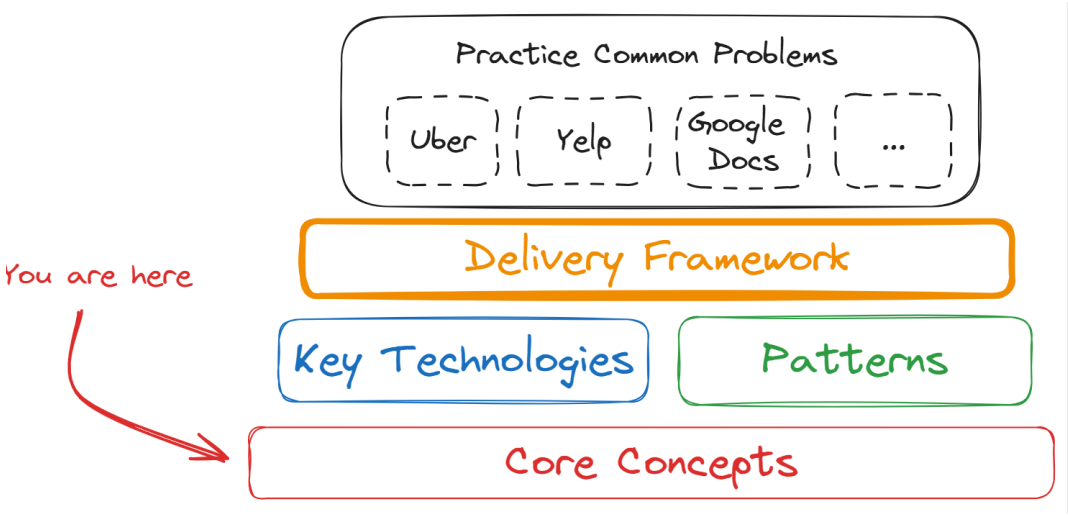
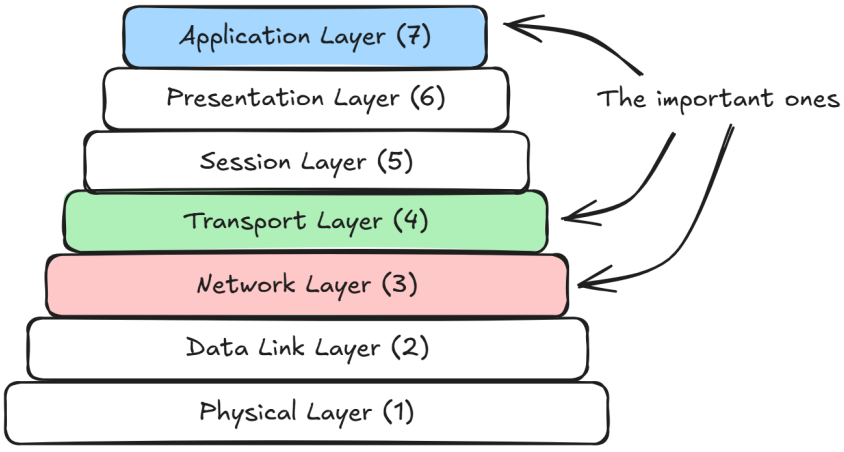
Core Concepts

Think of core concepts as the vocabulary and grammar of system design. Before you can discuss how to scale Instagram or design a ride-sharing service, you need to understand what caching is, when to shard a database, and how networks actually work. Interviewers assume you know these and will probe your understanding when you propose using them.

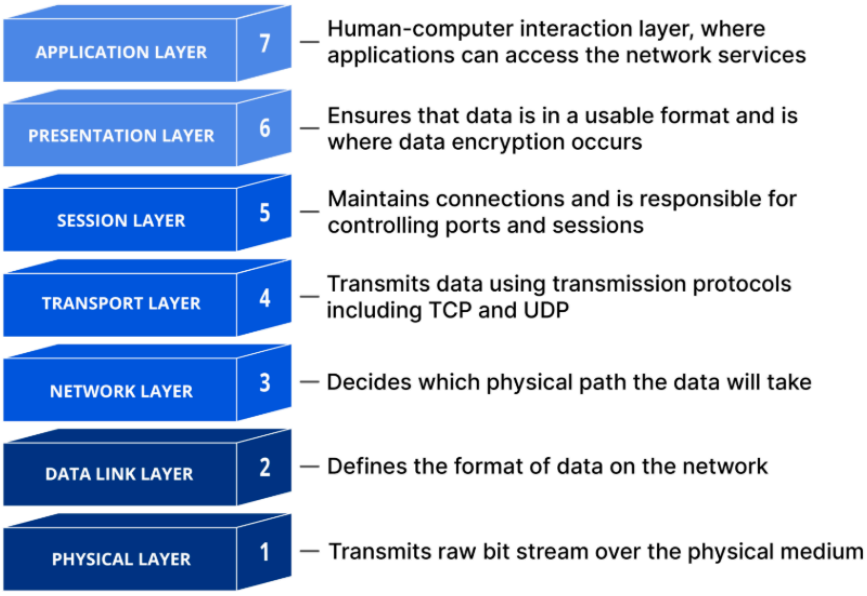


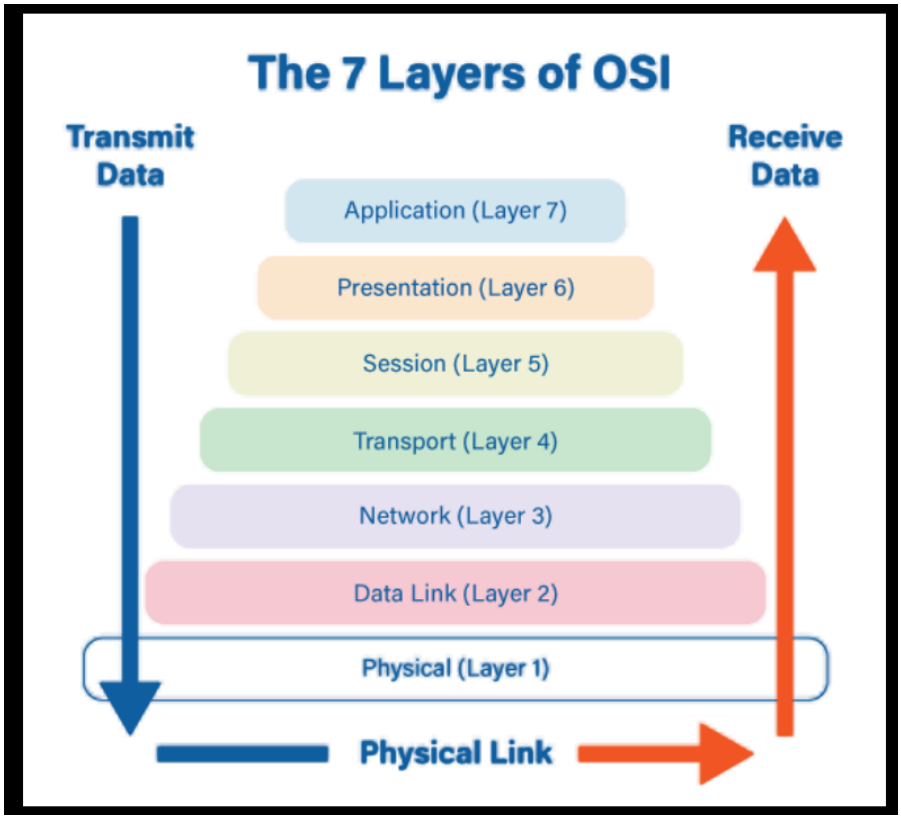
This page provides a quick overview of each core concept with just enough context to understand what it is, why it matters, and when to reach for it.

🌐 OSI Networking Layers — Explained for System Design Interviews



Networking Layers





This image shows the **7-layer OSI model**, but in system design interviews only **three layers matter the most**:

- **Layer 7 → Application Layer**

- **Layer 4 → Transport Layer**
- **Layer 3 → Network Layer**

Let's break them down from bottom to top.

✅ Layer 1 — Physical Layer (Not asked much)

- Deals with *wires, signals, fiber optics*.
- System design interviews almost never go here.

Think: actual electricity/light.

✅ Layer 2 — Data Link Layer (Not asked much)

- Handles MAC addresses, switching, frames.
- Relevant inside datacenters, but not in interviews.

Think: how machines talk within the same LAN.

⭐ Layer 3 — Network Layer (Important)

What it does

- Decides *where* a packet should go.
- Uses **IP addresses**.
- Routers operate here.

Key Protocols

- **IP (IPv4/IPv6)**
- **ICMP (ping)**

Why it matters in system design?

Because when you say:

➡️ *“Service A communicates with Service B over the network”*

…this is the layer responsible for routing packets globally.

You must know terms like:

- **IP addressing**
- **CIDR / subnets**
- **NAT**
- **Routing**
- **Anycast** (used by CDNs)

⭐ Layer 4 — Transport Layer (Very Important)

This layer is **critical** for understanding communication between services.

Main protocols

🟩 TCP

- Reliable
- Handles ordering, retries, congestion control
- Used by HTTP, gRPC, WebSockets

🟥 UDP

- Unreliable
- Very fast
- Used in streaming, gaming, DNS

Interview relevance:

When you say the system uses **HTTP**, you implicitly mean:

➡️ HTTP over TCP

This affects:

- Latency
- Persistent connections
- Load balancer behavior
- Retries & timeouts
- Backpressure

⭐ Layer 5 — Session Layer (Barely asked)

Responsible for:

- Sessions
- Authentication
- Connection control

In practice: HTTPS, cookies, JWTs work here conceptually.

But interviews don't drill this layer much.

⭐ Layer 6 — Presentation Layer (Barely asked)

Handles:

- Serialization (JSON, Protobuf, Avro)
- Encryption / SSL

❗ **Interview mapping:** When you discuss gRPC vs REST → you're talking about this layer.

Binary vs JSON = presentation layer concern.

⭐ Layer 7 — Application Layer (Most Important)

This is the most important for **system design interviews**.

Includes:

- **HTTP / HTTPS**
- **REST APIs**
- **GraphQL**
- **WebSockets**
- **gRPC**
- **SSE**
- **DNS**
- **SMTP (emails)**

This is *where your services speak business logic*.

📌 Why the diagram highlights these layers (7, 4, 3)

Because **these layers affect real distributed system behavior:**

Layer	Why It Matters in Interviews
7 – Application	Protocol choice: HTTP, WebSockets, SSE, gRPC
4 – Transport	TCP vs UDP, connection limits, latency
3 – Network	Routing, IP, load balancing, CDN anycast

📡 Now let's tie the explanation text to the OSI model

1. Communication protocols → Layer 7 + Layer 4

- “Use HTTP over TCP” = Application + Transport
- “WebSockets/SSE need persistence” = Layer 4 TCP behavior

2. WebSockets vs SSE → Layer 7

- SSE = server → client streaming
- WebSockets = bidirectional
- Both need long-lived TCP connections (layer 4)

3. gRPC → Layers 7 + 6

- Uses HTTP/2 (Layer 7)
- Uses binary Protobuf (Layer 6 serialization)
- Faster internal service-to-service communication

4. Load balancers → Layer 7 vs Layer 4

- **Layer 7 LB** → can inspect HTTP headers, routes intelligently
- **Layer 4 LB** → based only on TCP/IP

For WebSockets → you typically need **Layer 4** LB.

5. Latency & geography → Layer 3 (routing)

Fiber optic latency limits global performance:

- New York ↔ London ≈ 70–80ms minimum
  CDNs solve this by pushing data close to users.

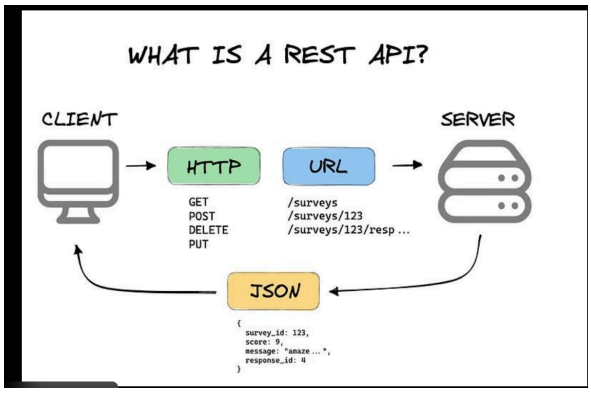
🧠 Interview Summary (What you should say)

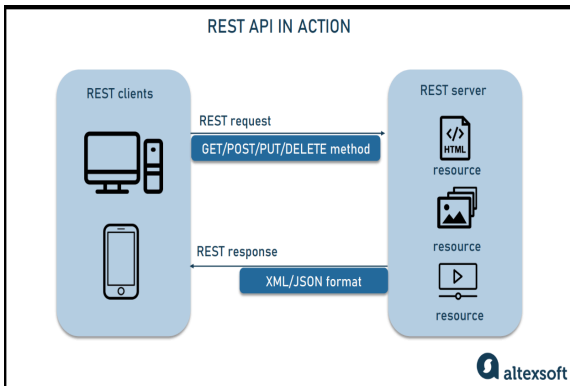
In system design interviews, the OSI model helps understand how services communicate:

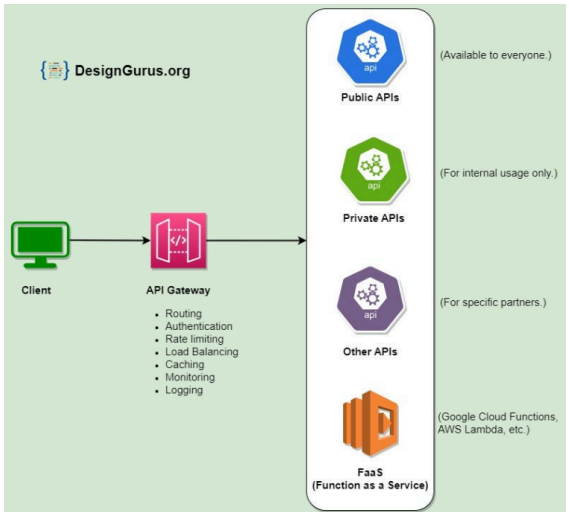
- Layer 7 → API protocol (HTTP, gRPC, WebSockets)
- Layer 4 → connection behavior, retries, congestion
- Layer 3 → routing, load balancers, IP, CDN

The other layers matter less for interview purposes.

🌐 API Design — Core Concept for System Design Interviews

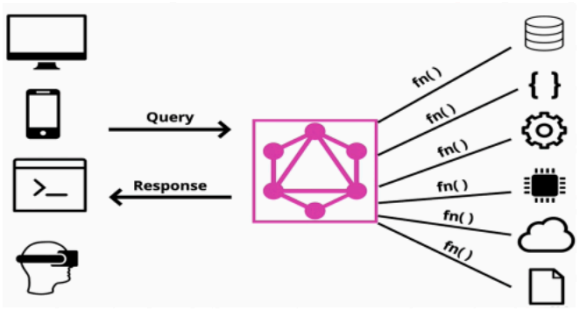


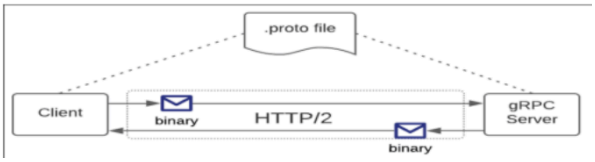


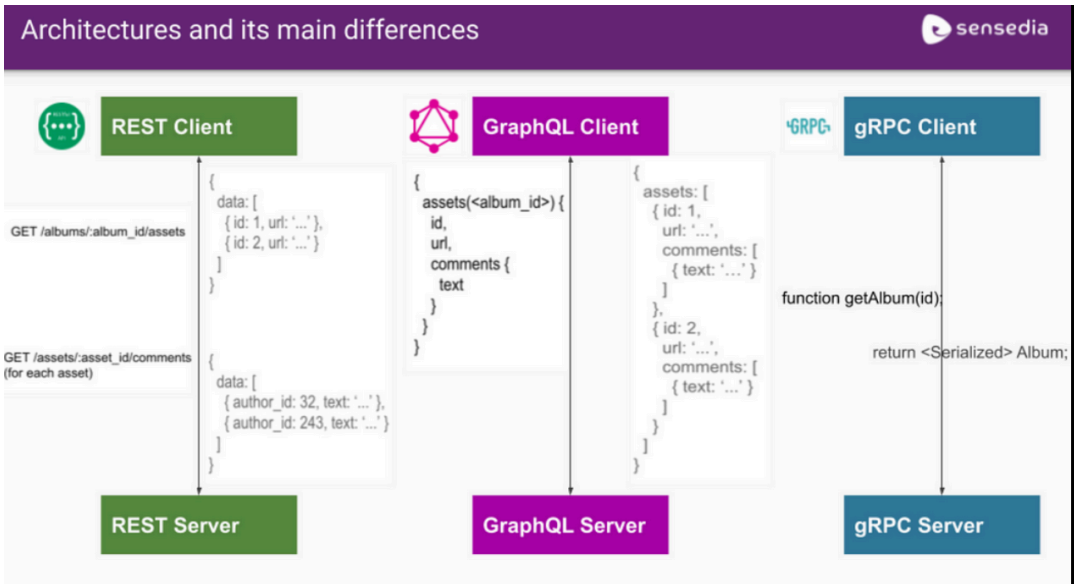

DesignGurus.org
Client
API Gateway
Routing
Authentication
Rate limiting
Load Balancing
Caching
Monitoring
Logging
Public APIs
(Available to everyone.)
Private APIs
(For internal usage only.)
Other APIs
(For specific partners.)
FaaS
(Function as a Service)
(Google Cloud Functions, AWS Lambda, etc.)


API
API
API Design
Best Practices


GraphQL
{ REST }
gRPC


Query
Response
fn( )
fn( )
fn( )
fn( )
fn( )
fn( )


.proto file
Client
binary
HTTP/2
binary
gRPC Server


Architectures and its main differences
sensedia
REST Client
GraphQL Client
gRPC Client
GET /albums/:album_id/assets
GET /assets/:asset_id/comments
(for each asset)
{
data: [
{ id: 1, url: '...' },
{ id: 2, url: '...' }
]
}
{
data: [
{ author_id: 32, text: '...' },
{ author_id: 243, text: '...' }
]
}
{
assets(<album_id>) {
id,
url,
comments {
text
}
}
}
{
assets: [
{ id: 1,
url: '...',
comments: [
{ text: '...' }
]
},
{ id: 2,
url: '...',
comments: [
{ text: '...' }
]
}
]
}
function getAlbum(id);
return <Serialized> Album;
REST Server
GraphQL Server
gRPC Server

API design is how **clients talk to your system**. Nearly every system design interview requires you to propose **some API endpoints**.

The interviewer doesn't want perfection—just **clean, reasonable endpoints** and an understanding of **when to use REST, GraphQL, or gRPC**.

1️⃣ Default Choice: REST APIs

✅ Why REST is the default in 90% of interviews

- Simple, predictable, everyone understands it
- Works across browsers, mobile apps, services
- Easy to scale, cache, load balance
- No special infra required

✅ How REST maps resources

It treats everything as a **resource** identified by a **URL**.

Examples:

- `GET /users/{id}` → fetch a user
- `POST /events/{id}/bookings` → create a booking
- `DELETE /posts/{id}` → remove a post

REST HTTP Methods

Method	Meaning
GET	retrieve data
POST	create
PUT	replace

PATCH	partial update
DELETE	delete

🎯 When to mention REST?

Always, unless interviewer hints otherwise.
REST is expected unless:

- Multiple frontends with diverse data needs
- Over-fetching/under-fetching issues
- Real-time streaming needed
- High-performance internal RPC

2️⃣ GraphQL — For Flexible Client Data Needs

✅ Why GraphQL exists

REST returns fixed responses → sometimes too much or too little data.

GraphQL lets the client request exactly what they need.

Example Request:

```
{
  user(id: "123") {
    name
    profilePic
    orders { id amount status }
  }
}
```

🎯 When to mention GraphQL in an interview?

When the problem includes:

- Mobile app + web dashboard + admin panel (different needs)
- Over-fetching (“REST returns unnecessary data”)
- Under-fetching (“client has to call multiple endpoints”)

⚠️ Problems with GraphQL (Important)

- **N+1 problem** (many DB queries triggered by one GraphQL request)
- Requires resolvers and query planners
- Not as easy to cache
- More complex backend

So **do NOT propose GraphQL unless there is a clear reason.**

3️⃣ gRPC — High-Performance Internal Communication

🚀 Key Benefits

- Faster than REST (binary Protobuf vs JSON)
- Uses HTTP/2 (multiplexing, streaming)
- Strict contracts via `.proto` files

🎯 When to mention gRPC

- **Microservices talking to each other internally**
- **Low latency required**
- **High throughput systems** (payments, search, ads ranking)

❌ Why not for external APIs?

- Browsers don’t support it well
- Harder to debug
- Requires special client libraries

⚡ **Common pattern in industry:**

- **REST for public API**
- **gRPC for internal microservices**

4️⃣ Pagination — Must Mention for Large Datasets

When returning lists:

- **Offset-based**: `?offset=0&limit=20`
  - Simple
  - Bad for fast-changing data (skips/duplicates)
- **Cursor-based**: `?cursor=abc123`
  - Better for real-time feeds
  - No missing items
  - Facebook/Twitter use this

🎯 Mention this whenever listing items (posts, orders, comments).

5️⃣ Authentication — Keep It Simple

Interview-safe options:

- **JWT tokens** → user session authentication
- **API Keys** → service-to-service authentication
- **OAuth** → only if interviewer asks for third-party login (Google, Facebook)

Do NOT go into too much depth unless asked.

6️⃣ Rate Limiting — Mention in Abuse Scenarios

Use when:

- The API could be spammed
- Bot protection needed
- Expensive operations (search, write-heavy APIs)

Common techniques:

- Token bucket
- Leaky bucket
- Fixed window / sliding window

Interview-safe phrasing:

> "We'll use rate limiting at the API gateway to prevent abuse."

7️⃣ Most Common Mistake in SD Interviews

👉 **Spending too much time on API design.**
Interviewers want:

- 4–5 clean endpoints

- Then move on to architecture, scaling, storage, caching

If you're still defining query params after 10 minutes → 🚫 bad sign.

Your goal:

> Define endpoints → get approval → move forward.

🎯 Example of What Good API Design Looks Like in an Interview

For an e-commerce system:

```
GET /products?category=shoes&limit=20
GET /products/{id}
POST /cart/{userId}/items
DELETE /cart/{userId}/items/{itemId}
POST /orders
GET /orders/{orderId}
```

Simple. Clean. Covers the basics.
Then immediately jump to architecture.

🧠 Interview Summary — What You Should Say

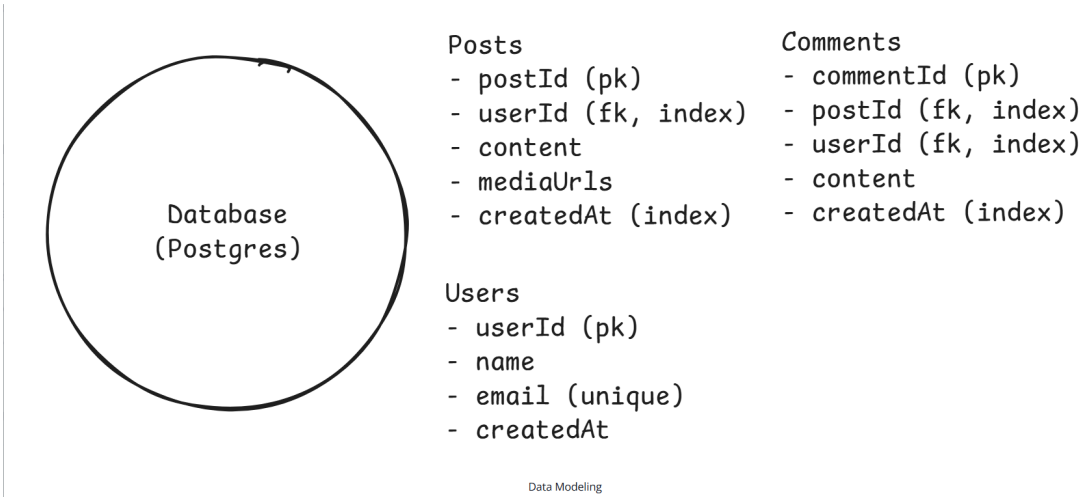
> "By default I'll expose REST APIs for the clients.
> If different clients need flexible data shapes, I can introduce GraphQL.
> For internal microservices, I'll use gRPC because of its performance benefits.
> All list responses will use pagination.
> I'll use JWT for authentication and apply rate limiting at the gateway."

This answer alone shows **maturity + practical knowledge**.

Data Modeling

Below is a **clean, structured, interview-ready explanation** of **Data Modeling** exactly based on the text you shared — but rewritten in a clearer, step-by-step format that impresses system design interviewers.

I'll break it into:
1) What is data modeling
2) Relational vs NoSQL
3) Normalization vs Denormalization
4) Access-pattern–based modeling (NoSQL)
5) Interview best practices



Data Modeling

🔶 1. What Is Data Modeling (and Why It Matters)

Data modeling is the process of deciding:

- **Which entities your system needs**
- **What attributes each entity should store**
- **How entities relate to each other**
- **Where and how the data will be stored (SQL/NoSQL)**

It affects everything downstream:

- **Query performance**
- **Scalability**
- **Consistency guarantees**
- **Ability to evolve the system**
- **Cost of reads and writes**

A *bad* data model forces hacks later. A *good* model makes scaling easier.

🔶 2. Relational vs NoSQL — The First Big Decision

✅ Relational Databases (Postgres, MySQL)

Use when:

- **Data is structured**
- **Clear relationships (users → posts → comments)**
- **You need strong consistency**
- **You need multi-row transactions**
- **You run complex queries (joins, filters, analytics)**

Benefits:

- **SQL is expressive**
- **Supports joins**
- **Enforces constraints (PK, FK, UNIQUE)**

- **ACID transactions**

Example schema (from your image):

Users

- **userId (pk)**
- **name**
- **email (unique)**
- **createdAt**

Posts

- **postId (pk)**
- **userId (fk, index)**
- **content**
- **mediaUrls**
- **createdAt (index)**

Comments

- **commentId (pk)**
- **postId (fk, index)**
- **userId (fk, index)**
- **content**
- **createdAt (index)**

This is clean, normalized SQL modeling.

✅ NoSQL Databases (DynamoDB, MongoDB)

Use when:

- **Schema changes frequently**
- **The system must scale horizontally**
- **You need extremely high throughput**
- **You can't afford slow joins**
- **Access patterns are very predictable**

Tradeoffs:

- **No joins**
- **No strong schema**
- **Eventual consistency (in most)**
- **You must explicitly design based on query patterns**

🔶 3. Normalization vs Denormalization

This is one of the most important tradeoffs interviewers look for.

✔️ Normalization (SQL default)

You split data into multiple tables and reference by IDs.

Example:

- `posts.userId` **references** `users.userId`

- `comments.postId` references `posts.postId`

Benefits:

- No duplicate data
- Strong consistency
- Updates are easy (change in one place)

Cost:

- Requires joins
- Joins get slow when tables grow
- Bad for high-read workloads

Use normalization as your starting point in interviews.

✔️ Denormalization (duplicate data intentionally)

You store extra fields to avoid joins.

Example:

- Each `order` stores the `userName` directly
- Each `post` stores the `userProfilePicUrl`

Benefits:

- Super fast reads
- No joins needed
- Great for high-traffic systems

Cost:

- **Updates are expensive (must update across many rows)**
- **Data can become inconsistent**
- **More storage needed**

When to use?

- **Read-heavy systems (social feed, analytics)**
- **Data that rarely changes**

Interview Tip: Start with normalized SQL → then say:

> **"If performance on read-heavy endpoints becomes an issue, we can selectively denormalize hot paths."**

This shows maturity.

🔶 4. NoSQL Access-Pattern–Driven Modeling

In NoSQL, schema is designed around queries, not around relationships.

Example problem:

> **"Get all posts for user X"**

In DynamoDB, best design:

- **Partition key = userId**
- **Sort key = createdAt**

This makes the query a single-partition O(1) lookup, extremely fast.

But another query like:

> **"Get all posts for hashtag #cricket"**

…is extremely slow because hashtag is not part of the key → requires scanning entire table.

Lesson:

> **In NoSQL, you must know your query patterns upfront.**
> **You cannot simply "add queries later" like SQL.**

🔶 5. Putting It All Together — Interview Best Practices

⭐ Start normalized in relational DB

Because:

- **Easy to reason about**
- **Enforces constraints**
- **Supports complex queries**
- **Interviewers expect this as the baseline**

⭐ Only denormalize if:

- **Read performance is bottleneck**
- **Data changes infrequently**
- **You identified a hot path**

⭐ Mention NoSQL if:

- **Massive scale**
- **Simple, predictable access patterns**
- **Need horizontal scalability**
- **Need very high throughput (e.g., millions of writes/sec)**

⭐ Speak in this format during interview:

> **"I'll start with a normalized SQL schema because the relationships are clear and consistency matters.**
> **If the read load grows, we can selectively denormalize specific fields like username or profile picture in the Posts table to avoid expensive joins.**
> **If the scale grows beyond relational limits, or if access patterns are highly predictable, we can move high-throughput parts to NoSQL."**

This is EXACTLY what interviewers want to hear.

Database Indexing

Indexing is one of the most important concepts in scalable system design.
A good index can make a query 1000× faster, and a missing index can make even a well-designed system crawl under load.

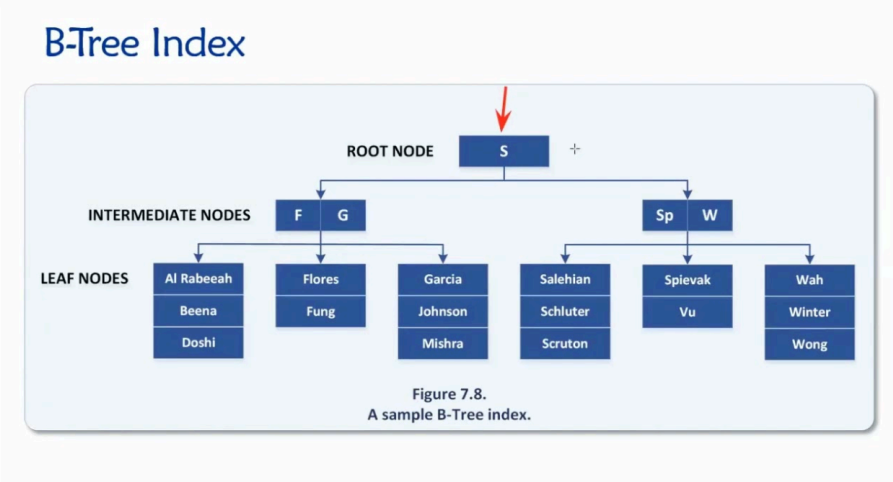
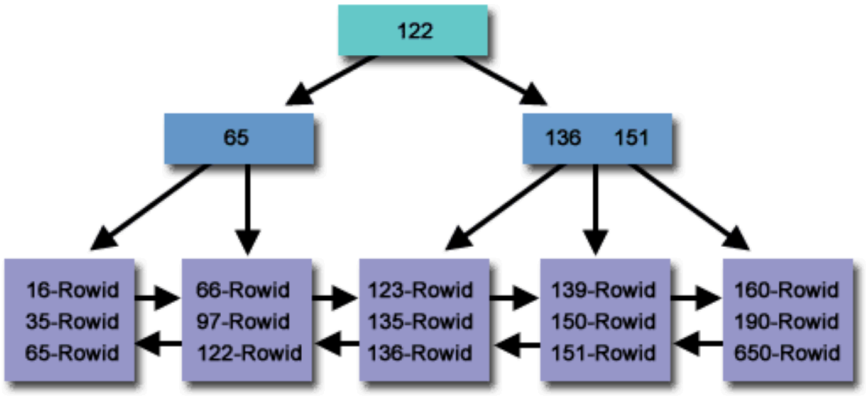
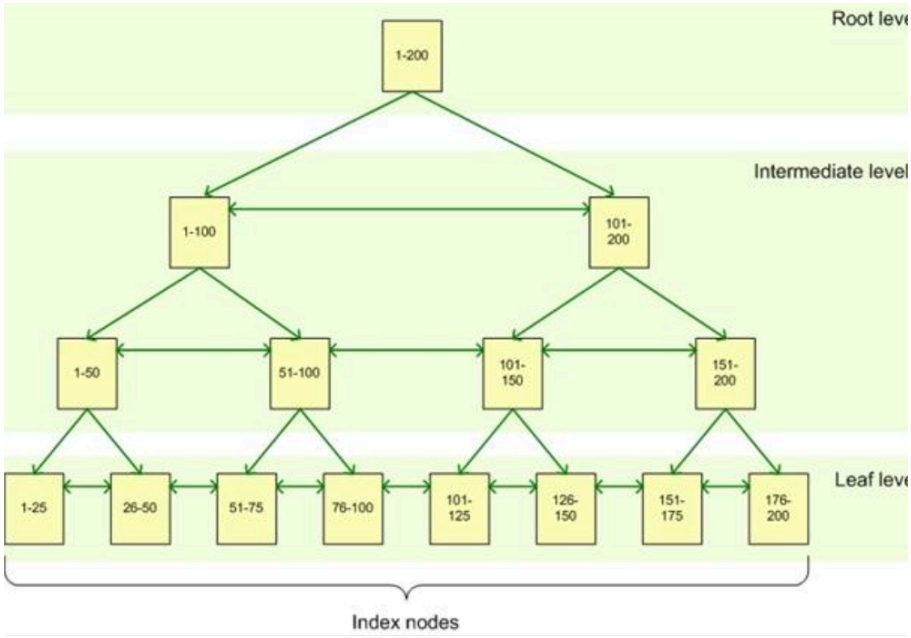
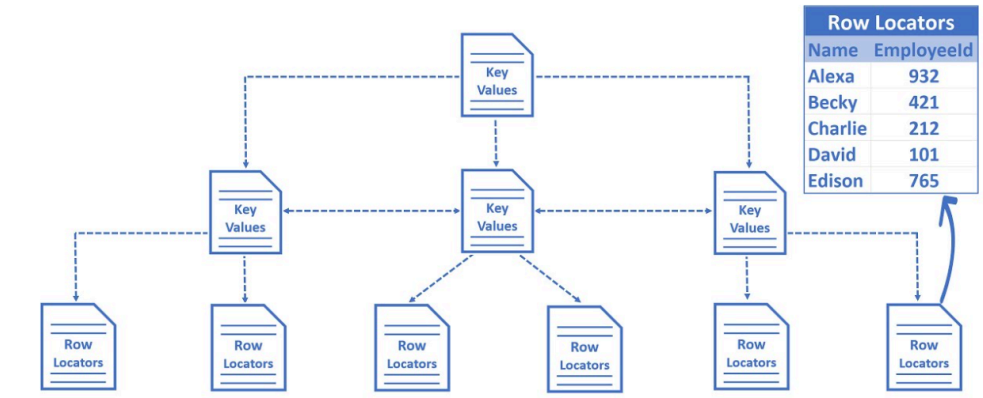


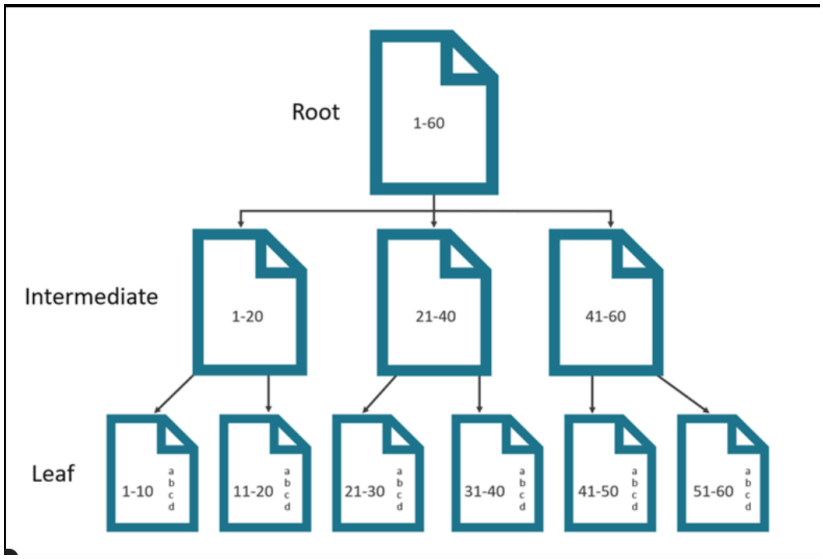
Figure 7.8.
A sample B-Tree index.

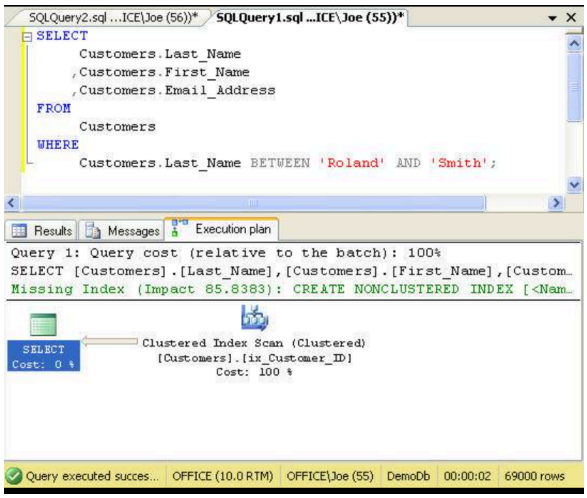




How do SQL Indexes Work







1️⃣ What is an Index? (Simple Definition)

An index is a lookup structure that lets the database jump directly to the required data without scanning the entire table.

Example:

Table: `users (id, name, email, createdAt)`

Query:

```
SELECT * FROM users WHERE email = 'alice@example.com';
```

- ❌ **Without index → full table scan → O(n)**
- ✔️ **With index on email → O(log n) using B-tree**

This matters at scale:

- **10M users = 10M rows**
- **Index reduces lookup time from seconds → milliseconds**

2️⃣ How Indexes Work (B-tree Basics)

Most relational DBs (Postgres, MySQL, SQLite) use B-tree indexes.

B-tree properties:

- **Sorted**
- **Allows binary search**
- **Supports:**

- Exact match queries: `WHERE email = ?`
- Range queries: `WHERE createdAt BETWEEN ? AND ?`

Visually, a B-tree lets the DB skip large portions of data instead of scanning everything.

3️⃣ Types of Indexes

✔️ 1. B-tree Index (most common)

Supports:

- **Exact lookups**
- **Prefix lookups**
- **Range queries**

Automatically created for:

- **Primary keys**
- **Unique constraints**

✔️ 2. Hash Index

- **Very fast for exact equality (`=`) lookups**
- **❌ Cannot support ranges (`<`, `>`, BETWEEN)**

Rarely used unless:

- **High-speed caching layer**

- **Very predictable queries**

✔️ 3. Full-Text Index

Used for text search:

- **Search posts containing the word “football”**
- **Search documents containing “machine learning”**

Databases with full-text search:

- **Postgres (`GIN/GiST indexes`)**
- **MySQL full-text**
- **Elasticsearch (external, more powerful)**

✔️ 4. Geospatial Index

Used for location queries:

- **Find restaurants within 5 km**
- **Find events near a user**

Tools:

- **PostGIS (Postgres extension)**
- **MongoDB geospatial indexes**

4️⃣ When to Create an Index (Interview Skill)

Think about query patterns, not tables.

If your system repeatedly does:

- **"Find user by email"**
  → Index `email`

- **"Get all orders for a user"**
  → Index `user_id` in orders table

- **"Get all posts for user ordered by date"**
  → Index `(user_id, createdAt)` composite index

Indexes should map directly to frequent WHERE clauses.

5️⃣ Composite Indexes — Extremely Important

Sometimes a single column isn't enough.

Example query:

```
SELECT * FROM events

WHERE city = 'San Francisco'

AND eventDate = '2025-12-25';
```

Best index:

```
(city, eventDate)
```

Order matters.
The index can serve queries that filter by:

- city
- city + date
  But not date alone.

6️⃣ Tradeoffs — Indexes Are Not Free

Indexes make reads fast, but:

❗ 1. Slow down writes

Every insert/update/delete must update the index:

- More disk I/O
- More CPU
- More locking

❗ 2. Take extra storage

Sometimes as much as or more than the table itself.

❗ 3. Too many indexes = slower system

Rule of thumb:

> Index only what you query frequently.

Interviewers love candidates who explicitly mention these tradeoffs.

7️⃣ External Search Systems (Elasticsearch, PostGIS)

Sometimes DB indexes aren't enough.

✔️ Elasticsearch for full-text search:

- **Autocomplete**
- **Fuzzy search**
- **Tokenization**
- **Ranking**

Works great for:

- **Searching tweets**
- **Searching product catalog**
- **Log search (ELK stack)**

It syncs from the main DB via CDC (Change Data Capture).
This introduces eventual consistency—acceptable for search.

✔️ PostGIS for geospatial queries:

- **Distance queries**
- **Intersection**
- **Bounding boxes**

Used for:

- **Maps**
- **Delivery apps**
- **Ride-sharing systems**

8️⃣ Interview Strategy — Exactly What to Say

When indexing comes up, answer like this:

> **"I'll add indexes on columns we frequently filter by.**
> **For example, users.email for login, orders.user_id for fetching user orders, and a compound index on (city, eventDate) for event lookups."**
>
> **"For full-text search like searching posts or tweets, I'll integrate Elasticsearch and sync it via CDC. This gives fast search with acceptable eventual consistency."**
>
> **"I'm aware that indexes speed up reads but slow down writes, so I'll only add them on hot query paths."**

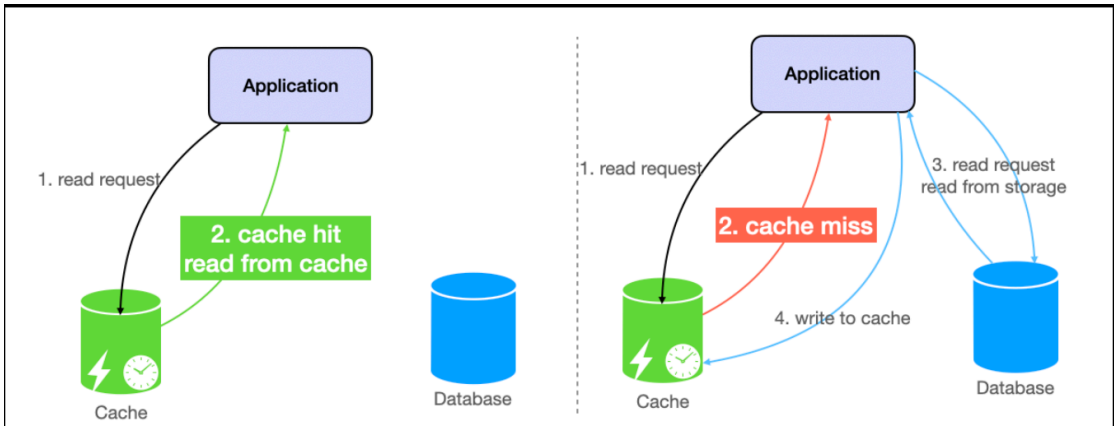
This shows mastery without overcomplicating.

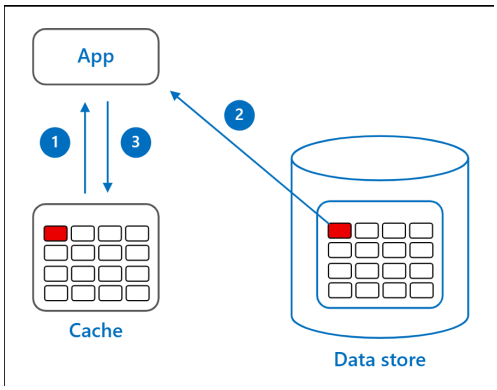
9️⃣ Small Table Summary

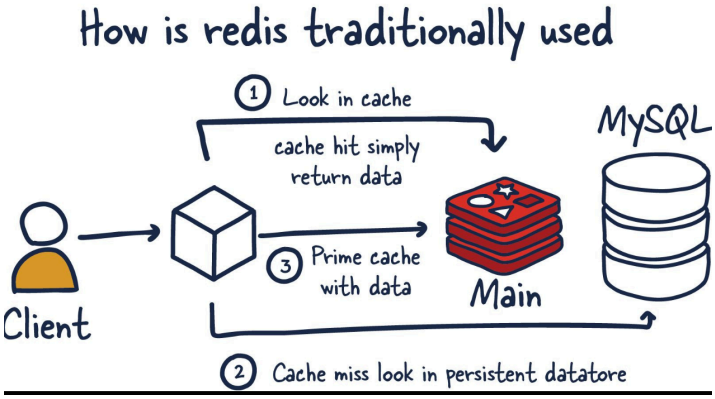
Type	Supports	Used For	Notes
B-tree	Exact + Range	Most queries	Default
Hash	Exact only	Fast lookups	No ranges
Full-text	Token search	Posts, docs	Often external
Geospatial	Coordinates	Maps	PostGIS

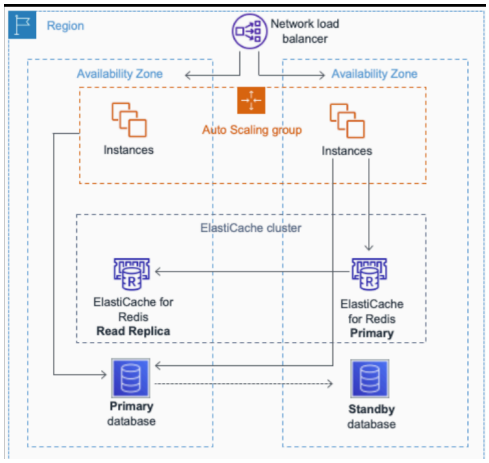
Caching

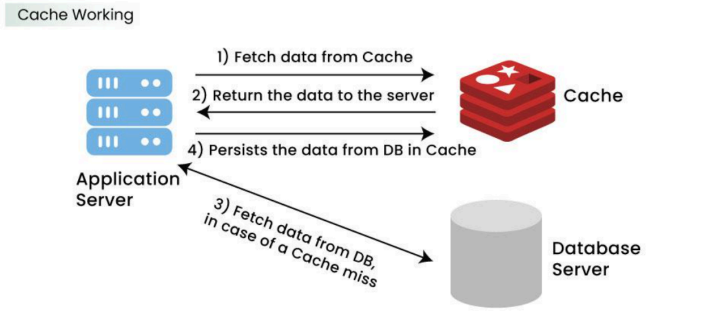
⚡ Caching — Core Concept for System Design Interviews

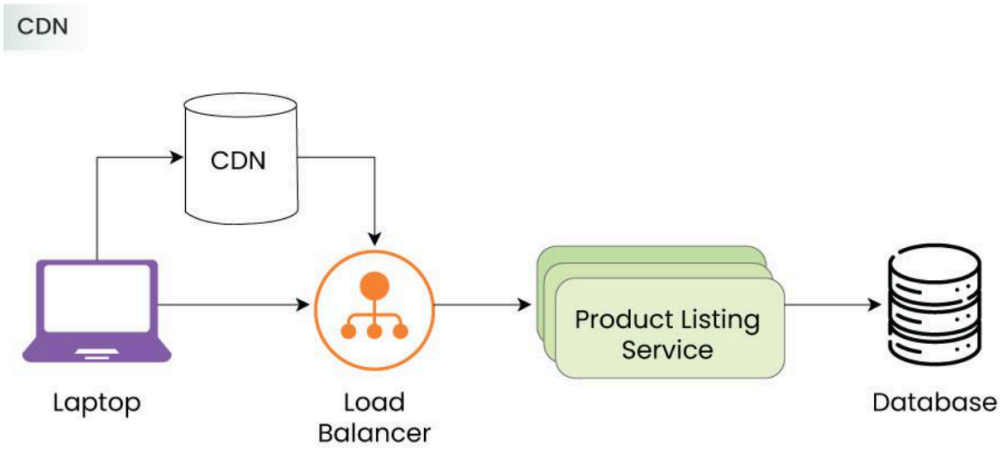

Application
1. read request
2. cache hit
read from cache
Cache
Database
Application
1. read request
2. cache miss
3. read request
read from storage
4. write to cache
Cache
Database


App
1
2
3
Cache
Data store


How is redis traditionally used
1 Look in cache
cache hit simply
return data
MySQL
Client
3 Prime cache
with data
Main
2 Cache miss look in persistent datatore


Region
Network load
balancer
Availability Zone
Availability Zone
Auto Scaling group
Instances
Instances
ElastiCache cluster
ElastiCache for
Redis
Read Replica
ElastiCache
for Redis
Primary
Primary
database
Standby
database


Cache Working
1) Fetch data from Cache
2) Return the data to the server
Cache
4) Persists the data from DB in Cache
Application
Server
3) Fetch data from DB,
in case of a Cache miss
Database
Server



Caching is how you make a system read data fast and reduce database load.
Your DB is slow (20–50 ms). Redis is extremely fast (0.5–2 ms).

At scale, that 20–50× speedup is the difference between:

- **Needing 10 DB servers**
- **Or just one DB plus Redis**

---

1 Why We Use Caching

✔ Massive performance boost

- **Redis lookup → ~1 ms**
- **SQL DB query → 20–50 ms**

✔ Protects your database

When 80–95% of reads come from cache, your database only handles:

- **Writes**
- **Rare cache misses**

✔ Reduces latency for users

Your app feels instantly responsive.

2️⃣ External Caching (Redis) — The Interview Default

The most common pattern is Cache-Aside (also called Lazy Loading).

🔄 Cache-Aside Workflow

1. **Read request comes in**
    - **Check Redis**
    - **If value exists → return it (cache hit)**
    - **If not → fetch from DB (cache miss)**
2. **On miss**
    - **Load data from database**
    - **Store it in Redis with a TTL**
    - **Return result**

This is used in 90% of real production systems.

3️⃣ The Hard Part: Cache Invalidation

Caching is easy; keeping it fresh is hard.

Common strategies:

✔ Strategy A — Invalidate Cache After Write

When DB updates:

- **Delete the old value in Redis**
- **Next read will fetch fresh data**

Pros:

- **Always fresh**

Cons:

- **Needs careful consistency control**
- **Race conditions possible (write occurs while cache is being rebuilt)**

✔ Strategy B — Short TTLs (Expire Quickly)

Let the cached value expire automatically, e.g. TTL=5 minutes.

Pros:

- **Very simple**
- **No explicit invalidation needed**

Cons:

- **Accept some stale data**
- **More DB load on expiry spikes**

✔ Strategy C — Hybrid: Invalidate + TTL

Most real-world systems use:

- **Immediate invalidation after writes**
- **Plus a TTL for safety**

4️⃣ Cache Stampede — A Critical Interview Concept

If Redis crashes or expires many keys at once, all requests hit the DB.
This sudden spike can take the DB down.

This is called a Cache Stampede or Thundering Herd.

Solutions:

✔ Staggered TTLs

Avoid many keys expiring at the same time.

✔ Background refresh

Refresh popular keys before they expire.

✔ Request coalescing

Only one request rebuilds cache; others wait.

✔ Circuit breakers

Prevent DB overload by:

- **Returning stale data**
- **Rejecting requests temporarily**
- **Falling back to in-process cache**

✔ Local fallback cache

Keep a tiny in-memory cache so the DB isn't overwhelmed.

5️⃣ Common Mistake: Caching Everything

This is a major red flag in interviews.

Do NOT cache:

- **Data that updates very frequently**
- **Per-user session data (unless small)**
- **Data that is not read often**

Cache only:

- **Frequently read**
- **Expensive to compute or query**
- **Doesn’t change rapidly**

Examples:

- **User profile**
- **Product details**
- **Feed items**
- **Configuration flags**

Profiling comes first → caching comes after.

6️⃣ Types of Caching (3 Levels)

🔵 1. External Cache (Redis) — *Main one in interviews*

Used for:

- **Application data**
- **User objects**
- **Feed items**
- **Session tokens (sometimes)**

🟢 2. CDN Caching (Images, video, JS, CSS)

Stored at edge locations close to users.

Used for:

- **Static files**
- **Public images**
- **Video streaming**
- **Web assets**

Does NOT handle:

- **App-level dynamic data**
- **Logged-in personalized content**

🟡 3. In-process Cache (Local memory)

Examples:

- **Guava Cache**

- **Caffeine**
- **LRU maps**

Used for:

- **Feature flags**
- **Configuration**
- **Things that rarely change**

Downside:

- **Each instance has its own copy**
- **No consistency across servers**

7️⃣ When to Bring Up Caching in Interviews

Bring it up when:

- **You see high read traffic**
- **DB becomes a bottleneck**
- **API latency needs improvement**
- **Data is stable and frequently accessed**
- **You're designing a feed, product catalog, or profile service**

Interviewers expect you to suggest it early in scaling discussions.

---

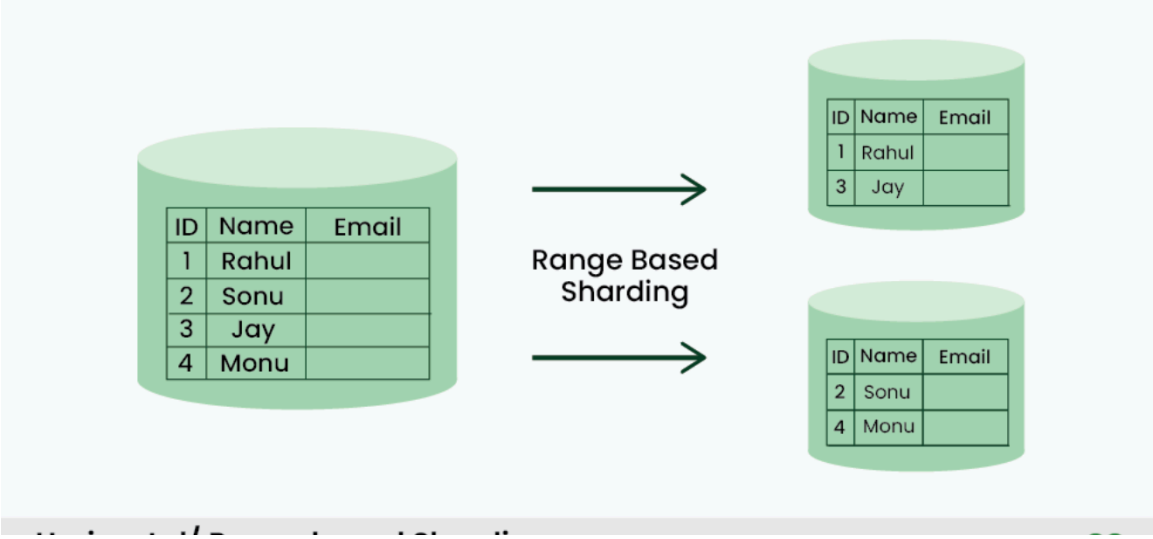
8 Interview-Ready Summary

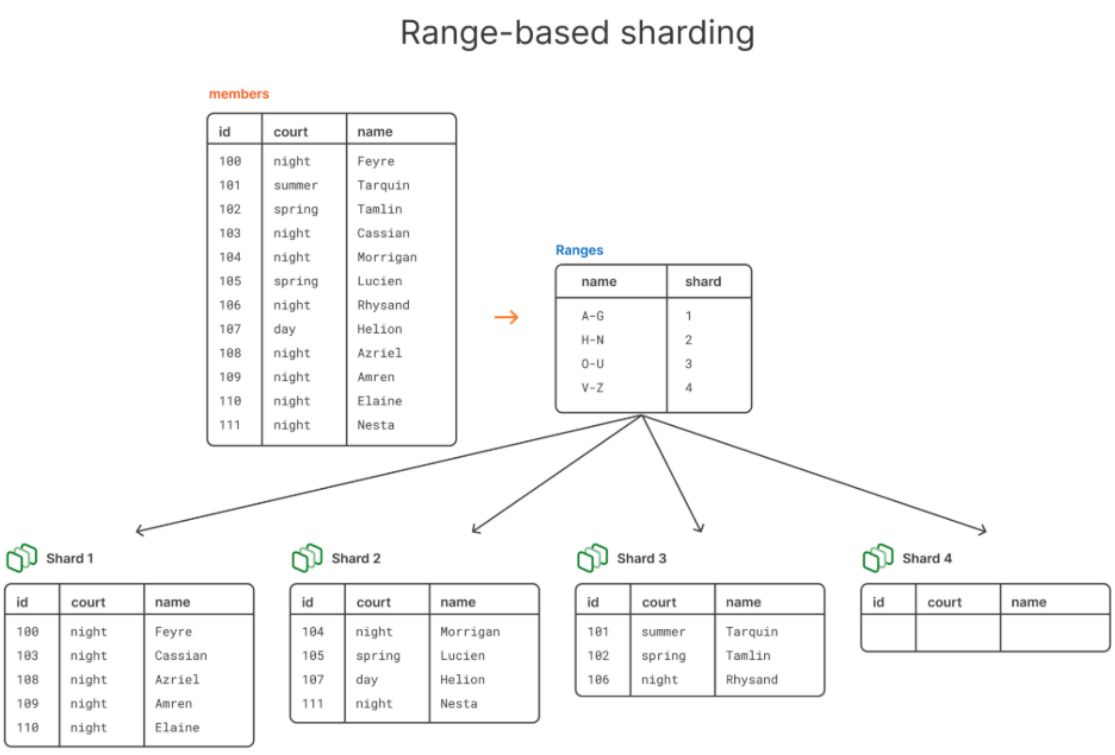
Here's exactly what you should say:

> **"To reduce DB load and improve latency, I'll add a Redis cache using the cache-aside pattern.**
> **On reads we check Redis first, then fallback to DB on misses.**
> **We'll use TTLs plus explicit invalidation after writes to keep data fresh.**
>
> **To avoid cache stampedes, we can use request coalescing, staggered TTLs, and possibly an in-process fallback cache.**
>
> **For static content like images, we'll use CDN caching at the edge."**

Perfect, concise, and shows deep understanding.

Sharding





The diagram shows:

- **An Original Database that has reached its limit (storage or throughput).**
- **To scale further, we split the data into multiple smaller databases, each called a shard.**
- **Each shard contains a *portion* of the overall dataset.**

This architecture is called Database Sharding.

🔍 Deep Explanation of the Given Diagram

🎯 1. Original Database (at capacity)

This is your single monolithic database:

- **It contains *all* records.**
- **It is handling too many reads/writes.**
- **Storage or throughput is maxed out.**
- **Adding read replicas no longer helps because writes or storage don't scale horizontally.**

At this point, sharding becomes necessary.

🪓 2. Splitting Data Across Multiple Shards

In the diagram, data is split into 3 shards based on an ID range:

Shard	ID Range	Example Data
Shard 1	0 – 10M	users with id < 10,000,000
Shard 2	10M – 20M	users with id between 10–20M
Shard 3	20M – 40M	users with id between 20–40M

This is called Range-Based Sharding.

🧠 3. How Does the System Use This?

When a new row enters the database:

- **It looks at the ID.**
- **Based on the range, it routes the request to the correct shard.**

Example:

A user record with:

```
user_id = 12,532,441
```

The routing logic checks:

```
12.5M lies in 10M–20M → Store in Shard 2
```

Every read/write will follow this rule.

⚙️ 4. How Data Balancing Works

Each shard has:

- **Its own CPU**
- **Its own RAM**
- **Its own disk**
- **Its own query load**

Meaning you effectively multiply your capacity:

If one DB can handle:

- **5,000 writes/sec**
- **500GB storage**

Then 3 shards can handle:

- **15,000 writes/sec**
- **1.5TB total storage**

🎭 Tradeoffs in This Sharding Approach

✅ Pros

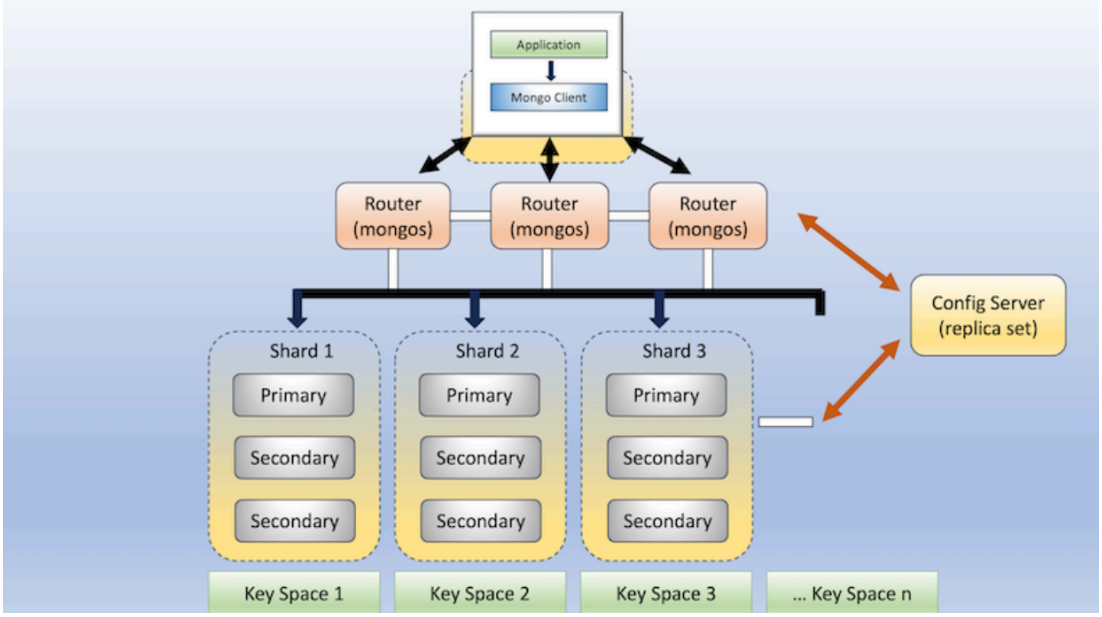
- **Very easy to understand.**
- **Great for predictable ranges.**
- **Queries within a range are extremely fast — only one shard is hit.**

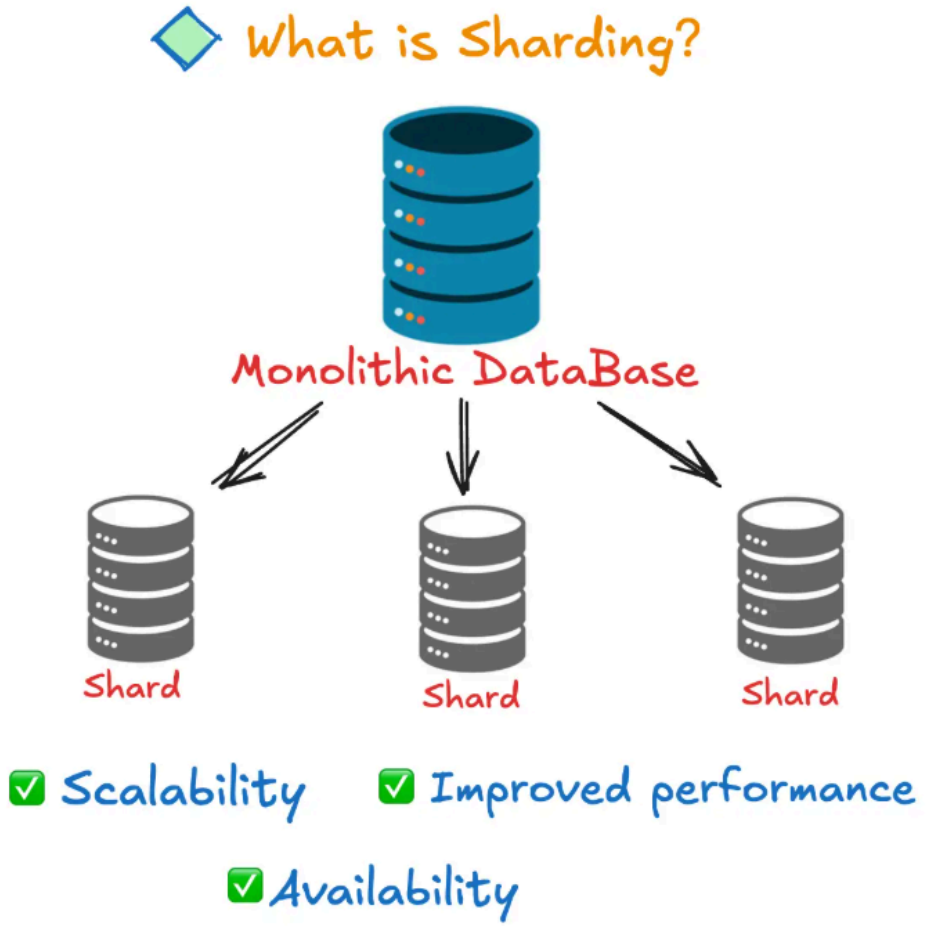
❌ Cons

- **Hotspot risk:**
  If most new users have high IDs, Shard 3 gets overloaded.
- **Rebalancing painful:**
  Changing the ranges later requires moving millions of rows.
- **Global queries expensive:**
  Example: *"Top 10 trending posts"* → must query every shard, merge, sort.

🛒 If This Were an E-Commerce System (Your Earlier Question)

Let’s connect it to an e-commerce example:





⚡ What could be sharded?

Entity	Good Shard Key	Why
Users	user_id	user-specific queries hit only one shard
Orders	user_id OR order_id	keeps a customer's order history localized
Products	product_id	searches by product_id become easy

Example:

- User with id = 9M → Shard 1
- User with id = 17M → Shard 2
- Orders placed by user 17M → Stored in Shard 2
- Product catalog → Might be sharded by product_id ranges

This keeps each shard responsible for a manageable portion of traffic.

🌍 Global Operations Become Harder

Operations like:

- “Give me all orders from last week”
- “What products sold the most today?”

→ require hitting every shard and merging results.

This is expected and acceptable — system design interviews want you to mention such tradeoffs.

🗝️ Choosing the Right Shard Key

You choose a key to ensure:

1. Even distribution of data
2. Minimal cross-shard operations
3. No single shard becomes a bottleneck

For most systems, the shard key is:

- user_id (hash) → Best for social networks
- order_id (range/hash) → Good for commerce
- company_id → Good for SaaS multi-tenant systems

🧨 Why Sharding Is NOT Done Early

Sharding brings complexity:

- Cross-shard transactions become expensive
- Hard to maintain consistency
- Rebalancing (resharding) is painful
- Requires additional routing logic (Shard Manager / Lookup Service)

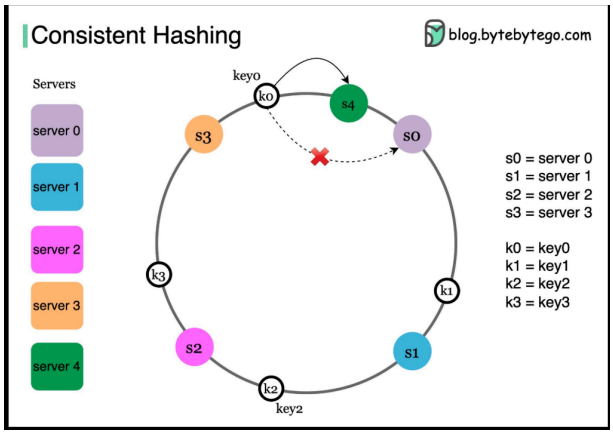
Interview tip:
👉 **Always say you will shard *only when* necessary, based on capacity calculations.**

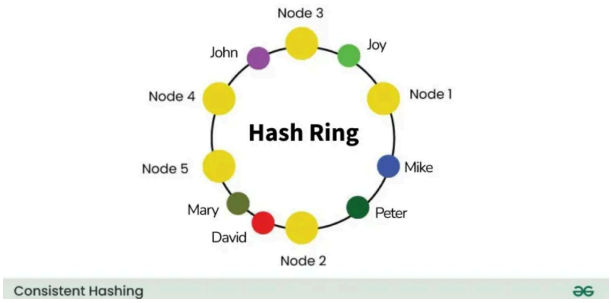
🚀 Summary of the Diagram in Simple Words

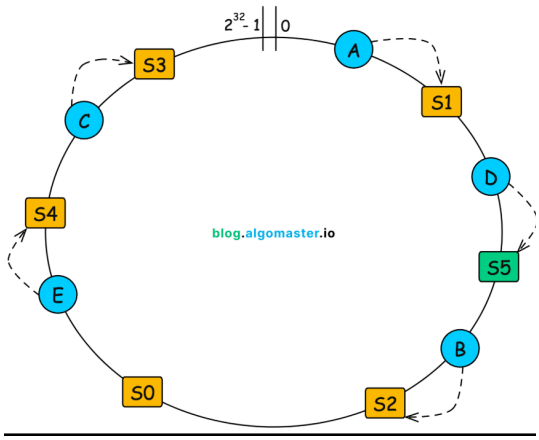
The image shows splitting a single overloaded database into multiple smaller ones using ID ranges, so:

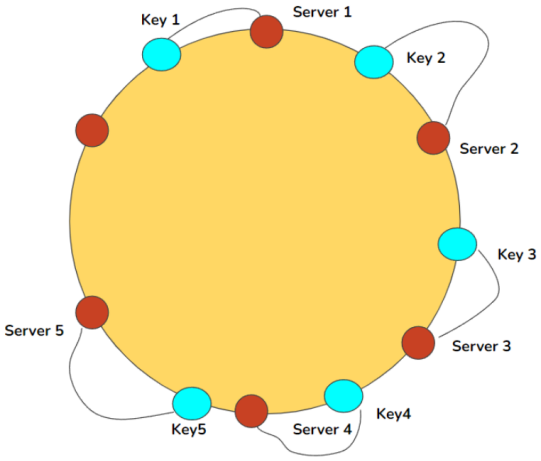
- **Each shard stores a subset of rows.**
- **Each shard handles a subset of traffic.**
- **The system can now scale horizontally.**

Consistent Hashing









You have:

- **A ring numbered from 0 → 100 (a virtual circular space).**
- **Several databases (DB1, DB2, DB3, DB4) placed at different points on the ring.**
- **A *key* (event 1234) that is hashed to a position on the ring (hash(1234) = 16).**

Consistent hashing maps keys to servers using a circular structure, not a simple modulo.

✨ Step-by-Step Explanation of the Diagram

Let's rewrite and expand what's happening in the image.

1️⃣ Convert the key into a hash value

The problem asks:

> **Where does event 1234 live?**

Step 1:
Compute:

`hash(1234) = 16`

So the key sits at position 16 on the ring.

2️⃣ Move clockwise to find the first database node

From position 16, you follow the ring clockwise until you hit the next database shard.

In the diagram:

- **Starting at 16 → moving forward → you reach DB2 at around position 25.**

So:

👉 Event 1234 is stored on DB2

This rule repeats for every key.

🧠 Why Do We Use This Instead of `hash(key) % N`?

Let's go deeper because this is the *interview-winning* part.

❌ Problem with Simple Hashing (`% N`)

If you pick shard like:

```
shard = hash(key) % number_of_shards
```

This works fine *only until* the number of servers changes.

Example:

If you have:

```
key → hash(key) % 3
```

And you add one more server (now % 4),
almost every key gets mapped to a different shard.

This causes:

- **Cache invalidation**
- **Database migration**
- **Load imbalance**
- **System instability**

For millions of keys → disaster.

✔️ What Consistent Hashing Fixes

In consistent hashing:

⭐ Adding a new server affects ONLY a small portion of keys

Because only the keys that fall between:

```
previous server → new server
```

need to move.

⭐ Removing a server also affects ONLY its range

Keys that belonged to that server simply move to the next one clockwise.

≠ In contrast, modulo hashing moves all keys.

🔍 Detailed Example Using the Diagram

⭐ Suppose we ADD a new server at position 18

What happens?

- **Only some keys between DB2 (25) and the previous server behind 18 move.**
- **NOT the entire dataset.**
- **The system stays stable.**

⭐ Suppose we REMOVE DB2

- **Only the keys DB2 was responsible for move to the next node clockwise.**
- **In this case, DB3 (around 50).**

Again → only a subset of keys move.

🎯 Key Insight of Consistent Hashing

Consistent hashing's magic is:

👉 Only O(K/N) keys move when you add/remove a server

Where:

- **K = total keys**
- **N = number of servers**

Meaning roughly 10% of keys move when you add/remove one of 10 servers.

For simple modulo hashing:

❌ You move 90%+ of keys

For consistent hashing:

✔️ You move ~10% of keys

That's why distributed systems *always* use consistent hashing.

🏗️ Where It’s Used in Real Systems

Consistent hashing is a core technique in:

- **Redis Cluster**
- **Memcached client libraries**
- **Cassandra**
- **Amazon Dynamo / DynamoDB**
- **Akamai CDN routing**
- **Envoy / NGINX load balancers**
- **Kafka partition assignment (variation)**

It’s foundational infrastructure-level tech.

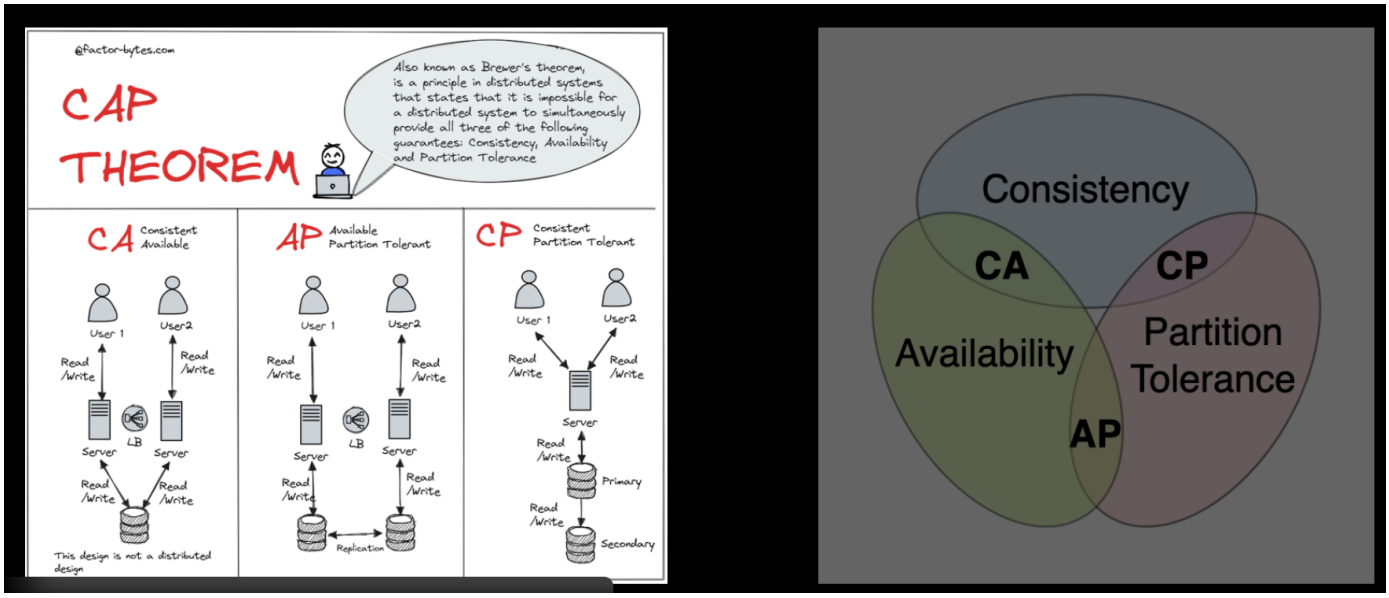
🧩 Stick This Explanation in Your Interview

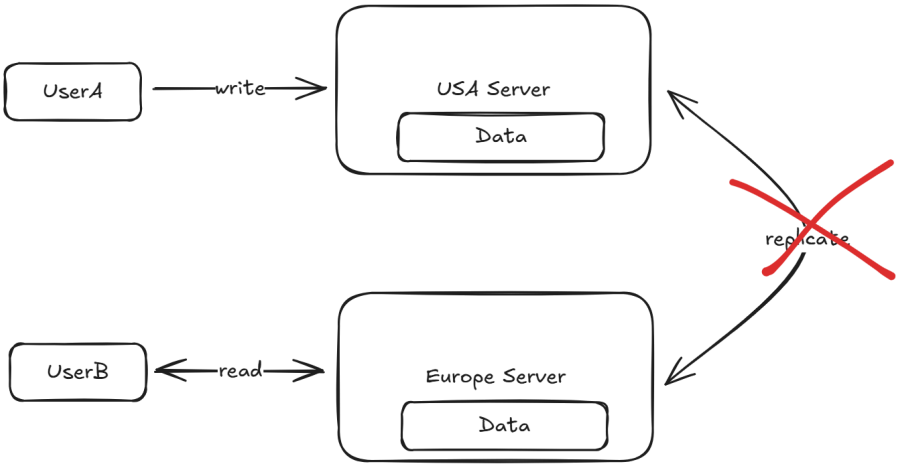
When choosing a strategy for distributing data across nodes, say:

> **"I will use consistent hashing so that adding or removing nodes causes minimal data movement and keeps the system stable."**

This is EXACTLY what interviewers want to hear.

CAP Theorem





CAP Theorem

🧠 1. What the Diagram Shows

The diagram has:

- A **USA server** storing data.
- A **Europe server** storing data.
- **User A** writes to the USA server.
- **User B** reads from the Europe server.
- A red X over the replication arrow between USA → Europe.

This picture illustrates **what happens when a network partition occurs** (i.e., replication link breaks).

🧩 2. What Is a Network Partition?

A *partition* means:

- The USA server and Europe server **cannot talk to each other** temporarily.
- The link used to replicate data between regions is **down**.

This is extremely common:

- Fiber cut under the ocean
- Routing failure
- Firewall rule issues
- Cloud network outage

Distributed systems **must work even when this happens** — that's where CAP comes in.

🔺 3. CAP Theorem in Simple Terms

CAP says in a distributed system, when a **partition** happens, you can only choose:

C — Consistency

All nodes return the same up-to-date data.

A — Availability

Every node responds to every request.

P — Partition Tolerance

System keeps working even when nodes cannot communicate.

Partition tolerance is mandatory — you cannot turn off network failures.

So you must choose between:

⭐ CP → Consistent + Partition Tolerant

You **sacrifice availability**.

OR

⭐ AP → Available + Partition Tolerant

You **sacrifice consistency**.

🖼️ 4. Applying CAP to the Diagram

Situation:

User A writes new data → USA server stores it.

BUT the USA → Europe replication link is broken.

Now Europe does **NOT** have the new data.

User B reads from the Europe server.

At this moment, the system must choose one of two behaviors:

🅰️ Option 1 — CP (Consistency + Partition Tolerance)

Europe server refuses to answer the read.

Why?

- It knows it might have stale data.
- To keep data **consistent**, it prefers to **fail** rather than lie.

What happens to the user?

User B sees:

❌ *Error: Data unavailable*
or
❌ *Server temporarily unavailable*

Good for:

- Banking
- Inventory (prevent overselling)
- Ticket booking
- Payments

Because you **must** read fresh data.

🅱️ Option 2 — AP (Availability + Partition Tolerance)

Europe server **answers the read**, even though the data is stale.

Why?

- System prioritizes availability.
- User always gets a response.
- Writes will replicate later, when the network heals.

What happens to the user?

User B sees slightly old data.

Good for:

- Social media feed
- Analytics dashboard
- Video recommendations
- News feed

Because **a few seconds of stale data doesn't hurt**.

❌ Why Is the Replication Arrow Crossed Out in the Diagram?

The red X means:

❗ **The link between USA and Europe is down → a partition has occurred.**

Now the system is forced to choose:

❓ Should Europe stop responding?
→ **Strong consistency (CP)**

❓ Or continue responding with maybe-stale data?
→ **High availability (AP)**

This is *exactly* the CAP theorem in action.

🧨 5. Real-Life Examples

✔️ Systems that choose AP (Availability)

- Instagram feed
- Twitter timeline
- YouTube recommendations
- Reddit comments
- Messaging "delivered" indicators

Why?

- Better for user experience.
- Temporary inconsistencies are acceptable.

✔️ Systems that choose CP (Consistency)

- Banking transactions
- Wallet balance
- Inventory stock

- Airline seat booking
- Payment systems

Why?

- Stale data can cause **money loss** or **double booking**.

🎯 6. What You Should Say in an Interview

Interviewers want to see that you understand **tradeoffs**.

When talking about replication or multi-region architecture:

Strong Answer For Social-Type Systems

> "Since the system must remain available even during cross-region network failures, we choose an AP model with eventual consistency."

Strong Answer For Financial or Inventory Systems

> "Because stale reads can cause double spending or overselling, the service chooses a CP model and rejects requests during partitions."

They love to hear **context-based decision making**.

🧵 Summary of the Diagram in One Sentence

The diagram shows a network partition between USA and Europe. You must choose whether Europe serves stale data (Availability) or stops serving requests (Consistency). This is the essence of the CAP theorem.

Numbers to Know

1. When to do calculations (and when NOT to)

They're saying:

- Don't start the interview doing random math like "1 billion users × 10 requests/day…"
- Instead, **only calculate when it helps make a decision**, for example:
    - "Do we need to shard the DB?"
    - "Can a single Redis instance handle our read load?"
    - "How many app servers do we need?"

Example (from the text):

> "We're expecting 50k requests/sec, each server can do ~5k, so 10 servers + headroom."

This is exactly the kind of simple calculation the interviewer wants:

- You **assume** a reasonable capacity per server.
- You **divide** and add some buffer.
- You **speak your reasoning** out loud.

The interviewer cares more about how you reason than about "perfect" numbers.

---

2. Modern hardware is stronger than you think

They're warning you: if you use *old / weak* numbers in your head, you will **over-design**:

- A well-tuned DB can handle **tens of thousands of queries per second**.
- A single Redis node can handle **hundreds of thousands of ops per second**.

- A single Postgres instance can handle **a few terabytes** of data comfortably.

So if you start sharding at **500 GB** or caching everything at tiny load, you're adding:

- **Complexity** (sharding, cache invalidation, distributed transactions)
- Without real need.

In interviews:
Only bring in **sharding / heavy caching / complex distributed systems** when your rough math shows a single node is actually becoming a problem.

---

3. Latency "ladder" – numbers to remember

These are the **latency levels** you should keep in your brain:

- **Memory access:** nanoseconds (super fast, basically free)
- **SSD read:** microseconds (still very fast)
- **Same–data-center network call:** ~1–10 ms
- **Cross-continent network call:** tens to hundreds of ms

How this helps in design:

- **Should we cache?**
  DB read: maybe 5–20 ms
  Redis: ~1 ms
  → If your endpoint is very latency-sensitive, cache helps a lot.
- **Should we do cross-region calls?**
  If crossing continents costs 100–200 ms RTT, and your SLA is 200 ms, you **cannot** call another continent synchronously on the hot path.

So in interview, you say things like:

- “Calling another region will add ~100–200 ms, that might break our latency SLO. Better to replicate data and serve locally.”
- “Cache hits are ~1 ms vs 20–50 ms for DB, so caching popular items makes sense.”

---

4. The table: what each component & “scale trigger” means

Let’s decode it row by row.

A) Caching

Key metrics (for something like Redis):

- ~1 ms latency
- 100k+ ops/sec per instance
- Memory-bound up to ~1 TB

Scale / redesign triggers:

- **Hit rate < 80%** → Most reads are still hitting DB → cache not effective; maybe:
    - Change what you cache
    - Increase TTL / capacity
    - Use better keys / caching strategy
- **Latency > 1 ms** → Node overloaded; consider:
    - More cache nodes (sharding)
    - Better network / config
- **Memory usage > 80%** → Evictions, thrashing; consider:
    - Bigger node or more nodes

- **Cache churn/thrashing** → Keys constantly evicted and reloaded; fix patterns, size, or LRU policy.

👉 In interviews:
"When cache hit rate drops below ~80% or cache memory is >80% full and we see evictions, we'd consider scaling out Redis or revisiting our cache strategy."

B) Databases

Key metrics (for a strong relational DB like Postgres/MySQL):

- Up to **50k transactions/sec**
- **Sub-5ms read latency** (when using buffer cache)
- **~64 TiB+ storage capacity** on a single instance (depends on infra)

Scale triggers:

- **Write throughput > 10k TPS**
    - Maybe start thinking:
        - Write sharding
        - Partitioning by user/tenant
        - CQRS, event sourcing, etc.
- **Read latency > 5 ms uncached**
    - Add read replicas
    - Better indexing, query tuning, or caching layer
- **Geographic distribution needs**
    - If users are global and latency matters → multi-region read replicas, or geo-partitioning, etc.

👉 In interviews:
"We'll stick to a single primary DB until we hit, say, >10k TPS writes or we're storing tens of TB and latency starts degrading; then we'd look at sharding or regional replicas."

C) App Servers

Key metrics (a typical web app server):

- **100k+ concurrent connections** (with async/event-driven I/O)
- **8–64 cores @ 2–4 GHz**
- **64–512 GB RAM**, can go to ~2 TB in big boxes

Scale triggers:

- **CPU > 70%** on average → system under consistent heavy load:
    - Scale out (more instances)
    - Optimize code, DB, caching
- **Response latency > SLA** → users see slow responses:
    - Scale horizontally
    - Fix slow dependencies (DB, cache, third-party calls)
- **Connections near 100k/instance**
    - Add more instances or use better connection handling.
- **Memory > 80%** → risk of OOM, GC issues, swapping.

👉 In interviews:
"We'd start with a small cluster of app servers and scale horizontally whenever CPU stays above ~70% and user-visible latency approaches our SLA."

D) Message Queues

Think Kafka / RabbitMQ / Pulsar-like systems.

Key metrics:

- Up to **1 million messages/sec per broker**
- **Sub-5ms end-to-end latency** (in ideal, local scenarios)
- Up to **50 TB storage** per cluster (varies, but this is ballpark)

Scale triggers:

- **Throughput near 800k msgs/sec** (close to 80% of 1M)
    - Add brokers
    - Add partitions
- **Partition count ~200k per cluster**
    - Too many partitions can hurt performance; split into more clusters or reorganize topics.
- **Growing consumer lag**
    - Consumers can't keep up
    - Add more consumers or rebalance partitions

👉 In interviews:
"If producer throughput approaches ~80% of broker capacity or we see increasing consumer lag, we'd add brokers and partitions, and scale consumers to catch up."

What should *you* actually memorize?

You don't need exact numbers; you need **ballpark ranges**:

- **Latency:**
    - Memory: ~nanoseconds

    - SSD: ~100 microseconds
    - Same-DC network: ~1–10 ms
    - Cross-region: ~100 ms
- **Cache (Redis-like):**
    - ~1 ms
    - 100k+ ops/sec
- **DB:**
    - Up to tens of thousands TPS
    - Single instance can store a few TBs+
- **App server:**
    - A few thousand RPS per server (you can assume 2–5k)
    - Scale when CPU > ~70% or latency > SLA
- **Message queue:**
    - Hundreds of thousands to ~1M msgs/sec per broker
    - Scale when approaching ~80% of that, or when consumer lag grows