Contents

Data Storage & Management

This document covers the fundamental data storage and management concepts essential for system design interviews.

Components

SQL vs NoSQL Databases

- **SQL (Relational)**
  - **ACID compliance, complex queries, structured data, strong consistency**
  - **Use when:** Complex relationships, transactions, strong consistency required
  - **Examples:** PostgreSQL, MySQL, Oracle
- **NoSQL**
  - **Horizontal scaling, flexible schema, eventual consistency, high performance**
  - **Document:** MongoDB, CouchDB (JSON-like documents)
  - **Key-Value:** Redis, DynamoDB (simple key-value pairs)
  - **Column-Family:** Cassandra, HBase (wide column storage)
  - **Graph:** Neo4j, Amazon Neptune (relationships and connections)

Data Partitioning (Sharding)

- **Partitioning Types (How data is split)**
  - **Horizontal Partitioning:** Split rows across multiple databases
  - **Vertical Partitioning:** Split columns/tables across databases
  - **Functional Partitioning:** Split by feature/service boundaries - data is partitioned based on the functionality or business domain rather than data characteristics. Each partition contains all data related to a specific feature or service.
    - Example: An e-commerce platform might have separate databases for user management, product catalog, order processing, and payment processing. Each service owns its data completely.

    * Benefits: Clear ownership, easier to scale individual features, supports microservices architecture
    * Use cases: Large applications with distinct business domains, microservices architectures, teams organized by feature
  - **Hybrid Partitioning:** Combines both horizontal and vertical partitioning techniques to partition data into multiple shards. This technique can help optimize performance by distributing the data evenly across multiple servers, while also minimizing the amount of data that needs to be scanned.
    * Example: An e-commerce website might partition the customer table horizontally based on geographic location, then partition each shard vertically based on data type. When a customer logs in, the query can be directed to the appropriate shard, minimizing data scanning while enabling parallel processing across different database servers.
- **Partitioning Strategies/Techniques**
  - **Key/Hash-based Partitioning:** Apply a hash function to key attributes to determine partition number. For example, with 100 DB servers and numeric IDs, use 'ID % 100' to get the server number. Ensures uniform data allocation but fixes the total number of servers since adding new servers requires changing the hash function and redistributing data. Consistent Hashing solves this problem.
  - **List Partitioning:** Each partition is assigned a list of values. When inserting a new record, determine which partition contains the key and store it there. Example: All users from Iceland, Norway, Sweden, Finland, or Denmark stored in a Nordic countries partition.
  - **Round-robin Partitioning:** Very simple strategy ensuring uniform data distribution. With 'n' partitions, the 'i' tuple is assigned to partition (i mod n).
  - **Composite Partitioning:** Combines multiple partitioning schemes to create a new approach. Example: Apply list partitioning first, then hash-based partitioning. Consistent hashing can be considered a composite of hash and list partitioning where hash reduces key-space to a listable size.
  - **Consistent Hashing:** Distribute data evenly across nodes, minimize reshuffling when nodes added/removed. Maps data to points on a circle where only K/n keys need remapping (K=keys, n=nodes). Used by DynamoDB, Cassandra, Memcached clusters.

Indexes

- **Index Types**
  - **Primary Index:** Clustered index on primary key
  - **Secondary Index:** Non-clustered indexes for faster queries
  - **Composite Index:** Multi-column indexes for complex queries
- **Trade-offs**
  - **Benefits:** Faster reads, improved query performance
  - **Costs:** Slower writes, additional storage overhead, maintenance complexity

Replication

- **Replication Topology**
  - **Primary-Replica (Primary-Secondary):** One write node, multiple read replicas
  - **Peer-to-Peer (Multi-Primary):** Multiple write nodes, conflict resolution needed
- **Replication Timing**
  - **Synchronous:** Changes made to the primary database are immediately replicated to replica databases before the write operation is considered complete. Provides strong consistency but higher latency.
  - **Asynchronous:** Write operations complete on primary first, then replicate to replicas later. Better performance and eventual consistency.
  - **Semi-synchronous:** Combines elements of both synchronous and asynchronous replication. Typically waits for acknowledgment from at least one replica before completing the write operation, balancing consistency and performance.

Queues & Message Systems

- **Point-to-Point:** Direct queue between producer and consumer
- **Publish-Subscribe:** Broadcast to multiple subscribers
- **Message Brokers:** Apache Kafka, RabbitMQ, Amazon SQS
- **Use Cases:** Asynchronous processing, decoupling services, event streaming

Real-time Communication

- **Client-Server Communication Patterns**
  - **Short Polling:** Client repeatedly requests updates at regular intervals
    - Simple to implement, but inefficient (constant requests even when no updates)
    - High latency (up to polling interval), increased server load
    - Use case: Non-critical updates where slight delays are acceptable
  - **HTTP Long-Polling:** Client sends request, server holds it open until data is available or timeout
    - More efficient than short polling, near real-time updates
    - Maintains HTTP compatibility, works through firewalls/proxies
    - Use case: Chat applications, notifications, live comments
    - Example: Facebook Messenger, Gmail notifications
  - **WebSockets:** Full-duplex communication channel over a single TCP connection
    - True bidirectional real-time communication, low latency
    - Persistent connection, efficient for high-frequency updates
    - Use case: Gaming, collaborative editing, live trading platforms
    - Example: Google Docs, Slack real-time messaging, online games
  - **Server-Sent Events (SSE):** Server pushes data to client over HTTP connection
    - Unidirectional (server to client), automatic reconnection
    - Simpler than WebSockets, built-in browser support
    - Use case: Live feeds, stock prices, sports scores, progress updates
    - Example: Twitter live feeds, real-time dashboards

Related Trade-offs

SQL vs. NoSQL Databases

- **Summary:** SQL databases provide ACID compliance, complex queries, and strong consistency with structured schemas, while NoSQL databases offer horizontal scaling, flexible schemas, and high performance for specific use cases.
- **Trade-off:** Structured data integrity and complex querying vs. flexible schema and horizontal scalability.
- **Database Comparison:**
  - **SQL (Relational):** ACID transactions, complex joins, structured schema, mature ecosystem, but limited horizontal scaling and rigid schema changes
  - **NoSQL:** Horizontal scaling, flexible schema, high performance for specific patterns, but eventual consistency, limited complex queries, and newer ecosystem
  - **Hybrid Approach:** Use SQL for transactional data requiring consistency, NoSQL for analytics, caching, or high-volume simple queries
- **Questions to Ask:**
  - Do you need ACID transactions and strong consistency?
  - How complex are your data relationships and query requirements?
  - What are your scalability requirements (vertical vs horizontal)?
  - How frequently will your data schema change?
  - What's your team's expertise with different database technologies?
  - Do you need complex joins and analytical queries?
  - What are your performance requirements for reads vs writes?

Strong vs Eventual Consistency

- **Summary:** Strong consistency guarantees everyone sees the same data simultaneously, simplifying development and ensuring correctness at the cost of speed and availability. Eventual consistency allows temporary data differences across the system, boosting performance and fault tolerance at the cost of requiring applications to tolerate brief out-of-sync periods.
- **Trade-off:** Data accuracy and simplicity vs. performance, scalability, and availability.
- **Design Considerations:**
  - **Strong Consistency:** Use for banking ledgers, financial transactions, or systems where accuracy is non-negotiable
  - **Eventual Consistency:** Use for social feeds, content systems, or services that must stay available
  - **Hybrid Approach:** Apply strong consistency for core transactions, eventual consistency for derived or less critical data
- **Questions to Ask:**
  - What's the impact if users see stale data temporarily?
  - Will brief inconsistency harm the user experience or business?
  - How critical is each piece of data to core functionality?
  - Can the application logic handle temporary inconsistencies?

Distributed System Theorems: - **CAP Theorem:** During network partitions, choose between Consistency and Availability (Partition Tolerance required) - **PACELC Theorem:** Extends CAP - during Partitions choose Consistency vs Availability; Else choose Consistency vs Latency - **BASE vs ACID: - ACID (Traditional Databases):** Atomicity, Consistency, Isolation, Durability -

prioritizes strong consistency and reliability - **BASE (Distributed Systems):** Basically Available, Soft state, Eventual consistency - prioritizes availability and partition tolerance - **Relationship:** BASE emerged as an alternative to ACID for distributed systems where CAP theorem forces trade-offs. While ACID ensures immediate consistency, BASE accepts temporary inconsistency in favor of system availability and performance - **Use Cases:** ACID for financial transactions, BASE for social media feeds, content delivery, and large-scale web applications

Polling vs. Long-Polling vs. WebSockets vs. Webhooks

- **Summary:** Different approaches for real-time communication and event notification, each with distinct trade-offs between latency, resource usage, complexity, and reliability. The choice depends on update frequency, latency requirements, and system architecture.
- **Trade-off:** Simplicity and reliability vs. real-time performance and resource efficiency.
- **Communication Pattern Comparison:**
  - **Polling:** Client repeatedly requests updates. Simple, reliable, but inefficient with high latency
  - **Long-Polling:** Server holds request until data available. Better efficiency than polling, near real-time, but connection management complexity
  - **WebSockets:** Persistent bidirectional connection. Lowest latency, efficient for frequent updates, but complex connection management and resource overhead
  - **Webhooks:** Server pushes notifications to client endpoints. Event-driven, efficient, but requires public endpoints and retry logic
- **Questions to Ask:**
  - How frequently do updates occur and what's the acceptable latency?
  - Are updates bidirectional or server-to-client only?
  - Can clients maintain persistent connections or are they behind firewalls?
  - Do you need guaranteed delivery or is best-effort sufficient?
  - What's the tolerance for connection drops and reconnection complexity?
  - Are clients always online or do they connect intermittently?
  - How many concurrent connections do you need to support?
  - Do you need event ordering and exactly-once delivery?