**Knowledge and Theory: System Design and Scalability**

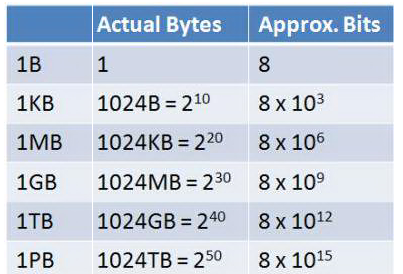
**Part 1 - Useful Knowledge for System Design and Scalability:**

**Quantifying Real World Objects:**

* Your generally given a “real world” description of entities like “10,000 Linked In profiles” or “1 million URL’s”, “500 720P Videos that are <= 20 minutes”. You should be able to quantify these “real world” entities very quickly. This will give you an idea of whether you can store this in one machine or will require several machines.

|  |  |
| --- | --- |
| **Real World Entity** | **# of Bits** |
| **Text** | # of bits = (8 bits per char) × (number of chars in the text)   * There is 8 bits per char because there are 256 unique chars defined by ASCII and 28 = 256 proves you can distinguish each with 8 bits. |
| **Image** | # of bits = (bits per pixel) × (number of pixels in the image).   * The BPI is determine by the file type * If an image’s quality is 100 × 100, it contains 10,000 pixels. |
| **Video** | # of bits = (bits per pixel) × (pixels per frame) × (frames per second) × (seconds in the video)   * The BPI is determine by the file type * The number of pixel per frame is determine by the video quality * If a videos quality is 720P each frame has 1280x720 pixels. |

**Quantifying Storage Sizes:**

****

**# of Machines:** When someone describes what information the application or website site will store, you should quantify this information in bits, and then quantifying typical machine storages sizes in bits, and then figure out how many machines you’ll need for this application.

**Hashing Machines:** Create data structures that can map/map requested information (usually a url link for page, photo, social media profile, video) to machine, and location on machine. For example, for instagram, youtube, wikipedia or facebook, lets there be V nodes where each profile, youtube video, wikipedia page, or facebook profile is a node. Let each node has a node ID that ranges from 0 – V. Have a hash method called “get machine (node id)” and “get location(node id)”.

**Question 1 –** Assume that you have some information about stocks and there were business clients (approximately 1000 of them) who wanted to look up this information somehow. How would you design the client-facing service?

* The variables that matter are user friendlyness for the clients, implementation difficulty, software flexibility, maintenance, efficiency and scalability.

**Option 1 – SQL Database:**

* The Pros is that its easy to facilitate querying, strong library, documentation, available tools, easy to back up, easy to integrate in any language or application.
* The cons is it maybe overkill in size if the clients only need to look things up. Its going to require developers to maintain it, server costs, server may be down, etc.

**Option 2 – Distribute XML Files:**

* Have a program generate an XML file that contains the important variables that the client needs and could be read by any browser. The pros is quick, easy and convenient.
* The cons is the clients can’t queury and must take all the information.

**Question 2 -** How would you design the data structures for a very large social network like Facebook or Linkedin? Describe how you would design an algorithm to show the connection,

or path, between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

* Create a graph type data structure with edge lists or adjacent nodes. The nodes are profiles and the adjacent nodes are friends.
* To find connections between two people, use breadth first search to traverse from one person to another.
* We can’t store each profile (assuming there’s 100s of millions of profiles) on one machine. Each profile/node needs an Id. Let’s assume there are V nodes in the graph and each one has an ID from 0 – V respectively. Then we should have a mapping function that maps machines to ID’s. So given an ID, it returns the machine and some information about location.
* Ideally, there should be a server class that maps ID’s to machines and holds this map one machine. This class could have various way’s to provide the machine of the profile your looking for. For example, link, name, ID, etc.
* So now, each person stores the ID’s of friends only in their friends list.
* Try to organize people and machines so that you don’t have to keep jumping machines when you look for a few of your friends. Perhaps, organize machines by cities.

**Question 3 -** Given an input file with four billion non-negative integers, provide an algorithm

to generate an integer which is not contained in the file. Assume you have 1 GB of

memory available for this task.

* Whenever a question gives you two pieces of information (K = # of distinct values) and ( L = # of available bits) and “coincedentally” L > K always think use a bit vector. What they are hinting is that you could have a bit vector of size K with indices 0 – K for each possible distinct value. Each bit could keep track of something useful.
* In this particular question, they did not specify K but by saying the numbers are integers, K is 232 = 4 billion. L is 1 GB = 4 billion bits. L is coincedentally greater than K.
* So what should we keep track of? We should keep track of whether an integer we are passing has been seen before.
* Initialize a sequence of 4 billion bit vectors all initialize to zero.
* Iterate all the #’s in the array and for each number, set(number as index, 1 to set the bit) in the bit vector.
* Then iterate through the bit vector and return the first zero value.

**Question 4 -** You have an array with all the numbers from 1 to N, where N is at most 32,000. The

array may have duplicate entries and you do not know what N is. With only 4 kilobytes

of memory available, how would you print all duplicate elements in the array?

* Whenever a question gives you two pieces of information (K = # of distinct values) and ( L = # of available bits) and “coincedentally” L > K always think use a bit vector. What they are hinting is that you could have a bit vector of size K with indices 0 – K for each possible distinct value. Each bit could keep track of something useful.
* In this particular case, we want to keep track of whether we have seen this item to know whether it is a duplicate.
* You could iterate around the array and check if each item in the array has been seen. If it has, print it. If it hasn’t, set its index in the bit vector.