Vertical Scaling

Vertical scaling, also know as scaling up, is the process of increasing the capacity of a single machine by adding more resources, such as:
1. CPU
2. Memory
3. Storage
4. Network bandwidth

How does Vertical scaling work?
When system starts experiencing performance bottleneck due to resource limitation, you can upgrade the existing hardware to boost its performance.
1. Moving application to more powerful server.
2. Adding components like more RAM or faster disk
3. Using virtualisation or cloud base solution like resizing VM instance.

Advantages:
1. Simpler Architecture:
Fewer components mean fewer complications, easier setup and simpler maintenance.
Ideal for applications that are inherently monolithic or not designed to scale horizontally.
2. Consistent Performance:
All processes run on the same machine, so there's no latency from network calls as in a distributed system.
3. Easier Management
There's only one server to monitor and manage, reduce the complexity of the distributed systems.

Challenges with Vertical Scaling.
1. Limitations:
Every machine has physical limit. This is known as vertical scaling ceiling.
Beyond a certain point, adding more resources doesn't significantly improve performance due to diminishing returns.
2. Downtime:
scaling vertically requires downtime for upgrades, which impact availability.
3. Cost:
High performance hardware are expensive.
Cloud instances which higher specs can also cost significantly more.
4. Single Point of Failure:
If the scaled-up machine fails, the entire system goes down unless there's a backup or failover mechanism.

Examples:
1. Database Scaling
Without changing code, user has previlage to increase RAM and processing power of CPU for database. That improves query performance.
2. Web Server Scaling
Upgrade server to higher configuration for black Friday sale.
Outcome: The server handles more concurrent suer without introducing load balancer or additional servers.

When to sure Vertical scaling:
1. Small to Medium application.
2. Quick fix
3. Cloud environments(does not involve significant downtime)

Key Points to Emphasise in System Design Interviews:
1. Trade-off:
1. Explain advantages(Simplicity) and limitations(single point of failure)
2. Highlight that it's often a stepping stone before transitioning to horizontal scaling.
2. Use cases:
1. Mention scenarios where vertical scaling is ideal.)where scaling horizontal is too complex.
3. Integration
1. Combine vertical scaling with other optimisation like caching, database indexing or query optimisation.

Scenario
Online Quiz Platform
You are tasked with designing a simple online quiz platform. Users can register, log in, and take quizzes. The platform initially supports a small user base but is expected to grow as more schools and organizations adopt the system.

Key Requirements:
1. Handle up to 10,000 daily active users at launch, scaling to 100,000 users within a year.
2. Provide low latency for quiz interactions.
3. Ensure high availability and reliable data storage for user records and quiz scores.
4. The budget is limited during the initial phase.

Question 1:

How would you design the platform to handle the current and expected load using vertical scaling?
Take a moment to think and outline your initial thoughts:
- What components (e.g., web server, database) would you include in the platform?
- How would you configure them to handle the load at launch (10,000 daily active users)?
- How would you vertically scale these components as the platform grows?

Analysis of Your Approach:
1. Initial Setup for 10,000 Daily Active Users (DAU):
- A single server with 16 GB RAM and a decent clock speed (e.g., AWS T2 instance) should handle the initial load for both the application and web server.
- Using AWS Aurora MySQL provides managed database services with high reliability and automated backups, perfectly suited for this use case.

2. Vertical Scaling Plan:
- Upgrade the application server (e.g., from T2.medium to M5.large or even C5.xlarge) as the load increases.
- Similarly, upgrade the database server instance size and adjust storage IOPS as necessary.
- Use snapshots of EBS volumes to migrate seamlessly to larger machines with minimal downtime.

3. Downtime Minimization Techniques:
- Blue-Green Deployment:
- Deploy updates to a new environment while keeping the old one live.
- Redirect traffic to the updated environment only after testing.
- Throttling:
- Limit the number of simultaneous user connections during migration.
- Provide informative messages (e.g., "Scheduled Maintenance" alerts).

Suggested Enhancements to Your Strategy:
1. Load Estimation for 10,000 DAU:
- Assume an average user session generates ~50 requests/day.
- With 10,000 DAU, the server must handle 500,000 requests per day (~5.8 requests/second). A T2.medium should suffice, but keep CPU credits in mind if the server experiences spikes.

2. Database Optimization:
- Add read replicas to offload heavy SELECT queries if necessary.
- Optimize queries and use caching (e.g., Redis or Memcached) for frequently accessed data like quiz results.

3. Scaling Timeline:
- Plan upgrades proactively:
- Upgrade to 32 GB RAM and C5.xlarge as soon as DAU exceeds 20,000.
- Monitor CPU, memory, and I/O metrics via CloudWatch or equivalent tools to trigger scaling decisions.

4. Additional AWS Tools:
- Consider using Elastic Load Balancers (ELB) as traffic grows to distribute the load effectively, even with a vertically scaled backend.
- Use RDS Aurora's auto-scaling capabilities for storage and instance resizing.

Let's dive deeper into each topic one by one. Which one would you like to start with:
1. Calculating server/database capacity
2. Minimizing downtime techniques
3. Optimizing costs

1. Calculating Server/Database Capacity
Capacity planning involves estimating how many requests per second (RPS) your system needs to handle, the load on your database, and the resources required to maintain performance.
Step-by-Step Process:
1. Estimate Traffic:
- Active Users per Day (DAU): 10,000 at launch.
- Requests per User: Assume each user generates ~50 requests/day (login, fetch quiz, submit answers, etc.).
- Total Requests per Day: Total RPS=(DAU×Requests per User)/Seconds in a Day Total RPS=Seconds in a DayDAU×Requests per User Total RPS =(10,000×50)/(10^5)   5 requests/second

Peak Traffic Estimate: During active quiz times, traffic spikes. Assume 10x peak factor for 1 hour of high activity:
1. Peak RPS=5×10=50 requests/second Peak RPS=5×10=50 requests/second.
1. Application Server Needs:
- A single T2.medium instance on AWS (~2 vCPUs, 4 GB RAM) can handle ~100 RPS under a typical load for a simple web app.
- With your estimated peak RPS of 50, this setup will comfortably handle your launch requirements.

2. Database Server Needs:
- Query Estimation: Assume that each request generates 1–2 database queries. Total Queries = 2× Total RPS=2×50=100 queries/second 2×Total RPS=2×50=100 queries/second.
- Read vs. Write Load:
- 80% of queries are reads (fetching quizzes or results).
- 20% are writes (saving quiz submissions).
- A T2.medium RDS instance with MySQL/Aurora can handle ~100 write IOPS and several thousand reads/sec. Optimize with:
- Indexes for write operations.
- Read replicas for scaling reads as DAU grows.

3. Network Bandwidth:
- Assume each request/response is ~100 KB.
- Bandwidth = 50×100 KB=5MBps 50×100KB=5MBps. Any AWS instance with moderate networking (~10 Gbps) can handle this easily at launch.

2. Minimizing Downtime Techniques
Blue-Green Deployment:
- Maintain two environments:
- Blue: Current production.
- Green: Newly scaled machine or instance.
- Steps:

1. Clone the database to the green environment.
2. Deploy updated code or scale resources in the green environment.
3. Test the green environment thoroughly.
4. Switch traffic to green with DNS or load balancer updates.

Read Replicas and Maintenance Windows:
- Use read replicas to offload traffic during upgrades. Writes will queue briefly during database upgrades, but reads remain unaffected.
- Schedule maintenance during off-peak hours (e.g., midnight local time).

Auto-Failover Solutions:
- Configure AWS RDS Aurora for automatic failover during instance resizing or maintenance.

3. Optimizing Costs
For Initial Setup:
- Use T2.medium or T3.medium instances for both app and database servers. These are cost-effective for low traffic.
- Opt for on-demand instances to avoid upfront commitments.

As Traffic Grows:
- Transition to Reserved Instances (e.g., 1-year or 3-year plans) once usage stabilizes to save up to 60% compared to on-demand costs.
- Leverage Savings Plans for flexible cost reductions across instance families.

Long-Term Optimizations:
- Use Elastic Cache (Redis or Memcached) for frequently accessed data like quiz questions and user sessions. This reduces database load and saves costs on larger DB instances.
- Consider spot instances for batch processing tasks like generating analytics.

Key Takeaways
- A T2.medium or T3.medium server is sufficient for the initial phase. Plan upgrades (e.g., M5.large or C5.xlarge) based on observed growth.
- Use techniques like blue-green deployments, read replicas, and maintenance windows to minimize downtime.
- Start small and leverage AWS's reserved and spot pricing models to reduce long-term costs.

1. Optimizing Specific Database Queries
1. Query Design Optimisation : Avoid Select *, Use where clauses effectively.
2. Indexing Strategies
1. Primary key indexes
2. Composite Indexes
3. Covering indexes
3. QueryExecution Analysis
4. Denormalization
5. Query Caching
6. Use Pagination
7. Database Connection Pooling

2. App- Leve Caching
1. Caching frequency accessed data
1. Type of Data to cache
1. Static Data
2. Session Data
3. Computed Result(Like leaderboard)
2. Tools for App-level caching
1. In-Memory cache(Redis or memcached)
2. Local Application Cache(app-level caching libraries)
3. Cache Strategies
1. Read-Through Cache(populate cache when a database read occurs, if cache is empty, query the database)
2. Write-Through Cache(Update cache immediately when the database is updated)
3. Cache Expiry:(Set TTL for cached items)
4. Lazy Loading(Cache-aside) (load data into cache only when request misses, then update it)
5. Cache invalidation(Ensure cache is invalidated when underlying data changes)
4. Benefits:
1. Reduce database query load significantly for frequently accessed data.
2. Provide near-instant response times for cached requests.
3. Combine Query Optimisation and Caching
1. Hybrid approach ensure maximum efficiency:
1. Optimize critical database queries for performance.
2. Use caching to offload repetitive queries for static and semi-static data.

Question 2: What are the potential bottlenecks in your design?

Application Server Bottlenecks
- CPU Overload: High CPU usage due to traffic spikes or inefficient code (e.g., unoptimized loops, lack of concurrency).
- Memory Exhaustion:
- Applications with large in-memory data structures (e.g., quiz question cache) may run out of RAM.
- Memory leaks can also exacerbate the issue over time.
- Network Bandwidth Saturation: A surge in quiz submissions or large file uploads (e.g., images or results exports) might exceed network capacity.
- Single Point of Failure: With only one application server, if it crashes, the entire platform goes down.

2. Database Bottlenecks
- High Write Load:
- Frequent quiz submissions (e.g., 20% of requests) can lead to disk I/O saturation.
- Writes locking rows can cause contention, slowing down read operations.
- Read Contention:
- Heavy read queries (e.g., loading quizzes or leaderboard data) can overwhelm the database if caching is not used.
- Lack of Indexing: Missing or poorly designed indexes can lead to full table scans, slowing down queries significantly.
- Scaling Limits:
- A vertically scaled database can only handle so much load before hitting resource limits (CPU, memory, or disk IOPS).

3. Caching Bottlenecks
- Cache Misses: High rates of cache misses can overload the database. For example, if quiz data is not pre-warmed in the cache, sudden quiz spikes may hit the DB.
- Stale Data: Without proper invalidation, outdated quiz results or leaderboard data might be served to users.
- Memory Limits in Caching Layer:
- Redis/Memcached servers might run out of memory for large datasets.
- Poor eviction policies (e.g., not prioritizing hot data) may reduce cache effectiveness.

User Authentication Bottlenecks
- Session Storage: If user sessions are stored in memory, memory exhaustion can occur during traffic surges.
- Authentication API: A surge in login requests might overwhelm the server if rate limiting or throttling isn't in place.

Deployment and Scaling Bottlenecks
- Downtime During Scaling: Even with careful vertical scaling (e.g., resizing instances), some downtime may occur, impacting user experience.
- Migration Bottlenecks:
- Moving from one instance size to another might introduce latency or require freezing writes temporarily.
- Blue-Green deployments can reduce this risk but might require additional resources temporarily.

Cost Constraints
- Scaling up to larger instances (e.g., from T2.medium to C5.xlarge) increases costs rapidly, potentially exceeding the budget, especially for storage-heavy workloads.

Addressing These Bottlenecks
1. Application Bottlenecks:
- Use asynchronous processing (e.g., worker queues) for non-critical tasks like leaderboard updates.
- Monitor CPU, memory, and network usage with tools like AWS CloudWatch and set alarms for early warnings.

2. Database Bottlenecks:
- Use read replicas to distribute read traffic.
- Partition data if quiz results grow significantly (e.g., sharding by quiz_id or user_id).
- Optimize queries and use connection pooling to reduce DB load.

3. Caching Bottlenecks:
- Increase cache memory as DAU grows and tune eviction policies for optimal performance.
- Pre-load hot data (e.g., quizzes) into the cache during low-traffic hours.
- Use distributed caching for better fault tolerance and scaling.

4. Authentication Bottlenecks:
- Implement rate limiting for login attempts to prevent server overload.
- Store sessions in Redis or a similar scalable datastore rather than in-memory.

5. Scaling Downtime:
- Use auto-scaling groups to automate instance upgrades with minimal manual intervention.
- Ensure backups and snapshots are current to recover quickly if an upgrade fails.

6. Cost Optimization:
- Use Reserved Instances or Savings Plans for predictable workloads.
- Offload expensive operations like analytics to batch processes during off-peak hours.

Question 3: How would you mitigate risks of failure due to vertical scaling limitations?

1. Mitigating Single Points of Failure
Solution: High Availability Setup
- Use Load Balancers (even for vertically scaled systems) to distribute traffic and provide failover.
- Set up a failover instance:
- Keep a secondary server running on standby.
- Configure automated failover using tools like AWS Elastic IP or Route 53 health checks.

Example for Database:
- AWS Aurora RDS provides automatic failover for primary database instances. When the primary instance fails, the secondary takes over without manual intervention.

2. Managing Resource Exhaustion
Solution: Proactive Monitoring and Auto-Scaling
- Monitor CPU, memory, and disk usage with tools like AWS CloudWatch or Datadog.
- Set up alerts for thresholds (e.g., 80% memory utilization) to trigger pre-emptive upgrades.
- Use auto-scaling for vertical resources:
- Many cloud providers (AWS, Azure) allow dynamic resizing of instances during peak usage.

Plan for Upgrades:
- Transition from small to larger instances (e.g., T2.medium   C5.xlarge) as usage grows.
- For databases, increase IOPS on SSD storage to match increased write operations.

3. Handling Downtime During Scaling
Solution: Blue-Green Deployments
- Prepare a parallel (green) environment with upgraded resources.
- Perform thorough testing on the green environment.
- Gradually shift traffic from the old (blue) environment to the green one.

Solution: Maintenance Windows
- Schedule upgrades during off-peak hours.
- Notify users in advance with clear maintenance schedules.

4. Balancing Cost and Performance
Solution: Reserve Resources and Use Spot Instances
- Use Reserved Instances for predictable workloads to reduce costs.
- For non-critical operations (e.g., analytics), use spot instances to save money.

Optimize Resource Allocation:
- Scale up only the components under strain. For example:
- Increase memory for the database to handle larger caches.
- Use more CPU cores for the application server if user traffic grows.

5. Implementing Horizontal Scaling as a Fallback
While focusing on vertical scaling, it's crucial to design the system to pivot to horizontal scaling if required.
Hybrid Scaling Strategy:
- Add multiple instances behind a load balancer if vertical scaling reaches its limits.
- Scale read traffic by adding database replicas to distribute the load.

Example:
- When the primary application server cannot handle traffic, deploy additional servers to handle requests without disrupting the existing architecture.

6. Database-Specific Failure Mitigation
Solution: Multi-AZ Deployments
- In AWS, use Multi-AZ (Availability Zone) configurations for RDS instances.
- Keeps a synchronized copy of the database in another AZ, enabling automatic failover during failures.

Solution: Backups and Rollbacks
- Automate database snapshots and backups.
- Implement rollbacks for data corruption scenarios.

7. Caching to Offload Workload
Solution: Use Distributed Caching
- Leverage tools like Redis Cluster for fault-tolerant, scalable caching.
- If a cache node fails, the cluster ensures continued availability.

Pre-Warming Caches:
- Populate caches with frequently accessed data during low-traffic periods to prevent database strain after restarts.

8. Educating and Preparing the Team
Solution: Disaster Recovery and Incident Response Plan
- Train the team to handle scaling issues and failures.
- Test disaster recovery scenarios periodically to ensure smooth recovery during outages.

Summary of Failure Mitigation
1. Use redundancy (failover instances, Multi-AZ deployments) to handle single points of failure.
2. Monitor resources and scale preemptively to avoid exhaustion.
3. Employ techniques like blue-green deployments and read replicas to reduce downtime and distribute load.
4. Introduce caching and horizontal scaling as fallback strategies for long-term growth.

Simulation: Applying Failure Mitigation Strategies

Scenario:
An online quiz platform is scaling to serve 100,000 daily active users (DAU). The system has been vertically scaled to a C5.xlarge instance for the application and a MySQL RDS instance with 32 GB RAM for the database. Recently, it encountered performance degradation and a temporary outage during a high-traffic event.
Your task: Apply failure mitigation strategies to improve resilience and scalability.

Step 1: Analyze Current Bottlenecks
- CPU Spikes: Application server (C5.xlarge) maxed out CPU during peak traffic (~500 requests/second).
- Database Latency: Query response times increased due to frequent writes and insufficient indexing.
- Downtime During Scaling: A manual instance resizing caused a 15-minute outage.
- Cache Misses: Redis cache was not pre-warmed, causing excessive database load after restart.

Step 2: Apply Mitigation Strategies
1. Application Server: Handle CPU Spikes
- Monitor Traffic Patterns: Use AWS CloudWatch to track CPU utilization and set alerts at 80% usage.
- Auto-Scaling:
- Introduce an auto-scaling group to dynamically adjust instance size or count based on traffic.
- Add a secondary instance during peak traffic, even though you're primarily vertically scaling.
- Background Tasks:
- Offload leaderboard updates to a worker queue (e.g., using AWS SQS + Lambda).
- Hybrid Scaling:
- Introduce a load balancer (AWS ALB) to prepare for a hybrid vertical + horizontal scaling approach.

2. Database: Reduce Latency
- Optimize Queries:
- Analyze slow queries with EXPLAIN and add necessary indexes (e.g., on quiz_id and user_id).
- Read-Replica Setup:
- Add a read replica for MySQL RDS to handle 80% of the traffic (read-heavy queries).
- Configure the application to route read traffic to the replica while writes go to the primary database.
- Connection Pooling:
- Use HikariCP (or equivalent) to limit simultaneous database connections and reduce contention.

3. Downtime During Scaling: Eliminate Outages
- Blue-Green Deployment:
- Clone the existing instance and scale the clone to a higher tier (e.g., M5.2xlarge).
- Redirect traffic to the upgraded instance once it's tested.
- Multi-AZ Deployment:
- Enable Multi-AZ in RDS to ensure automatic failover to a secondary database during outages.

4. Cache: Prevent Misses and Reduce Database Load
- Pre-Warm Redis Cache:
- Preload frequently accessed data (e.g., active quizzes, user sessions) during low-traffic hours.
- Cache Key Example: quiz:questions:<quiz_id> for quiz data.
- Set TTL and Eviction Policies:
- Use Redis's allkeys-lru policy to evict least recently used keys when memory fills.

Step 3: Disaster Recovery Plan
Database Backups:
- Automate daily snapshots of RDS with point-in-time recovery enabled.

Failover Instance:
- Configure a secondary standby application instance (e.g., another C5.xlarge in a different availability zone).
- Use Route 53 health checks to route traffic to the standby server if the primary fails.

Regular DR Testing:
- Simulate outages and failover scenarios every quarter to ensure preparedness.

Step 4: Monitor and Scale Dynamically
Resource Monitoring:
- CPU, memory, and disk usage with AWS CloudWatch.
- Database query times and IOPS to detect hotspots.

Traffic-Based Scaling:
- Use metrics like requests per second (RPS) and cache hit rates to predict resource needs.
- Automatically resize the application server or database instance during predicted peaks (e.g., exam season).

Summary of Mitigations
1. Application: Auto-scaling, load balancers, and worker queues for non-critical tasks.
2. Database: Query optimization, read replicas, and Multi-AZ failover for reliability.
3. Cache: Pre-warm Redis, implement LRU eviction, and cache TTLs to reduce database load.
4. Downtime: Blue-green deployments and maintenance windows for seamless scaling.
5. Disaster Recovery: Automated backups, failover instances, and regular testing.

Simulation: Handling a Sudden Traffic Spike

Scenario
Your online quiz platform has been improved with the failure mitigation strategies. During a major event (e.g., a national quiz competition), traffic suddenly spikes from 500 requests per second (RPS) to 2,000 RPS over a short period. Let's simulate how the system handles this spike with the implemented improvements.

1. Trafic Spike Begins: 500 -> 2000 RPS
Application Scaling Triggers:
1. Auto-scaling Triggers
2. Asynchronous Processing

Outcome:
Application servers maintain stable performance, with latency below 50ms for user requests.

2. Database Response:
Read-Replica Handling
Write Optimizations

Outcome: Database handles ~1600 read queries per second via the replica and ~400 write queries per second on the primary instance without significant delays.

3. Caching Layer Performance
Redis Cache Handling
Cache quiz metadata
Pre-warmed cache
Redis eviction policies

Outcome: Cache hit rate remains high reducing the load on the database and ensuring consistent response times for users.

4. Monitoring and Alerts
Cloudwatch alerts:
Alert for high CPU usage and increased IOPS are sent, notifing the engineering team.

Scaling Adjustment:

Outcome: Proactive monitornig prevents system crashes, and the platform scales dynamically to meet user demand.

5. Traffic Spike End: 2000 -> 500 RPS
Auto-scaling cooldown
Cache Eviction and clean up

Outcome: Resources scale back to normal levels, saving costs while maintaining performance.

Lesson Learned from Simulation

1. Autoscaling ensured application availability by dynamically adding resource as traffic increased.
2. Database Replication and optimized queries distributed the load effectively, preventing a primary database bottleneck.
3. Redis Cache significantly reduced latency and database load, handling the majority of repeated requests.
4. Monitoring and Alerts provided real-time visibility into system performance, enabling proactive scaling.
5. Cost Efficiency: By scaling back after spike, resources were optimized to save cost.