**What is a Trade-off** (বিনিময়) **in System Design?**

A **trade-off** means that when you make one design choice, you often have to **sacrifice** or **compromise** on another aspect of the system.

👉 Because you **can’t optimize everything at the same time** (performance, scalability, cost, consistency, etc.), you must **balance competing factors**.

**📊 Common Trade-offs in System Design**

1. **Consistency vs Availability (CAP Theorem)**
   * If you want strong consistency, you may sacrifice availability during network partitions.
   * If you want high availability, you may accept eventual consistency.
2. **Latency vs Throughput**
   * A system optimized for very low latency (fast response) may not handle as many requests per second.
   * A high-throughput batch system (e.g., Kafka consumers) may introduce more latency.
3. **Read vs Write Optimization**
   * Denormalizing data improves read performance but makes writes more complex (must update multiple places).
   * Normalization makes writes clean but reads slower.
4. **Scalability vs Simplicity**
   * A single database is simple but doesn’t scale.
   * Sharding scales well but adds operational complexity.
5. **Caching vs Freshness**
   * Caching improves performance and reduces DB load.
   * But data may become stale (not real-time).
6. **Reliability vs Cost**
   * Replicating data across multiple regions improves reliability.
   * But it increases infrastructure costs significantly.
7. **Flexibility vs Performance**
   * A generic solution (e.g., SQL queries for everything) is flexible.
   * A specialized solution (e.g., precomputed indexes) is faster but less flexible.

**🔑 Example of a Trade-off**

**Scenario:** Designing a messaging app.

* If you store all messages in one DB: ✅ strong consistency, ❌ doesn’t scale.
* If you shard messages by user\_id: ✅ scalable, ❌ cross-user queries become complex.

👉 Here, you trade **simplicity and consistency** for **scalability**.

**🎯 Interview Tip**

When asked about system design:

* **Never say "this is the best solution."**
* Instead say: *“If we choose X, we gain A and B, but lose C. If we choose Y, we gain D but sacrifice E. Given the requirements, I’d lean toward X.”*

**System Design Trade-offs Cheat Sheet**

**1. Consistency vs Availability (CAP Theorem)**

* **Consistency**: All nodes see the same data at the same time.
* **Availability**: The system responds even if some nodes are down.
* **Trade-off**: Strong consistency = lower availability (e.g., Spanner). High availability = eventual consistency (e.g., DynamoDB, Cassandra).

**2. Latency vs Throughput**

* **Low Latency**: Fast response for single requests.
* **High Throughput**: Handle many requests per second.
* **Trade-off**: Can’t fully optimize both. Kafka → throughput optimized. Redis → latency optimized.

**3. Read Optimization vs Write Optimization**

* **Read-optimized**: Denormalization, caching, materialized views.
* **Write-optimized**: Normalized schema, fewer redundant writes.
* **Trade-off**: Faster reads = complex writes; clean writes = slower reads.

**4. Simplicity vs Scalability**

* **Simple**: Single database, monolith app → easy to manage.
* **Scalable**: Sharded DB, microservices → complex operations.
* **Trade-off**: Scalability adds operational overhead.

**5. Caching vs Freshness**

* **Caching**: Improves performance, reduces load.
* **Freshness**: Data always up-to-date.
* **Trade-off**: Cache may serve stale data → choose TTL or cache invalidation strategy.

**6. Reliability vs Cost**

* **Reliable**: Multi-region replication, backups, failovers.
* **Cheap**: Minimal redundancy, single region.
* **Trade-off**: Higher reliability = higher infrastructure costs.

**7. Flexibility vs Performance**

* **Flexible**: Generic SQL queries, dynamic schemas.
* **Performant**: Precomputed indexes, denormalized tables.
* **Trade-off**: Flexibility slows down performance; performance sacrifices flexibility.

**8. Strong Consistency vs Eventual Consistency**

* **Strong Consistency**: Correct data always (bank transactions).
* **Eventual Consistency**: System settles later (social media likes, counters).
* **Trade-off**: Strong consistency = slower writes. Eventual = faster but may show stale results.

**9. Availability vs Partition Tolerance**

* **Availability**: System always responds.
* **Partition Tolerance**: Works even with network issues.
* **Trade-off**: Under CAP theorem, in a partition you must pick either Consistency or Availability.

**10. Generalization vs Specialization**

* **Generalization**: One big service does everything (monolith, SQL DB).
* **Specialization**: Separate services/databases for different use cases (CQRS, Polyglot Persistence).
* **Trade-off**: General = simple, but less optimized. Specialized = fast, but more complex to maintain.