 4. Locking — the real rules (not the watered-down version)

Types of locks:

✔ Shared Lock (S-Lock)

- For reading
- Many readers allowed
- Writers must wait

✔ Exclusive Lock (X-Lock)

- For writing
- Only 1 writer
- Blocks both readers and writers

Gap Locks and Next-Key Locks

This blows beginners' minds.

When searching for overlapping bookings:

SELECT ... FOR UPDATE

MySQL may lock:

- The row itself
- Gaps between rows (to prevent phantom reads)

This is why some innocuous queries suddenly block each other.

□ 5. Isolation Levels (very important but often misunderstood)

MySQL default = REPEATABLE READ

Meaning:

- If you read something, you will always see the same value within the transaction
- Even if other transactions update it

Levels:

1. **READ UNCOMMITTED** — dirty reads (bad)
2. **READ COMMITTED** — most DBs use this (PostgreSQL)
3. **REPEATABLE READ** — MySQL default (stronger)
4. **SERIALIZABLE** — essentially locks everything (very slow)

□ 6. Deadlocks (and why they're normal)

Deadlocks happen when:

- Transaction A locks row 1 then needs row 2
- Transaction B locks row 2 then needs row 1

They block each other → deadlock.

InnoDB automatically:

- Detects deadlock
- Kills one transaction
- Returns error 1213

Deadlocks are **normal**, not a sign of bad schema.
But bad indexing worsens them.

□ 7. How MySQL actually executes queries (mind-blowing clarity)

INSERT:

1. Lock row/gap
2. Insert into clustered index
3. Update secondary indexes
4. Write to redo log
5. Commit

UPDATE:

1. Lock row
2. Create new version in undo log
3. Update clustered index
4. Update secondary indexes
5. Write redo log
6. Commit

SELECT:

- Read snapshot from MVCC
- No locks unless FOR UPDATE or LOCK IN SHARE MODE

□ Final Big Concepts You Should Lock Into Your Brain

1. Indexes are sorted trees, not magic speed dust.

They accelerate queries but slow down writes.

2. InnoDB is a transactional engine that guarantees data safety.

It uses:

- MVCC
- Redo logs
- Undo logs
- Doublewrite buffers

3. ACID is not optional for booking/payment systems.

Otherwise you get:

- Double bookings
- Ghost payments
- Orphan rows

4. Transactions + proper indexing prevent race conditions.

Your stored procedure is correctly designed for atomicity.

5. Locking is complex but critical for high concurrency systems.

The real challenge is preventing unnecessary locks and deadlocks.