Performance Engineering Real Time – Troubleshooting Interview Questions

1. How do you troubleshoot a CPU spike issue in production?

Articulate:

First, we need to identify which process is consuming high CPU using **top or htop**. Once identified, we drill down further using **ps -eo pid,%cpu,cmd --sort=-%cpu | head** to check the top-consuming processes. If it's a Java application, the next step is to analyze the threads causing the high CPU using **jstack <pid>.** Capturing multiple thread dumps at intervals helps identify looping threads or high CPU-consuming transactions. Additionally, using **pidstat -p <pid> 1** allows us to check which threads are consuming excessive CPU. If necessary, we analyze CPU profiling data with dynatrace or Jprofiler. If a specific method is taking longer, we optimize the code or tune JVM settings accordingly.

Approach:

1. **Check CPU utilization using** *top, htop, vmstat, or mpstat.*
2. **Identify the process consuming high CPU** *(ps -eo pid,ppid,cmd,%cpu --sort=-%cpu | head).*
3. **Analyze thread-level CPU consumption using** *pidstat -p <pid> -t 1.*
4. **Check Java applications** (jstack <pid> to analyze thread dumps).
5. **Check garbage collection (GC) issues if using** JVM (jstat -gcutil <pid> 1s).
6. **Profile the application** using Dynatrace,JProfier
7. **Check for infinite loops, memory leaks, or inefficient code.**

2. How do you investigate high memory usage in an application?

Articulate :

First, we need to check the total memory consumption using free -m or top. If memory is consistently high, we identify which process is consuming the most memory using ps aux --sort=-%mem | head. For Java applications, we use jmap -heap <pid> to check heap utilization. If excessive memory consumption is observed, we take a heap dump using jmap -dump:format=b,file=heapdump.hprof <pid>.

This heap dump is analyzed in Eclipse MAT to check for memory leaks or excessive object retention. If necessary, we optimize object allocations in the code or tune JVM heap settings (-Xms, -Xmx).

Additionally, we check swap usage (vmstat 1) to ensure the system is not swapping, which can impact performance.

The next step is code profiling using VisualVM/Dynatrace tool to identify potential memory leaks or inefficient object retention.

Approach:

1. **Check memory usage** using *free -m, vmstat, or top.*
2. **Identify process consuming high memory** *(ps aux --sort=-%mem | head).*
3. **For Java apps, check heap usage** *(jmap -heap <pid>, jstat -gc <pid>).*
4. **Capture heap dump** *(jmap -dump:live,format=b,file=heapdump.hprof <pid>).*
5. **Analyze heap dump** using *Eclipse MAT or jhat.*
6. **Check for memory leaks** with VisualVM, YourKit, or Dynatrace.
7. **Monitor swap usage** (*vmstat 1 to see if swapping occurs).*
8. **Profile the application** *(valgrind, heaptrack, pmap -x <pid>).*

3. A Java application is experiencing frequent OutOfMemoryErrors. How would you handle it?

First, we need to check the application logs for OutOfMemoryError occurrences and determine the heap space exhaustion type (heap, metaspace, or native memory). We then enable GC logging **(-XX:+PrintGCDetails -XX:+PrintGCDateStamps)** to check garbage collection behavior. Using **jmap -heap <pid>,** we analyze the heap allocation.

If a memory leak is suspected, we generate a heap dump with **jmap -dump:format=b,file=heap.hprof <pid>** and analyze it using Eclipse MAT to identify memory leaks. If necessary, we increase heap size (**-Xmx and -Xms)** or optimize object allocation in the application code. Additionally, we monitor GC behavior using **jstat -gcutil <pid> 1s** and, if needed, fine-tune GC parameters.

Approach:

1. **Check GC logs** (-XX:+PrintGCDetails -XX:+PrintGCDateStamps).
2. **Analyze heap usage** (jmap -heap <pid>).
3. **Generate a heap dump** (jmap -dump:format=b,file=heap.hprof <pid>).
4. **Analyze the heap dump** using Eclipse MAT to find memory leaks.
5. **Monitor active threads** (jstack <pid> to check blocked threads).
6. **Increase heap size (if necessary)** (-Xmx and -Xms JVM options).
7. **Optimize application code** by fixing memory leaks.

4,What are common causes of high CPU utilization in Java applications, and how do you troubleshoot it?

Common causes:

- **Inefficient loops or recursion**
- **Thread contention** (Locks, Deadlocks)
- **Garbage Collection (GC) overhead**
- **Excessive logging**
- **Inefficient SQL queries causing high DB calls**

5. How do you analyze a Java application's garbage collection (GC) behavior?

First, we enable GC logging with **-XX:+PrintGCDetails -XX:+PrintGCDateStamps** to capture detailed garbage collection events. Using jstat -gcutil <pid> 1s, we monitor heap utilization and GC frequency. If Full GC is happening too frequently, we analyze memory allocation patterns using jmap -heap <pid>.

If necessary, we capture a heap dump **(jmap -dump:format=b,file=heap.hprof <pid>)** and analyze it in Eclipse MAT to identify excessive object retention. If the issue is due to excessive object creation, we optimize the code to reuse objects or introduce pooling mechanisms. Additionally, if heap space is insufficient, we fine-tune JVM memory settings **(-Xmx, -Xms)** and select an appropriate GC algorithm **(G1GC, ZGC).**

Technique:

1. **Enable GC logging** (-XX:+PrintGCDetails -XX:+PrintGCDateStamps).
2. **Monitor GC behavior** (jstat -gcutil <pid> 1s).
3. **Analyze GC logs** using gceasy.io or GCViewer.
4. **Check heap utilization** (jmap -heap <pid>).
5. **Adjust GC tuning parameters** (-XX:MaxHeapFreeRatio, -Xms, -Xmx).
6. **Optimize object creation** to reduce GC overhead.

6.What steps do you follow when troubleshooting slow database queries?

First, we need to identify the slow queries causing response time latency using Dynatrace, Oracle AWR reports, or SQL profiles. Once slow-running queries are identified, we analyze the execution path using EXPLAIN ANALYZE to check whether indexes are being used effectively. If missing indexes are found, we suggest creating indexes to optimize retrieval. For large tables, we consider partitioning strategies to improve performance. Additionally, tuning connection pooling settings can help reduce total response time. It's also important to monitor disk I/O using iostat to rule out storage latency.

Approach:

1. **Identify slow queries** (SHOW FULL PROCESSLIST or pg_stat_activity).
2. **Check query execution plan** (EXPLAIN ANALYZE).

3. **Optimize indexes** (CREATE INDEX based on query patterns).
4. **Partition large tables** if necessary.
5. **Optimize connection pooling** (HikariCP, c3p0).
6. **Tune database parameters** (innodb_buffer_pool_size, work_mem).
7. **Check disk I/O performance** (iostat, sar -d).

7. How do you handle thread contention in a multi-threaded Java application?

First, we capture thread dumps (jstack <pid>) at multiple intervals to check for **blocked or waiting threads**. If excessive thread blocking is observed, we analyze **synchronized blocks** and optimize them by switching to **ReadWriteLock** or **ConcurrentHashMap** instead of using synchronized collections. If too many threads are causing CPU contention, we analyze thread pool configurations (ExecutorService) and optimize the thread count based on available CPU cores. Using **Java Flight Recorder (JFR)** helps identify which threads are consuming the most CPU time.

8.How do you troubleshoot high disk I/O in an application?

First, we use iostat -x 1 to check disk utilization and latency. If disk I/O is consistently high, we identify the processes causing the issue using iotop. If a database is involved, we analyze slow queries (EXPLAIN ANALYZE) and optimize indexing or query execution strategies. We also check if log files are growing rapidly (**du -sh /var/log**) and rotate them if necessary. If disk I/O spikes during batch jobs, we consider optimizing batch processing or using asynchronous I/O mechanisms. In virtualized environments, we check if storage contention exists by monitoring **sar -d**. If necessary, we move frequently accessed data to in-memory storage (Redis, Memcached) to reduce disk dependency.

9. How do you handle performance bottlenecks in a Kubernetes environment?

First, we check application performance using Kubernetes metrics (**kubectl top pods).** If CPU or memory limits are being hit, we increase resource requests and limits (**resources.limits.cpu/memory).** If pods are being throttled, we analyze node-level resource utilization using kubectl top nodes. If network latency is suspected, we use kubectl exec to run ping and traceroute tests between pods. If the application is experiencing high load, we enable Horizontal Pod Autoscaler (HPA) to scale dynamically. Additionally, we check if persistent volumes are causing I/O bottlenecks using **kubectl get pvc** and optimize storage class settings if necessary.

10. How do you reduce cold start time in a serverless function?

First, we analyze **cold start logs** using cloud monitoring tools like AWS CloudWatch or GCP Stackdriver. If cold starts are frequent, we increase the **provisioned concurrency** (AWS Lambda) to keep functions warm. If the function has high startup latency, we optimize **dependency loading** using dependency bundling tools (esbuild for JavaScript, GraalVM for Java). We also reduce function execution time by optimizing **I/O operations and database connections** (e.g., using connection pooling). If necessary, we refactor large functions into **smaller, modular functions** to improve response times.

11. How do you optimize database connection pooling?

First, we monitor **connection pool metrics** using tools like dynatrace. If the pool is exhausted, we increase **maxPoolSize** while ensuring it aligns with database capacity. If connections are frequently being created/destroyed, we **enable keep-alive settings** to reuse idle connections. Using **connection validation queries** ensures connections are not broken (SELECT 1). Additionally, tuning **idle connection timeout** helps free up unused connections to avoid unnecessary memory consumption.

12. How do you investigate high disk I/O in a Kubernetes cluster?

First, we monitor disk I/O using iostat -x 1 and check persistent volume (kubectl get pvc) utilization. If I/O latency is high, we analyze which pods are performing heavy disk reads/writes using kubectl top pods. If the issue is database-related, we optimize query indexing and enable **read replicas**. If logs are causing high disk writes, we implement **log rotation and compression**. If the issue is with persistent storage, we consider **moving to SSD-based storage** for improved performance.

13. How do you troubleshoot high network latency in an application?

First, we check network response times using ping and traceroute to identify where the delay is occurring. If the issue is between application components, we use **tcpdump or Wireshark** to analyze packet flow. For microservices, we inspect **service-to-service communication latency** using distributed tracing tools like **Jaeger or Dynatrace**. If slow DNS resolution is suspected, we test DNS lookup times (dig <hostname>). Based on findings, we optimize network routing, enable **persistent connections (Keep-Alive)**, or implement **gRPC** for lower-latency communication.

14. How do you investigate slow application startup times?

First, we check logs for long-running initialization tasks. If the application is a Java-based service, we use **-XX:+PrintGCApplicationStoppedTime** to identify if GC pauses are delaying startup. If dependency loading is slow, we optimize classpath scanning and use **lazy initialization** where possible. Profiling with **JProfiler or YourKit** helps identify slow method calls during startup. If containerized, we check if the image size is too large and optimize **Docker layers**. If database connections are slow to establish, we enable **connection pooling**.

15. How do you troubleshoot Kubernetes pod crashes due to OutOfMemory (OOM)?

First, we check pod logs (kubectl logs <pod>). If OOM errors are reported, we inspect **container memory usage** (kubectl top pods). If memory limits are too restrictive, we increase resources.limits.memory. If Java is running inside the container, we adjust JVM heap (-Xmx should be below container memory limit). If a memory leak is suspected, we take heap dumps

from the running container **(kubectl exec -it <pod> -- jmap -dump:format=b,file=heap.hprof <pid>).** If scaling is needed, we configure **Horizontal Pod Autoscaler (HPA)**.

16. How do you analyze slow API response times in a multi-region deployment?

First, we analyze latency using **APM tools** and correlate with region-specific performance metrics. If cross-region network latency is observed, we optimize request routing via **CDNs or Global Load Balancers (GCP, AWS Route 53)**. If database queries take longer in specific regions, we introduce **read replicas** closer to users. If API calls involve excessive serialization overhead, we switch to **efficient formats (gRPC, Avro)**.