SDC

Find a topic about system design to research and present in a video along with this research journal/wiki

What is system design?

Process of creating blueprints or architectural plan in order to effectively build complex software systems.

What is optimization in the context of software development?

optx in SD is the process of improving aspects of software systems to achieve increased performance, reliability, efficiency, security, and cost-effectiveness.

Why is optimization important?

Software optimization is critical in addressing the ever-evolving needs of consumers. As technology rapidly evolves, yesterday's solutions risk becoming obsolete. By optimizing software, we can improve aspects such as performance, reliability, cost-effectiveness, user experience, and scalability. Optimization is paramount in staying relevant when meeting the dynamic demands of consumers.

What are the key methods to increasing optimization?

Algorithm optx - algos are step-by-step procedures for solving problems. Choose most efficient algo and data structures

Example: sorted array (data structure) using binary search (algorithm) has lower time complexity that unsorted array using linear search, when looking for an element inside of an array, improving app perf.

Code refactor - restructure existing code to make cleaner, more readable and more efficient without changing external behavior

Example: implementing imperial iteration using for loop vs declarative reduce to find sum. While not computationally more optimized, reduce is more concise, expressive, and readable, making it more efficient for users to read and write, so refactor loop into reduce

Database optx - efficient db design and query optx crucial to interact with large volumes of data. organize data in a way that is easy and quick to find what you're looking for, like a library. techniques include indexing, denormalization, query optimization.

Example: indexing inc speed of data retrieval (search, filter, sort). An index is a data structure associated with a db table that allows the dbms to quickly locate rows based on values of one or more columns. So instead of scanning the entire table with a SELECT * query, the DBMS traverses the index data structure to find the relevant rows, reducing data retrieval time.

Caching - technique for storing frequently accessed and expensive data in temporary storage (such as memory or disk) for faster retrieval and performance. Cache memory can be located closer to CPU or at a higher level of the memory hierarchy, providing faster access times when compared to retrieving from the original source, such as a database, file system, or network.

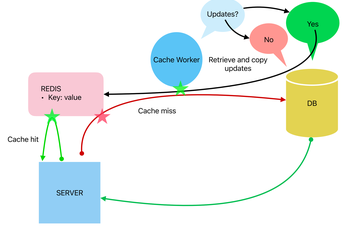
Example: weather service. Want Seattle weather. Scan whole weather data source is more expensive than already having a copy of seattle weather stored locally (cached) in a server. Now, instead of making request from external weather data source each time, can retrieve data from local server. if unavailable, THEN fetch from external weather data source. If each city weather is stored locally in each a server for a region or each city, then can access those first before going to external weather source. Consider network latency, reliability (downtime, outage of source), rate limits/quotas, cost (fees per fetch) for going over internet to fetch externally.

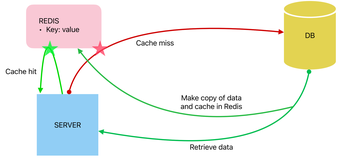
Why use caching?
- Subsequent requests for the same information
  - can be served faster, reducing latency and improving response times for users
  - Served with reduced resource consumption
    - **Reduced load** on underlying systems like db, file systems, network
      - **By minimizing # requests and computations for retrieving same data repeatedly**
        - So?
          - improves **efficiency** and **scalability**, allowing system to handle increasing users or requests without overloading system
    - Reduced cost
      - Esp for high traffic and resource intensive apps where each resource usage can be costly
        - db query, network bandwidth usage, computing resources in cloud environments
    - By off loading repetitive and resource intensive tasks from underlying architecture
      - So?
        - allow for flexibility with fluctuating demand, like web apps, APIs, services
        - handle more (requests, tasks, etc) WITHOUT sacrificing perf or reliability
  - Can be accessed offline
    - by storing freq accessed data locally on device or within app
    - So?
      - increase resilience and usability by continuing to provide service despite connectivity issues or being offline
- Summary: performance, scalability, cost-effectiveness
  - So?
    - provide service that is faster, more efficient, and reliable with superior UX

Redis
- Is a popular in-memory data store commonly used for caching in software apps
  - Why is it popular?
    - Speed, flexibility, simplicity, scalability, robust feature set
- in-memory data store allows fast read and write operations, suitable for caching freq accessed data that needs to be retrieved quickly
- key-value store
  - data stored and retrieved with unique keys
    - Easy to cache data from expensive computations or db queries against specific key
- expiration and eviction policies to manage cache size and prevent memory overflow
- data structures and operations to manipulate cached data
- options for persisting data on disk
- high perf and scalability, capable of handling millions of requests per sec with low latency
- versatile and flexible, not limited to caching
- robust and active community and ecosystem (client libraries, tools, integrations)

In-memory data store
- Stores data in computer memory (RAM) instead of on disk
  - Fast read and write
  - Reduced latency
  - Improving overall system perf

Other data storage types
- Disk-based
- Hybrid
- Cloud





Concurrency Ctrl - manage concurrent access to shared resources and prevent data corruption

Example: concurrent access and updates to bank account. If no controls or management in place, one's updates may overwrite another's. Financial loss. Techniques to control for this include lock, optimistic concurrency control, transaction management, and isolation levels.

Performance Monitoring and Profiling - id and address performance issues in apps. Monitor and profile key metrics to create most impact based on available resources. Use that info to optimize for better responsiveness, user experience, scalability.

Metrics - measurements used to evaluate perf. Ex) response time, throughput (num requests processed per sec), db query time, error rates, CPU and memory usage.

Instrumentation - adding code to collect metrics, such as log statements, built-in monitoring tools, integrating third party libraries.

Visualization - Make sense of performance data collected thru instrumentation. Graphs, dashboards. Interpret and id anomalies in data.

Why monitoring matters:
- Data driven decisions about optimizations and improvements
- Insight into how changes in code or infrastructure impact perf
- Id perf issues before they become critical

Performance monitoring: high level - observing and collecting information on behavior of software system while it is running. Like checking vitals.

Profiling: A step further than monitoring - dives deep into code to id how and where perf bottlenecks occur

CPU profiling - which parts of code consuming most CPU? pinpoint most impactful areas for change

Memory profiling - see how app uses memory. Id memory leaks, memory allocation patterns, excessive memory usage

I/O profiling - understand how I/O operations such as file I/O or network requests impact overall perf. These impacts can be optimized

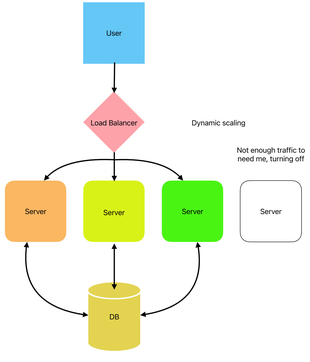
Why profiling matters:
- id specific areas of code for optx
- detailed insight into bottlenecks and their root causes
- allows prioritization of optx based on impact and resource usage

Load Testing and Scalability - load test simulate large # of concurrent users or requests, test perf under hvy load. Id bottlenecks and opt resource utilization to improve scalability for increasing future loads

Load balancing - as volume of access increases beyond capability of app server to handle, must scale vertically and/or horizontally. If scale horizontally, how does user know which server to connect to? Load balancer.

Ways to load balance
- Round robin
- Smart load balancing:
  - LB is aware of each server's load due to servers constantl sending load data to LB
  - LB makes decision on where to send user based on most recent server load data
  - software and setup can be expensive
- Use algo to distribute load (like random server select)
- many other methods to balance load...

Important because
- Scalability
- Reliability by providing redundancy (back-end servers), minimize downtime by detecting and replacing unhealthy servers
- Performance by dist workload evenly



Networking Optx

UI/UX optx

Security optx