Database Optimization

DB Indexes

- Without an index, for a query, the DB must scan every row, compare each field. This is called full table scan.
- DB Index :- A data structure that helps the 'db' find rows faster.
- They store :- • Column value, • Pointer to the actual row.

Why

- Backend systems often search by email, filter by status, sort by data,
  Or without indexes queries become slow. CPU usage increases
- B-Tree (Common) :- • Allows logarithmic search time (log(n))
  - Efficient range queries
  - Fast sorting
- Indexes help when :-
  - Filtering (WHERE clause)
  - sorting (order by)
  - Joining tables
  - Range queries (BETWEEN)
- Indexing works when there is uniqueness in tables.

How to decide what to Index

Q. What queries run most frequently?
Q. What columns appear in WHERE?
Q. What columns are used for sorting?
Index based on usage, not guesswork.

Summary :- • Fast lookups
• Efficient filtering, sorting & joins.

Indexing strategies

Indexing is not about adding indexes everywhere, it is about designing indexes based on real query patterns.

* Always index based on :- How data is queried, not how it is stored.

1) Index columns used in where Clause :-

If ~~query~~ queries frequently filter by :-

email =?
user_id =?
status =?

} Should be indexed. because without indexed forces full table scan.

2) Index columns used in JOINs :

Always index — 1) foreign keys 2) JOIN columns.

3) Composite index (Imp)

If queries filter by multiple columns :- create single index instead of using two separate indexes.

4) Covering Index Strategy

Create an index that includes all needed columns. If index includes both/all, DB does not need to read table at all, It reads only the index.

5) Index for Sorting :- If queries frequently use :-

ORDER BY created_at DESC;

- Index on created_at helps.

Note :- Index works best when column has many unique values.

Query Optimization

- Designing queries & indexes so the 'DB' can retrieve data using minimal resources. It reduces:-
  - CPU usage
  - Disk I/O
  - Memory consumption
  - Query execution time

Execution of a Query

*) Parses SQL → Creates execution plan → Decides whether to use index
↓
Executes plan

Methods to Optimize

1) Use indexes Properly (Already explained in last page)
2) Avoid 'SELECT *', select only needed columns.
3) Optimize Joins - Always index join columns.
4) Avoid functions on Indexed columns.
5) Use caching where Appropriate.
6) Never return 1 million rows in API, use limit & offset

Real Backend example

- suppose an app:- → lists user orders
  → sorted by latest
  → filtered by user_id.

Best strategy :- → Index on user_id
→ Composite index (user_id, created_at)
→ Pagination.
↙

- Pagination is the process of dividing large amount of data into smaller, manageable chunks (pages) when returning results from a database or API.

Normalization vs Denormalization

1) Normalization means organizing data into separate tables to reduce duplication & maintain consistency.

It ensures :-
- Data consistency
- Reduced redundancy
- Easier updates
- smaller storage usage.

2) Denormalization means intentionally adding duplicate data to improve read performance.

Ex :- Instead of joining users & orders every time, store both together (No join is needed)

Trade-off Between them

Normalization	Denormalization
• Better consistency	• Faster reads
• Slower reads (due to joins)	• More storage
• Faster writes (less duplication)	• Harder updates
	• Risk of inconsistency.

- Normalization is best when system is transaction heavy, writes are frequent, data relationship matters.
  Ex :- Banking, ERP systems, Payments.
- Use denormalization when large scale systems, read heavy, reporting systems. Ex :- social media feeds, caching layers.

Real system design

E-commerce system →
- Core transactional tables :-
  - Normalized (orders, users, payments)
- Product listing page
  → May use denormalized data for faster reads.
- Many system use hybrid approach :-
  → Normalized core database
  → Denormalized read models.