Lesson 11 -> Database Indexing

1️⃣ What Is Database Indexing?

A **database index** is a **data structure that allows fast lookup of rows** without scanning the entire table.

Think of it like:

- A book index → jump directly to page
- Instead of reading the entire book

Without Index

```
SELECT * FROM orders WHERE user_id = 123;
```

➡ Full table scan
➡ O(N)

With Index

```
Index(user_id) → pointer to row
```

➡ O(log N)

2️⃣ Why Indexing Is Important

1. Faster Reads

- Queries run orders of magnitude faster

2. Reduced Disk IO

- Fewer pages read from disk

3. Efficient Sorting & Filtering

- ORDER BY
- WHERE
- JOIN

4. Enables Scalability

- Without indexes, scaling hardware won't help

Trade-Off (Important)

Indexes are **not free**:

- Consume disk
- Slow down writes
- Need maintenance

3️⃣ How Indexes Work Internally (Simplified)

Most DBs use:

- **B-Tree / B+Tree**
- Some use **LSM Trees** (Cassandra)

Basic idea:

```
Key → pointer to row location
```

Example:

```
user_id = 123 → page 42, row 7
```

4️⃣ Types of Indexes

4.1 Primary Index (Primary Key Index)

What it is

- Index on **primary key**
- Usually **unique**
- Automatically created

Characteristics

- Fastest lookups

- Enforces uniqueness

Example

```
PRIMARY KEY (order_id)
```

Used for

- Row identity
- Joins
- Point lookups

4.2 Secondary Index

What it is

- Index on **non-primary columns**

Example:

```
CREATE INDEX idx_user_id ON orders(user_id);
```

Characteristics

- Can be non-unique
- Slower than primary index
- Essential for filtering

4.3 Clustered Index (Very Important)

What it is

- **Defines physical order of data on disk**
- Table rows stored in index order

Key Rule

👉 **Only one clustered index per table**

Example (MySQL InnoDB)

- Primary key is clustered index

```
Disk: [ PK=1 row ] [ PK=2 row ] [ PK=3 row ]
```

Pros

✅ Very fast range queries
✅ Fewer disk reads

Cons

❌ Insert/update expensive
❌ Reordering cost

4.4 Non-Clustered Index

What it is

- Index separate from table data
- Stores pointer to row

```
Index → Row pointer → Data
```

Pros

✅ Multiple per table
✅ Flexible indexing

Cons

❌ Extra lookup (bookmark lookup)

Clustered vs Non-Clustered (Key Difference)

Aspect	Clustered	Non-Clustered
Physical order	Yes	No
Count per table	One	Many
Speed	Faster for ranges	Slightly slower
Use cases	PK, time-series	Filters, joins

5️⃣ Composite (Multi-Column) Indexes

What it is

Index on multiple columns.

Example:

```
CREATE INDEX idx_user_date ON orders(user_id, order_date);
```

Important Rule

- Order matters

Works for:

- user_id
- user_id + order_date

Does NOT work for:

- order_date alone

6️⃣ Covering Index

What it is

- Query can be answered **entirely from index**
- No table lookup needed

Example:

```
SELECT user_id, order_date FROM orders
```

Index contains both columns.

Benefit

🚀 Extremely fast queries

7️⃣ Indexing in Different Databases

MySQL (InnoDB)

- Clustered index on primary key
- Secondary indexes point to PK

PostgreSQL

- No clustered index by default
- Can cluster table manually

MongoDB

- B-Tree indexes
- Secondary indexes supported

Cassandra

- LSM-tree based
- Primary key defines partition + clustering

Redis

- No traditional indexes
- Uses key-based access
- Sorted sets for range queries

8️⃣ Indexing Pitfalls (Architect Thinking)

Over-Indexing

- Too many indexes → slow writes

Wrong Column Order

- Composite index misuse

Low Cardinality Columns

- Index on boolean/status often useless

Index on Hot Columns

- Causes write contention

9️⃣ ML & Retail Use Cases

Feature Store

- Index on entity_id
- Index on (entity_id, timestamp)

Recommendation Systems

- Index user_id
- Index item_id

Order Systems

- Clustered by order_date
- Secondary on user_id