1. In object storage, if we use a “simple” hashing algorithm, to add/remove a storage device would lead to high overhead (i.e., data will be moved around devices). The consistent hashing algorithm is then used to solve the problem. Please explain how the consistent hashing algorithm works.

Ans: In consistent hashing algorithm, while adding/removing a new drive, each drive will get a new range of hash values it is going to store. However, each object’s hash value will remain the same. If an object’s hash value is within the new range, it will remain in that drive only, therefore no need to move the data to other drive. For other objects who are not in the new range, we have to map them to the new drive i.e. we have to move the objects to new drive as the new range. This has the advantage over basic hashing function that the number of objects to be moved are very few.

2. What is “dual I/O stacks” in storage virtualization? What is the consequence of dual I/O stacks? What are the solutions you can come up to mitigate this problem?

Ans: Dual I/O stacks are actually two level I/O stacks that resides in individual machine i.e. in each virtual machine as well as in the native host (hypervisor). The major problems with dual I/O stack are, because there are two I/O stacks, it makes the IO path longer so it increases the time taken to fetch the information. Another problem is that because there are two IO stacks, it makes the path longer to fetch the information i.e. long latency. There are 3 possible solutions to these problems:

i. Simplify IO stack in the VM by using thin para-virtualization layer which is also called as front IO end. This can directly communicate with IO so it helps in eliminating multiple IO layers.
ii. Shorten the IO path by either creating a short path between a virtual machine front-end and emulator or completely placing the emulator inside a native host. We can insert emulator inside a native kernel.
iii. Leverage hardware technology assistance such as VTD or SRLV. These designate one device or part of a device using virtual functions to specific VMs. It by-passes the whole native host and can achieve same performance as that in native case. Because VM can directly access underlined designated IO devices directly without going through IO stack.

3. Why do we need a “byte-addressable” storage stack in cloud block storage system? What problem does tradition block-addressable storage cause in cloud block storage system, and how does the byte-addressable storage stack solve them?

Ans: There is a data granularity mismatch between the storage layers on the client side and the network layer between client and the storage server side. This granularity mismatch is because storage layer is block addressable and the network layer is byte-addressable. But the actual cloud storage devices are byte-addressable. Therefore, we need byte-addressable storage stack in cloud block storage system. There are following 2 problems caused due to traditional block-addressable storage devices:

i. Non-aligned write blocking: if addressable block size is perfect integer with the byte size of multiple of actual block size, the block is fetched from the page cache. This is in the case of well-aligned writes. But in non-aligned writes, the page is not available in the cache. So, it is fetched from the storage back-end and put in the cache. This operation is time consuming.
ii. Transmission Amplification: While flushing data from page cache, the whole page is flushed out even if only few bytes within that page are dirtied.

To overcome these problems, the byte-addressable storage stack implements 3 different mechanisms:

i. BASS_Cache: it keeps the track of dirtied pages at the level of byte granularity. It means that, the cache keeps the record of which bytes are dirtied within a particular page.
ii. BASS_Block: this is a generic block layer. It makes it possible to generate arbitrary length IO requests. As the requests are of arbitrary length, it facilitates transfer of only needed data even if its size is not exactly equal to the actual page size.

iii. BASS_Backend: it conducts a block level alignment to provide correctly accessible block adjustable storage devices. Basically, it makes the storage devices compatible with byte-addressable architecture.

4. There is a saying that "every write for SSD" is a random write. Do you agree or not? Why?

Ans: I agree with the saying because of the following reason:

As there are no moving mechanical parts in SSD for read/write operations, rather information is stored in microchips or NAND gate-based memory units. SSD is organized into blocks of 64 or 128 pages to store the data and read/write operation is done by per-page basis.

SSD consists of embedded processor or controller which makes the decision on how to store the data on disk. Basically, SSD works in a similar way to that of memory management works in terms of primary memory. It uses page table mechanism but with actual storage disk. Therefore, the data being written on SSD is written randomly and not contiguously. Hence, I agree to the saying that every write for SSD is a random write.

5. Why are the pros and cons of object storage in comparison with block storage? Especially, what makes objects storage much scalable?

Ans: **<u>Block-storage:</u>**

- It is faster, used to store hot data which is frequently changed like file systems.
- It is more IO centric with lot of optimization both from hardware side like HDD and SSD and also from software side like performance optimization for VMs.
- Data is store without any concept of data format or type. In fact, it is stored in series of 0s and 1s.
- It is a high-level applications or file systems to keep track of data location, context and its meaning.

<u>Object-storage:</u>

- It can be bigger
- It can be scaled out by adding multiple new nodes
- It is ideal for achieving large amount of data that are not frequently accessed
- It is and object-based solution where we manage and manipulate data as objects
- It is software centric
- It is made of object identifier, data and metadata
- All objects are stored in a flat structure without complex object organization. Thus, objects can be accessed directly without traversing hierarchical directories.