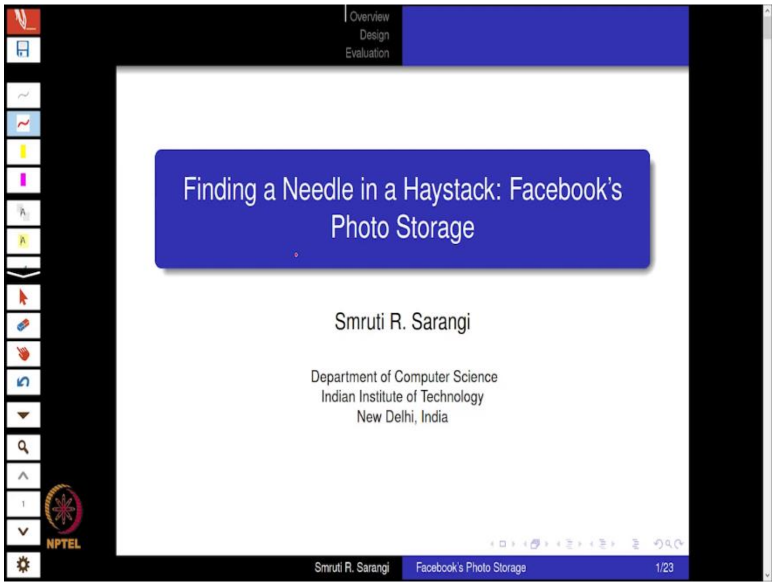
Advanced Distributed System
Professor Smruti R. Sarangi
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi
Lecture 22
Facebook's Photo Storage System

(Refer Slide Time: 00:17)



Finding a Needle in a Haystack: Facebook's Photo Storage

Smruti R. Sarangi

Department of Computer Science
Indian Institute of Technology
New Delhi, India

So, in this lecture, we will discuss Facebook's photo storage, the title of the paper is a needle in a haystack. So, a haystack is a large storage. A haystack is essentially a stack of hay and the needle is a very small needle, which is very hard to find if it is inside a haystack. So, in this case, the haystack is the photo storage and the needle is a single photo, which needle as to say is very hard to find.

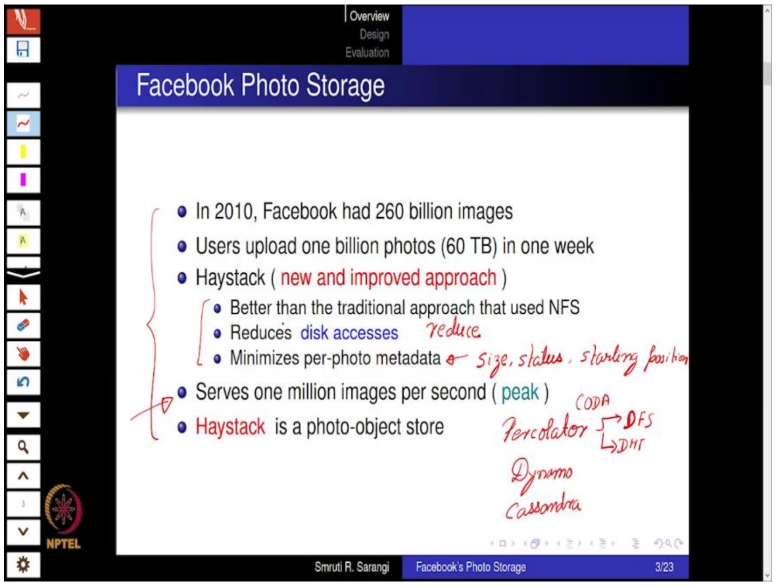
(Refer Slide Time: 00:50)

Overview
Design
Evaluation

Outline

1. Overview
2. Design
3. Evaluation

Smruti R. Sarangi   Facebook's Photo Storage   2/23



Overview
Design
Evaluation

Facebook Photo Storage

- In 2010, Facebook had 260 billion images
- Users upload one billion photos (60 TB) in one week
- Haystack (new and improved approach)
  - Better than the traditional approach that used NFS
  - Reduces disk accesses reduce
  - Minimizes per-photo metadata ← size, status, starting position
- Serves one million images per second (peak)
- Haystack is a photo-object store

CODA
Percolator → DFS, → DHT
Dynamo
Cassandra

Smruti R. Sarangi   Facebook's Photo Storage   3/23

So, we will discuss an overview of the approach, and then the design and the evaluation. So, this paper is around 10 years old, it is slightly dated. So, as of 2010, also, Facebook was very big. It had 260 billion images, which was very, very large for that time, even for now as well. Users would upload a billion photos a week, which would roughly be the order of 60 terabytes and the haystack system, which was a new and improved approach.

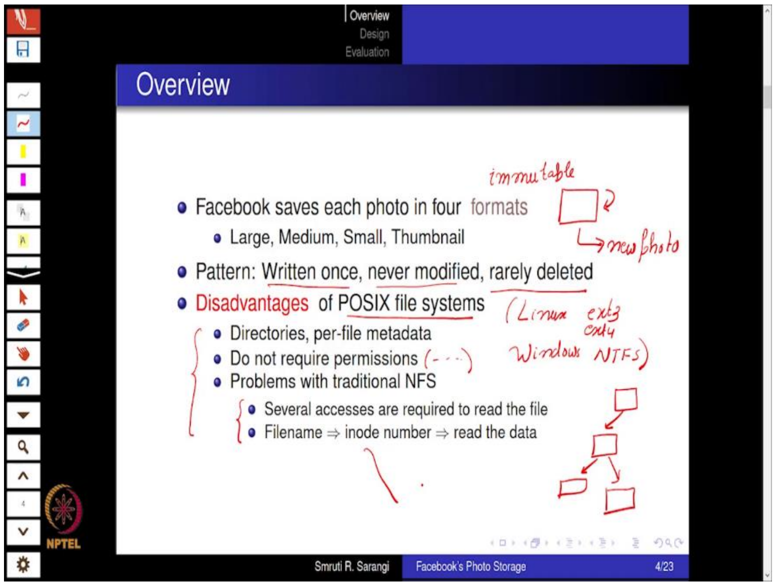
In this approach, this was replacing the traditional approach which was used the network file system NFS. This reduced the number of disk accesses. Because an approach the main problem with a traditional file system like NFS was the increased number of disk accesses. So, this is something that the aim was to explicitly reduce.

And also, the all the aim was to minimize per photo metadata. Because we would keep some data for the photo as well as some meta data to essentially identify, so, the main thing it would contain is the size and the status of the photo. And of course, something like its starting position on the disk. So, the starting position, the size and the status, we will come to what the status is later in the lecture. All these three things the size, the status and the starting position of a photo. All these are essentially things that need to be minimized.

And also, we need to serve a million images per second, which was Facebook's peak serving rate those days. So, that is the reason they had to create a novel custom bespoke solution a photo object store essentially a store only for photos. So, this was known as haystack and haystack of course, is in the line of things that we have been since studying over now. So, we have looked at a few projects that essentially use the same elements.

So, we have looked at percolated Google percolator, which is built on Bigtable. So, essentially, there is a distributed file system and there is a DHT, some former you have also seen CODA, which is a distributed file system. And then we have discussed dynamo, which is, again, a one-hop DHT. And then we discussed Cassandra, which was for the inbox search problem. So, of course, in the inbox search, we are looking at small messages not large photos. So, if a viewer were to ask why we cannot use Cassandra to store photos, well, Cassandra is not meant for that. Cassandra, the main aim of Facebook, Cassandra was to store small messages, not large photographs, that was not the aim in this case it is.

(Refer Slide Time: 04:31)



Overview

- Facebook saves each photo in four formats
  - Large, Medium, Small, Thumbnail
- Pattern: Written once, never modified, rarely deleted
- Disadvantages of POSIX file systems
  - Directories, per-file metadata
  - Do not require permissions
  - Problems with traditional NFS
    - Several accesses are required to read the file
    - Filename ⇒ inode number ⇒ read the data

So, Facebook saves each photo in 4 formats. So, the formats are large, medium, small, and thumbnail. So, these 4, so essentially, every photo that we create, it is stored in these 4 formats. And let us say we apply, some sort of a transition to a photo like we rotate it, then essentially, we create a new photo.

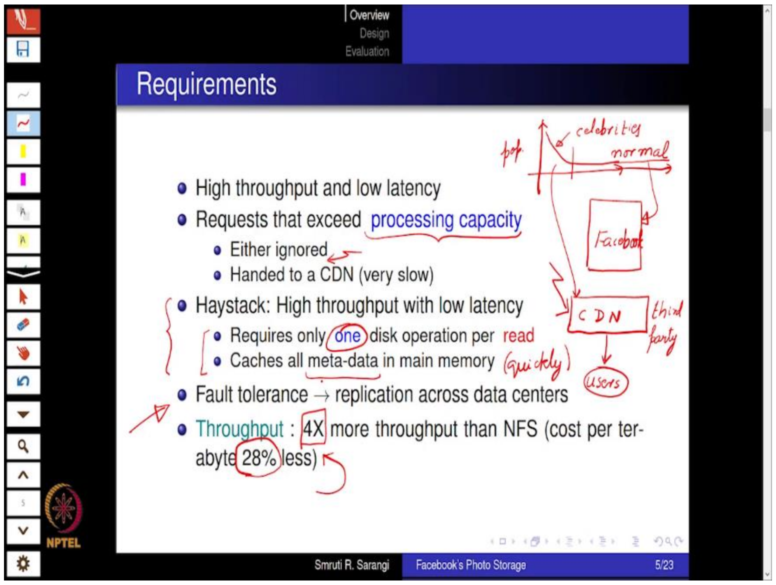
So, similar to our previous assumptions, we assume that data is immutable. Immutable means that it does not change. And so which means that we write once and read many times. And if I were to create a new version of it, like rotate it so, essentially, that is like uploading a new photo, which is the same pattern it is written once never modified. And also, it is rarely deleted, in the sense that we very rarely delete photos from Facebook. Unless a photo has been uploaded and error, we typically do not delete photos.

So, any POSIX file system, what is a POSIX file system? Well, the Linux and Windows file system, so Linux, ext3, ext4. So, the Linux and Windows file systems are these file systems are POSIX compliant file systems. So, they have a lot of overheads, it is okay for a general file system but it is not the best solution for specific for something for storing a specific kind of data specific type of data.

So, what adds to the overhead? Well, we have directories that adds to the overhead in the sense that the file systems support a very hierarchical system. So, that adds, kind of adds to the overhead.

The further thing is that we store a lot of additional data, like permissions, and so on, which are not required, writes of storing photos, we do not require directories, we do not require permissions. There are problems with traditional NFS distributed version of a POSIX file system in a sense, we need several accesses to read a file not one, several. So, given the file, we have to retrieve the inode number by traversing the directory tree then read the data. Even if we do caching, it does not help a lot in reducing disk accesses.

(Refer Slide Time: 07:36)



Overview
Design
Evaluation

Requirements

- High throughput and low latency
- Requests that exceed processing capacity
  - Either ignored
  - Handed to a CDN (very slow)
- Haystack: High throughput with low latency
  - Requires only one disk operation per read
  - Caches all meta-data in main memory (quickly)
- Fault tolerance → replication across data centers
- Throughput : 4X more throughput than NFS (cost per terabyte 28% less)

Smruti R. Sarangi Facebook's Photo Storage 5/23

So, the requirements that we have for a photo object store, the requirements are high throughput and low latency, it is not high throughput and high latency as was the case and other approaches, which was like a more of a batch processing. So, in that case, you need high throughput and high latency, when you typically do not care how long it takes. But this is not the aim here. The aim here is a very high throughput and a low latency. And so basically, for that, we have to essentially create a very tiered system that would allow this.

So, first, we define the certain processing capacity, which means a certain quality of service for a server, which essentially talks about the number of maximum number of requests that can be served per cycle. So, if you are exceeding this, you can ignore messages. Not a good idea. Or what we can do is we can have Facebook system. And we can have something like Corona or Akamai a content distribution network.

So, what the CDN does is the CDN is a set of third-party servers, which essentially buffers Facebook data, some of the popular data, and it serves that to the users. So, in this case, what the CDN actually does is that it acts like a cache for Facebook. It takes popular data and it serves it to the users. And so, the CDN is a third party offering and so in this case, whatever is popular can be served. But the main problem and you would also see this in the paper that if I were to plot kind of the popularity on the y axis and order the Facebook pages on the x axis the graph would be something like this.

So, let us say the you will have a set of very popular Facebook pages, mainly for celebrities, politicians. But for most people, most normal users and mind you, this will be a massive tail, you really have a long, long heavy tail of just regular normal users. So, their photographs will not be extremely popular. As a result, the caching strategy of the CDN will not really help. So, it will essentially help for the celebrity part. But for the regular normal users, we need to use the default Facebook system.

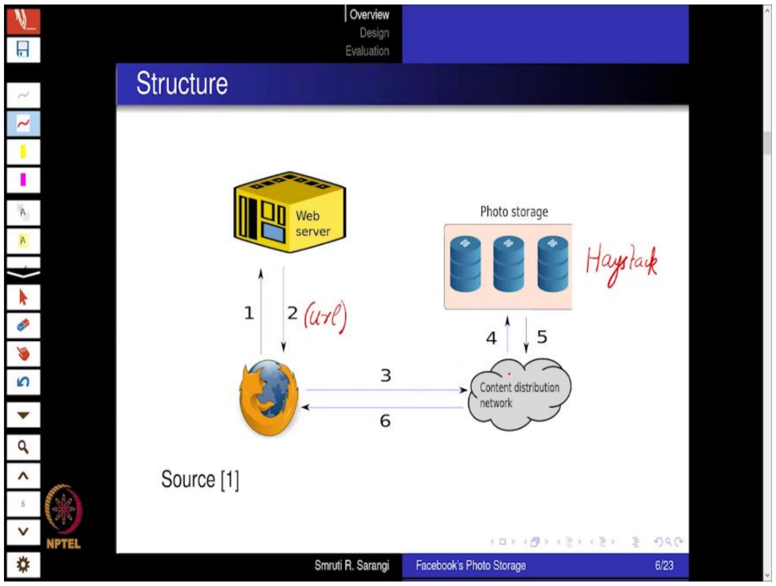
So, this is the two layered architecture or this is the two layered structure of Facebook and CDN where the CDN is third party, as I said, like Corona, or Akamai, which serves the more popular pages, Facebook pages, of course, but we need Facebook's indigenous haystack system for the rest of the requests and they have to be served with a high throughput and a low latency.

We have some additional requirements as well, that we only want one disk operation per read, not many, not more than one. And also, another recurrent pattern that we have been seeing is that all the metadata of a file that needs to be stored in main memory. So, instead, the meta data, at least can be accessed very quickly. So, the meta data gets stored in the main memory cache and the rest on disk. But there also we are allowed a single access.

Fault tolerance. Well, this is also a recurrent pattern that we have seen. We saw that in Amazon Dynamo where the replication was across data centers. So, this is required, because if one data center fails, we would like another data center to take over. Furthermore, even within a data center, we would like to have replication across racks. So, it is not a one set of racks fail another set of racks can take over.

So, with all of these modifications, haystack registers a 4X throughput improvement over baseline NFS. And the cost per terabyte, the cost per terabyte of data serving a terabyte of data is 28 percent less, which is substantial for a server farm. So, now we need to understand where these numbers come from and what exactly did Facebook do to get such a large improvement in throughput, and also simultaneously at a lower cost.

(Refer Slide Time: 12:51)

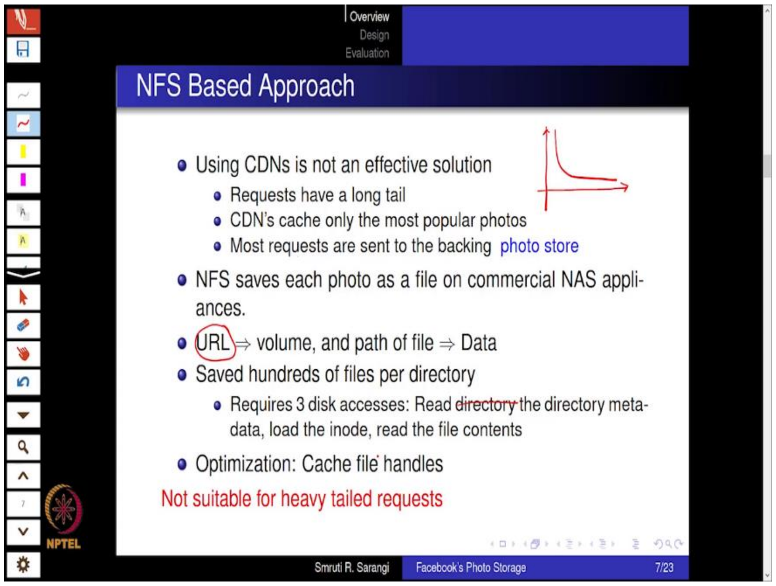


Structure

Source [1]

So, this is the typical flow of kind of a simplified, simplistic Facebook system. So, in this case, the web browser actually requests the web server for a photo, or for a set of photos on a page. The web server replies, but it does not reply with the photo, it replies with a kind of internal Facebook URL. The internal Facebook URL is sent to the CDN, which acts as a cache for the photo storage it acts for a cache for haystack. If CDN has the data it returns it otherwise, it forwards the request to the photo storage which then retrieves the photo from the haystack storage and returns it via the CDN. Some of the popular photos may be cached by the CDN for later users.

(Refer Slide Time: 13:54)



NFS Based Approach

- Using CDNs is not an effective solution
  - Requests have a long tail
  - CDN's cache only the most popular photos
  - Most requests are sent to the backing photo store
- NFS saves each photo as a file on commercial NAS appliances.
- URL ⇒ volume, and path of file ⇒ Data
- Saved hundreds of files per directory
  - Requires 3 disk accesses: Read ~~directory~~ the directory metadata, load the inode, read the file contents
- Optimization: Cache file handles

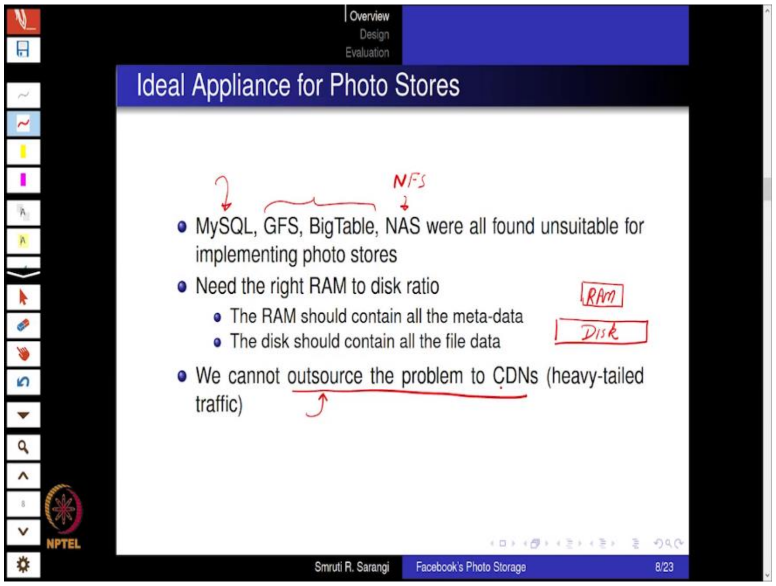
Not suitable for heavy tailed requests

So, let us now look at the default approach that uses a distributed file system NFS. So, as we have discussed using CDNs content distribution networks is not an effective solution. The primary reason being that requests have a very long tail the popularity distribution has a very long tail and the CDN caches only the most popular photos. So, most requests are anyway sent back to the backing photo store which in this case is haystack.

Furthermore, what NFS would do is that it would save each photo as a file on a commercial NAS appliance. So, NAS appliance or network storage appliance where the storage device is connected on the network and is accessible over the net. So, what we are essentially doing is that in this case, give in the URL, it is being mapped to some sort of a storage volume and the data is coming from the path of the file. Of course, we can do a default implementation where we save hundreds of files per directory, to kind of minimize the depth of the directory tree. This would require 3 disk accesses read the directory metadata.

So, this is the first access node, node of the file and read the file contents. So, the optimization is that we can cache file handles. So, caching file handles does not really help because of the heavy tailed nature. So, this is not very helpful. And the reason is that for this range, which is very, very long, caching does not work.

(Refer Slide Time: 15:55)



Ideal Appliance for Photo Stores

- MySQL, GFS, BigTable, NAS were all found unsuitable for implementing photo stores
- Need the right RAM to disk ratio
  - The RAM should contain all the meta-data
  - The disk should contain all the file data
- We cannot outsource the problem to CDNs (heavy-tailed traffic)

There are a few competing technologies like MySQL, well, this is MySQL is meant for relational data. It is a relational database. The data that we are looking at here is not of a relational character, hence, it is not useful. We have looked at the combination of GFS and Bigtable in Google percolator.

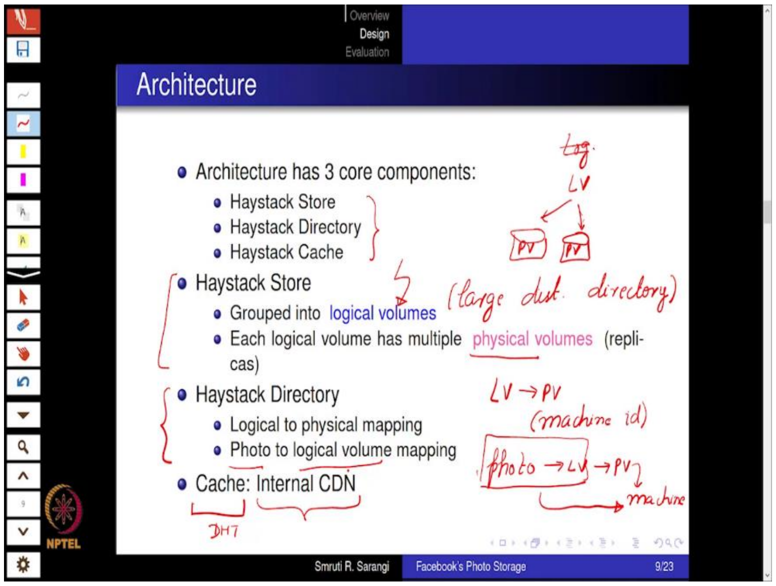
So, in GFS, we divide a file into chunks, we replicate the chunks and stored them on different servers. But in this case, we are dealing at the granularity of photos. So, this is not a very scalable approach plus GFS tries to give us kind of a general file system, which is not what we want.

And big table is again, built on top of GFS that takes a large multi dimensional table, splits it into small chunks, replicates those chunks. Again, this is not the best solution for us, because our data is not a multi dimensional table. And we have already looked at the shortcomings of traditional NFS on NAS storage devices.

So, what we actually want to do is that we want to create a new kind of file organization that uses elements that we have seen in the past. So, we will have we will store all the metadata of the file, which includes the its position on the disk, and its size and its status flag. So, this in some sort of easily searchable form an index in RAM and then actual file will be stored the actual photograph will be stored on disk.

So, of course, we would like the RAM to disk ratio to be as high as possible in the sense more the RAM the better it is. Of course, there are substantial cost implications with a larger RAM particularly in a very large data center. So, the aim here is to find the right mix the right balance such that we can minimize the cost for creating the data center as well as kind of maximize the RAM to disk ratio right at the same time. And as we have been discussing several times the problem for serving users Facebook pages cannot be outsourced to CDNs. So, for the Facebook pages of celebrities, yes, but not for regular users because of the heavy tail pattern.

(Refer Slide Time: 18:33)



Architecture

- Architecture has 3 core components:
  - Haystack Store
  - Haystack Directory
  - Haystack Cache
- Haystack Store
  - Grouped into logical volumes
  - Each logical volume has multiple physical volumes (replicas)
- Haystack Directory
  - Logical to physical mapping
  - Photo to logical volume mapping
- Cache: Internal CDN

So, the architecture of haystack has 3 components store, a directory and a cache. So, we will discuss them in turn. So, the store is the actual photo store that stores all the photographs. So, what we do is that the entire photo store which is of course a distributed store is grouped into logical volumes, think of this as a large directory, a large distributed. That is a logical volume.

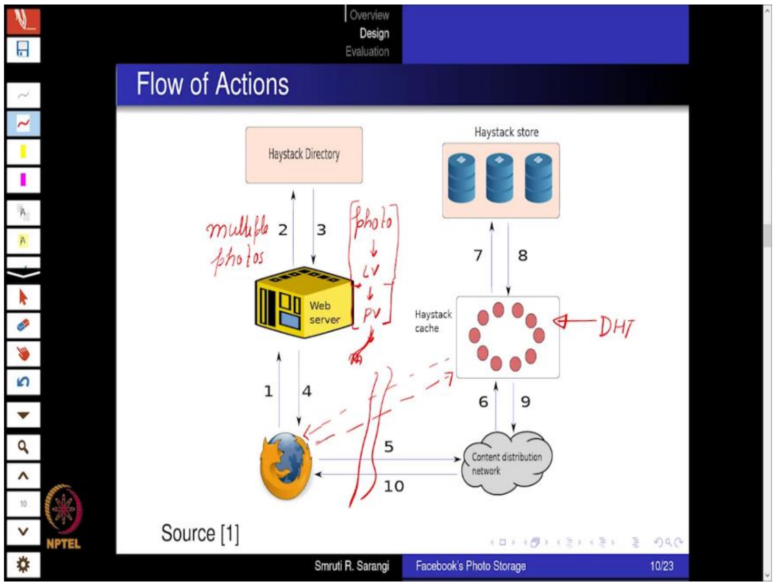
So, this can be thought of as a large distributed directory. Each logical volume will be stored on several servers. So, then each replica will be called a physical volume. So, each logical volume, let me call it an LV will be stored on multiple servers. On each of them, each replica will be called a physical volume.

And needless to say, on a single server will never have two replicas of the same logical volume, because the aims of redundancy are not really being served. We then have a haystack directory which actually does the overall control which exercises the overall control in this process. So, what the haystack directory does is that it does a logical to physical mapping in the sense, given the logical volume ID it creates gives you the physical volume ID, which is essentially includes the ID of the machine that stores the physical volume how server, it is specified either IP address is the easiest.

Second, given the photograph, it maps it to the logical volume. So, what it essentially does is given an identifier for the photograph, it maps it to the logical volume and further maps it to the physical volume and physical from the physical volume, we find the machine that stores it. And what the machine actually gets is a tuple of the photograph and the logical volume. Because, the machine might be storing multiple physical volumes one corresponding to each different logical volume. So, this tuple is actually given to the machine such that it can find the photograph from within it.

And what is the cache? Well, the cache is a DHT, a regular DHT that we have seen in the past, something like Pastry, chord, dynamo etc. So, it is a regular DHT, which is an internal CDN of sorts, which can cache data and give it to you. But of course, it is not a small in machine cache, it is a large distributed cache. And the large distributed cache is organized as a DHT.

(Refer Slide Time: 21:38)


Overview
Design
Evaluation

Flow of Actions

So, let us now take the previous figure that we have shown and add some details into it some extra detail some additional details into it. So, one of the detail is detail is like this, that given our web browser. So, essentially, we open a Facebook page on a web browser, the web browser sends the request to the web server. The web server then sends the request to the haystack directory.

So, what does the request have? Well, the request has either the identifier of one photo or multiple photos. Then what the haystack directory actually does is that the haystack directory then the first decision that it takes is whether it should direct the user directly to the CDN or to the or directly to the haystack cache in the sense one connection of this nature exists where the user can be directly directed to the haystack cache by bypassing the CDN.

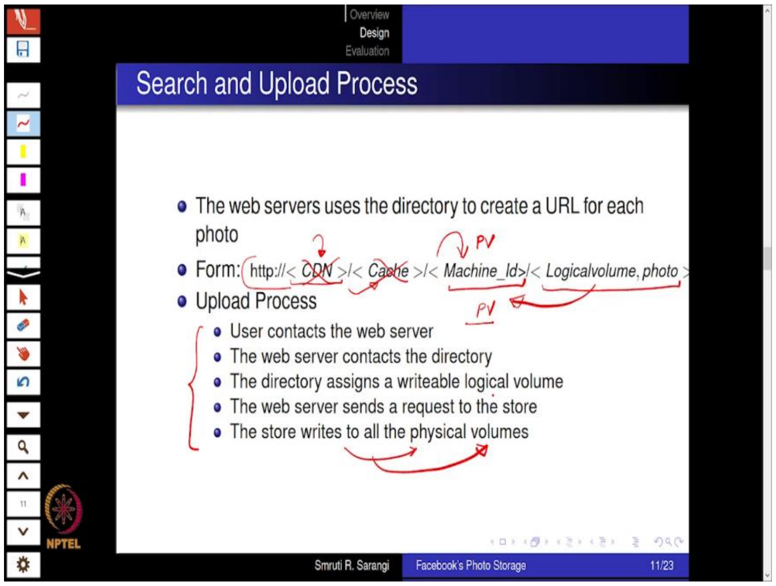
And second, given a photo it does the mapping photo to logical volume to physical volume to physically where the machine is. So, this of course is not done by the haystack cache, so, the haystack cache would only do this much a mapping and what this would be done with a haystack directory. This is something what it would return to the user well also this would also be there. So, these 3 things will get returned to the user.

So, user means the user's browser and the user's browser can then send it to the CDN, if the CDN has the photo then it will of course send it back otherwise, the CDN will send the request to the haystack cache, the haystack cache as you can see from the structure, it is organized as a like the DHT.

And then if the haystack cache has the photo, well and good, that is great, then the photo will be sent back to the photo will be sent to his back to the user or if it does not have it, it will send it to the haystack store. The haystack store will then search for the photo within the physical machine. And once it gets the photo, it will return the photo to the haystack cache. And the haystack cache will of course store it over here. So, this at least takes care of some temporal needs. Then, of course, we have an option that depends on what the directory has originally configured the request to be. So, the first option is that it can be sent directly back to the user directly back to the browser.

And the other option is that it can be sent to the CDN for further caching, which is what we show in this figure. And after caching it the CDN can return the result to the browser. So, this is completely a function of exactly how the request has been configured by the haystack directory where it actually goes first. So, this is like you can think of the request response cycle as having two distinct phases. So, the first phase is where we talk to the directory, get an expanded URL for the photo. And then we make a fresh request via the CDN via the CDN cache and store.

(Refer Slide Time: 25:38)



Overview
Design
Evaluation

Search and Upload Process

- The web servers uses the directory to create a URL for each photo
- Form: http://< ~~CDN~~ >/< ~~Cache~~ >/< Machine_Id>/< Logicalvolume, photo >
- Upload Process
  - User contacts the web server
  - The web server contacts the directory
  - The directory assigns a writeable logical volume
  - The web server sends a request to the store
  - The store writes to all the physical volumes

Smruti R. Sarangi | Facebook's Photo Storage | 11/23

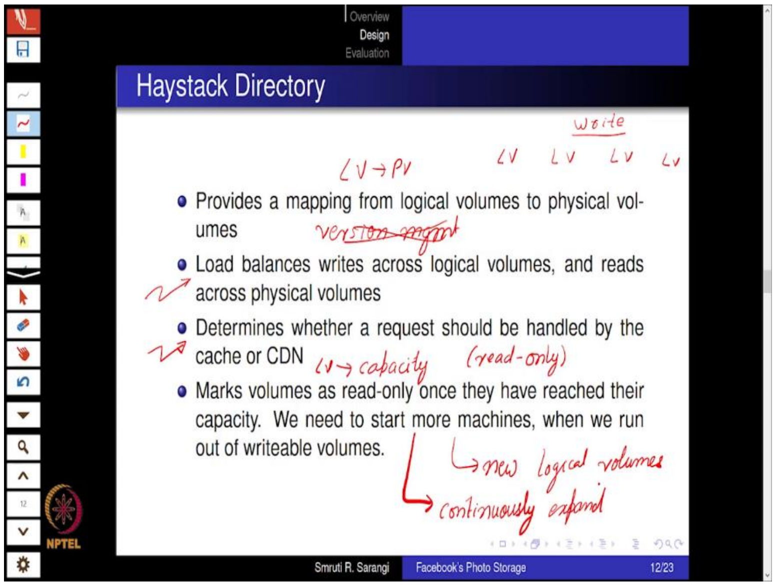
So, the web servers use the directory to create a URL for each photo. So, this that is the job of the web server. And so then the URL that they create will be of this type it will be HTTP, because HTTP protocol is being used. So, the first entry will contain an entry to the CDN. So, it will be the IP address of the CDN. And so, it will point the user to the CDN. And then the user's browser can retrieve the photo from the CDN.

So, what is being sent? Well, the logical volume and photo is being sent such that the CDN gets the first chance. However, if the CDN does not have it, then automatically it is sent to the cache, the CDN sends it to the haystack cache, if cache has the logical volume photo tuple it will return that to the user. If the if again the cache does not have it, it is sent to the physical volume, which is a physical machine within the haystack store.

The physical machine will have it for sure. So, it will again take a look at the logical volume photo. So, logical volume, of course within the physical machine is stored as a physical volume. And then it will look up the physical volume, find the photo and return. So, the upload process is kind of analogous where the user contacts the web server, the web server contacts the haystack directory.

So, in this case, the directories job is so since it is a new photo, it is a fresh photo the director's job is to assign a writable logical volume. And so, the writable logical volume, so these logical volumes are fresh, any logical volume can be assigned. So, in this case, the directory assigns a writable logical volume. The web server on its part, then sends a request to the haystack store, the store will write the new photo to all the physical volumes that store a replica of the logical volume.

(Refer Slide Time: 28:01)



Haystack Directory

- Provides a mapping from logical volumes to physical volumes
- Load balances writes across logical volumes, and reads across physical volumes
- Determines whether a request should be handled by the cache or CDN
- Marks volumes as read-only once they have reached their capacity. We need to start more machines, when we run out of writeable volumes.

So, let us now discuss the functionality of the haystack directory. So, as we have discussed before, it provides a mapping from a logical volume to a physical volume. So, this is like deciding which replica we need to send, we need to send a request to so, note that in this case, since this is read only data it is immutable data, we do not really need a quorum, so we do not really need to read from a quorum of replicas, there is no version management.

So, version management and so on that we had in Amazon, all of this is not there, we just can send it to any one replica and read. And every single photo is protected by a checksum. So, there are internal mechanisms for finding out if bits or bytes have been corrupted or not. So, one of the things that the haystack directory does, is that it does a load balancing of the rights across the logical volumes, which means that across all the logical volumes that are there.

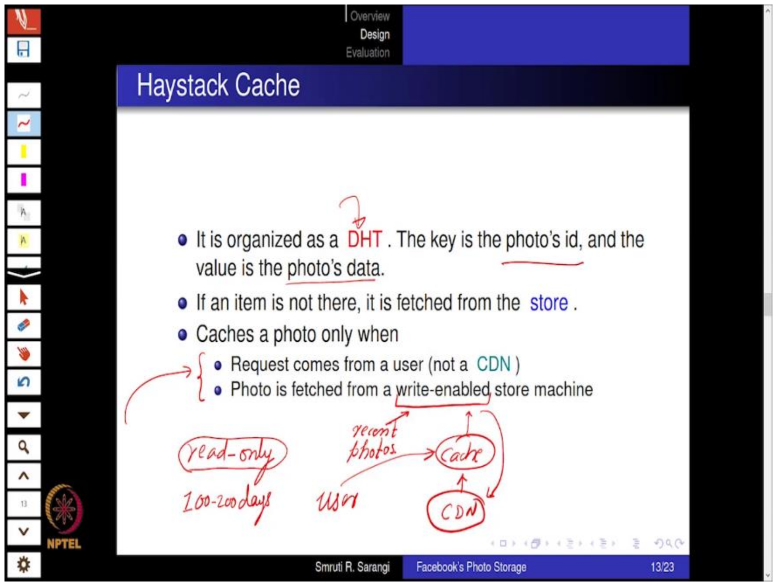
So, so let us say that writes needs to be done. So, of course, the write can be sent to any logical volume because it is a fresh photo. So, since the write can be sent to any logical volume, so we need to choose that logical volume. Whether it be some load balancing, there is not a lot of traffic in any one volume. So, it will do a load balancing.

Also, it makes a determination if the request will be handled by the haystack cache, or the CDN. If it is a popular page, it can be handled by the CDN else it is directly sent to the haystack cache, or it might be sent to the haystack cache via the CDN. Also, it marks volumes as read only once they fill up. So, every logical volume has a capacity. So, this is the maximum size of the logical volume. So, once the logical volume fills up, the entire logical volume is kind of sealed, and it is marked as read only. This means that the logical volume will further not expand.

So, what we need to do is we need to start more machines and create new logical volumes because as I said, every logical volume has a maximum size. So, this means that to create more of the writable volumes to write new photographs, we need to start more machines. This means the data center Facebook's data center has to continuously expand just to grow. Because what was found out was that roughly 20 to 25 percent of the photographs are deleted.

So, in general, we do not delete photographs from our Facebook profile unless they have of course been uploaded in error. So, since the photographs do not have to be deleted Facebook's storage capacity keeps on increasing. That is the reason we have a cap on the size of a logical volume. And once we reach the cap, it is marked read only. And we initiate we bring in more machines into our system.

(Refer Slide Time: 31:50)


Overview
Design
Evaluation

Haystack Cache

- It is organized as a DHT. The key is the photo's id, and the value is the photo's data.
- If an item is not there, it is fetched from the store.
- Caches a photo only when
  - Request comes from a user (not a CDN)
  - Photo is fetched from a write-enabled store machine

Smruti R. Sarangi   Facebook's Photo Storage   13/23

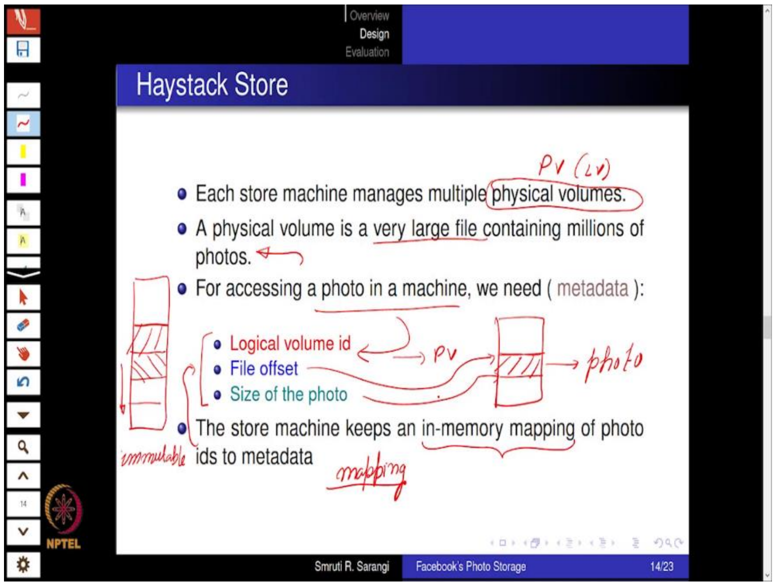
The haystack cache well, it is very simple. It is organized as a regular DHT. The key is the photo's iD and the value is the photo's data. Of course, if an item is not there it is sent to the store. So, here there are some interesting two interesting schemes that we should talk about. So, the haystack cache can get a request either from the CDN or from the user's browser directly, depending upon the way that the haystack directory has configured the request. So, if it is coming from the CDN, then it caches a photo. If it is actually coming from the user, not from the CDN.

The reason is that the CDN itself is a cache. So, there is no reason why you should also cache the photo. Because if CDN does not have a photo, it will anyway get it from the store and then the CDN will cache it, so there is no reason for us to cache it twice. But if the user is directly requesting the haystack cache, then actually cache the photo. The second is that it does not cache photos for read only logical volumes.

The reason is that a volume becomes read only after a certain time. So, this can be roughly let us say, after 100 to 200 days, once photos are not used anymore, we sort of configure the system such that the logical volume fills up, and it becomes read only. And at that point, what the engineers of Facebook actually saw is that we do not need to have quick access to these photos. Because users very rarely look at photos that are more than 3 months or 6 months old, very rarely I will take a look at them.

So, they are still a part of your Facebook profile. But they are not accessed accessed that frequently. That is the reason there is no point in caching those photos. Hence, when a volume becomes read only we do not cache its photos only for write enable volumes that are in a sense holding more recent data recent photos, we catch them in the haystack cache.

(Refer Slide Time: 34:21)



Overview
Design
Evaluation

Haystack Store

- Each store machine manages multiple physical volumes.
- A physical volume is a very large file containing millions of photos.
- For accessing a photo in a machine, we need (metadata):
  - Logical volume id
  - File offset
  - Size of the photo
- The store machine keeps an in-memory mapping of photo ids to metadata

Smruti R. Sarangi Facebook's Photo Storage 14/23

Let us now come to the haystack store, which is the third component of our discussion. So, each store machine will manage multiple physical volumes, where each physical volume actually corresponds to different logical volume. So, this is here is one of the key innovations of the Facebook engineers that they design the physical volume to be a single very large file, not multiple files. Otherwise, we would have had to do a directory access and get the details of that. It is a single file and it is a very large file.

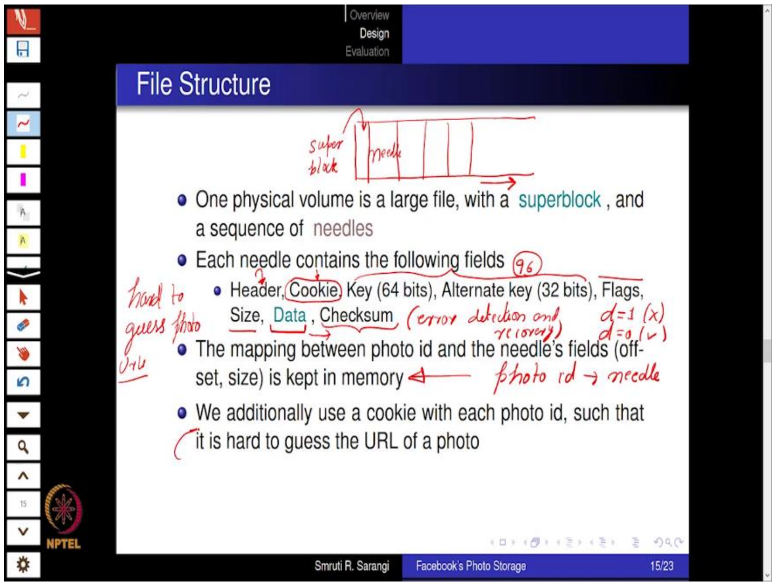
So, in the large file of course, as a large file it is organized as a stack, so pretty much what we do is that we have the data for one photo over here, the data for the next photo, so on and so forth. So, essentially, we just concatenate the photos, the data for the photos on the large file and it is organized as a stack. So, it is never the case that we delete a photo there is a hole over here, we tried to put in something that is not done. So, we essentially have immutable data, data that does not change. And like a regular stack, we just keep on adding photos this way in this into this very large file.

So, for accessing a photo on a machine, what we actually need is a logical volume iD, because that is how it will map it to the physical volume, which is this huge large file, the offset of the file, which is the starting index of the index of the byte within this large file, and the size of the photo, which tells us how large is the photo, such that we only read this part from this large file, so this becomes the photo.

So, as you can see, the data organization is very simple. We have a large file, where these photos are concatenated one after the other. The only thing that we need to store is first idea of the file, which comes from the logical volume iD the offset, which is the starting position and the size of the photo, which is the end position within this file. Furthermore, the store machine or the machine that stores these volumes, keeps a mapping in memory in memory mapping or photo iDs to the meta data and meta data is this data.

So, from every photo iD to this better data there is a mapping which is stored in memory with the accessible stored in memory. So, this is similar in the similar pattern that we have been seeing, where we keep the mapping in memory and the actual data, some of it in memory, most of it on the disk.

(Refer Slide Time: 37:18)



File Structure

- One physical volume is a large file, with a superblock, and a sequence of needles
- Each needle contains the following fields (96)
  - Header, Cookie, Key (64 bits), Alternate key (32 bits), Flags, Size, Data, Checksum (error detection and recovery)
- The mapping between photo id and the needle's fields (offset, size) is kept in memory ← photo id → needle
- We additionally use a cookie with each photo id, such that it is hard to guess the URL of a photo

Now, let us come to the file structure. So, as we have discussed, the physical volume is a huge large file which is growing like a stack. So, first, it starts with something called the super block which kind of identifies has all the meta information regarding this and a sequence of needles, where a needle is essentially one photo. It is a sequence of needles. So, each needle has the following fields.

It has a header. So, of course header identifies the name of the photo, type, and so on. Then it has an important field known as a cookie. So, the cookie is a random number, the aim of the cookie is essentially to defeat a certain kind of attack. So, imagine I have my Facebook profile and in that if let us say my photos are kind of named, wwwfacebook.com/srsarangi/1.jpg 2. jpg 3.jpg, it will become very easy to guess the iDs of photos, I can start doing bulk downloads. And I can even if there is little bit of laxity and security, I can guess the URLs of photos of other users, I can download them want to defeat that.

So, a cookie is a large number. So, it is a 32 to 64 bit number. So, this is a random number which is associated with each photo. So, to get access to a photo, we kind of need to know its URL and also supply this random cookie which you can think of it like a password for the photo and only if the cookie matches the photo is sent back. So, this makes it hard to guess this makes it hard to guess photo URLs.

Because for every photo that we get the cookie has to be supplied and we will not know the cookie unless we are a legitimate valid user and the photo is also being accessed in a legitimate and valid manner the browser will not have the cookie for a photo. Every photo internally is identified with a key and an alternate key.

So, the sum of the key and alternate key is 96 bits. So, that identifies a given photo then we have the flags field. So, the flags field is very interesting. So, we have till now maintained that a photo is not deleted. So, if photo is not deleted, then so, that is the reason we have an ever growing list in one direction.

But in practice, we can delete photos from Facebook, even though it is rare. But nevertheless, we can delete photos, we can delete entire albums from Facebook, this is possible to do. So, in this case, we assure some mechanism to delete and the mechanism is very simple, we keep a single bit the delete bit. If this is set to 1, that means it is deleted, the delete bit is set to 0, it means it is fine.

So, what we do is we keep a single status bit a single status field within a flag. And this is within the flags field of the needle. And to delete a photo well, it is as simple as just saying that we have set a bit to 1 which means that the next time the haystack system actually sees the flags, it will see the delete bit and it will automatically conclude that the needle has been deleted.

Then the regular fields we have the size field number of bytes the data the data or the photo the actual data of the photo and a checksum to just verify that if a single bit or a single byte has been corrupted or a few bytes have been corrupted, it will allows us error detection and recovery. So, the mapping between the photo iD so the id of a photo and the needles fields, the offset the size etcetera is kept in memory. So, the memory we have a mapping we have an index from the photo iD to the corresponding needle. And so, the aim of the cookie field was mentioned that is hard to guess the URL of a photo.

(Refer Slide Time: 42:27)

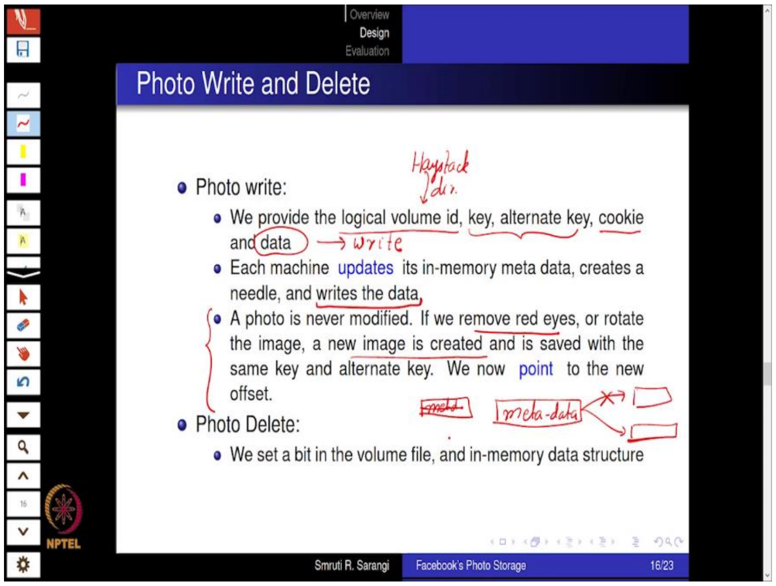

Overview
Design
Evaluation

Photo Write and Delete

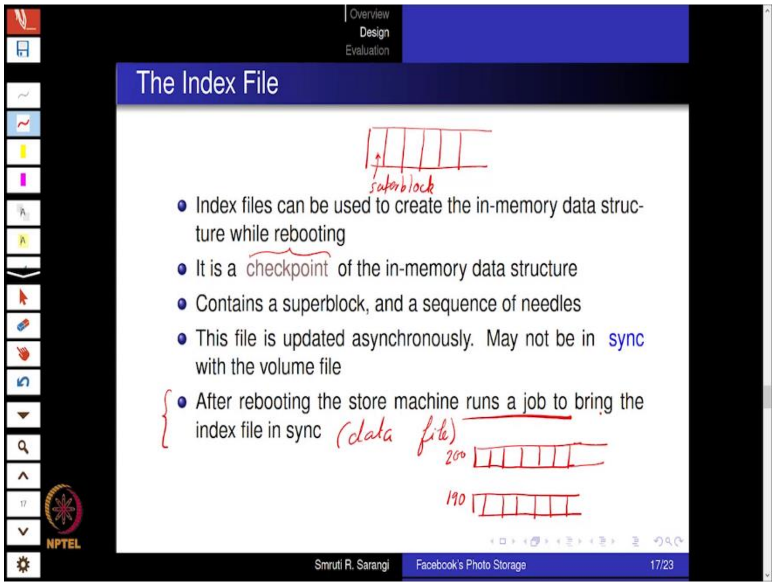
- Photo write:
  - We provide the logical volume id, key, alternate key, cookie and data → write
  - Each machine updates its in-memory meta data, creates a needle, and writes the data.
  - A photo is never modified. If we remove red eyes, or rotate the image, a new image is created and is saved with the same key and alternate key. We now point to the new offset.
- Photo Delete:
  - We set a bit in the volume file, and in-memory data structure

Smruti R. Sarangi Facebook's Photo Storage 16/23

So, now let us come to the write and delete operations. So, for the write operation, we provide the logical volume iD which of course we get from the haystack directory the key and the alternate key the cookie, which is a random number, and the data the data or the photo once we have all of this we go for the write each machine each physical machine updates its in memory metadata index, it creates a needle and it writes the data.

So, each machine independently that stores the associated physical volume updates in-memory metadata index place the needle writes the data and since a photo is never modified, if we remove read eyes rotate the image a new image is created and is saved with the same key and alternate key so with the same pair of keys we save it but it is just that in the meta data, in the metadata that is stored in memory previously were pointing to 1 needle, we just changed the mapping to the new needle photo delete well, we have discussed this in the past that we set a single bit in the volume file and in-memory data structure to indicate that the file the specific needle, the photo associated with the needle has been deleted.

(Refer Slide Time: 44:21)



The Index File

- Index files can be used to create the in-memory data structure while rebooting
- It is a checkpoint of the in-memory data structure
- Contains a superblock, and a sequence of needles
- This file is updated asynchronously. May not be in sync with the volume file
- After rebooting the store machine runs a job to bring the index file in sync

So, let us now discuss the structure of the index file. So, the index file is used to create the in-memory data structure. So, while doing a reboot of a machine the index file is created. So, it is essentially checkpoint so it is either created or it is read. So, it works like a checkpoint of the in-memory data structure.

So, the in-memory data structure what we do is we read the index file, we map it into memory and that becomes the in-memory index. This has a similar organization we have a superblock and again, a set of needles, where in this case, a needle is essentially a pointer to the needle in the data file, again, a set of needles, so all of these are needles.

So, the index file and the data file are kind of updated a synchronously. So, it is not like we take a lock or a semaphore or something and update both in tandem. So, the both of them might not be in sync all the time. So, the application is aware of that. So, of course, the data file is primary and the index file is kind of secondary. So, that is kept in mind.

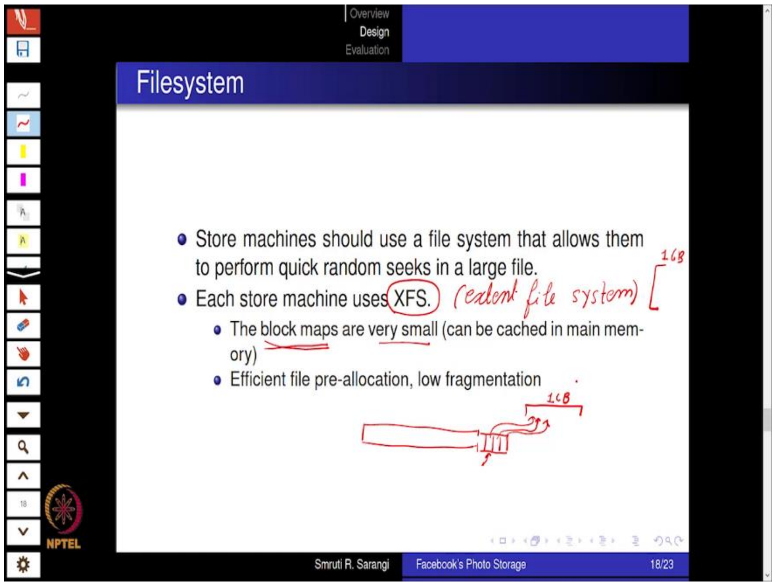
So, one thing that we do is after a reboot, the store machine first runs a job to bring the index file in sync with the data file. After that, for a long time, it does remain in sync gradually, a synchronous, some amount of a synchronicity keeps up but then again, when the load kind of reduces there, again, brought in sync.

So, we can say that the lag between the index file and the data file is variable. But definitely either periodically or after the reboot, we ensure that we read the data file and ensure that all the entries that are not there in the index, are brought in. One advantage is that since the data file is uploaded, is updated sequentially just like a stack, it is updated in the sequence and the index file is also updated in the sequence so, we can pretty much say that there is a one to one mapping between the data file and the index file updates.

So, the index file updates that we could not actually do in a given timeframe. If we just queue them, then they will also be in the same order. So, what we need to do is we need to apply them to the index file in the same order and finding out how many outstanding updates are remaining, that is actually very easy, you just need to check the size of the queue. I will or we just keep account.

So, let us say if the data file has sent 200 updates and the index file has seen 190 then we know that the 10 most recent updates to the data file are not there in the index file that is subject to the fact that we apply the updates in order and given that the data structure is also inherently sequential, this can easily be done to simplify the entire process.

(Refer Slide Time: 47:31)


Overview
Design
Evaluation

Filesystem

- Store machines should use a file system that allows them to perform quick random seeks in a large file.
- Each store machine uses XFS. (extent file system)
  - The block maps are very small (can be cached in main memory)
  - Efficient file pre-allocation, low fragmentation

Smruti R. Sarangi Facebook's Photo Storage 18/23

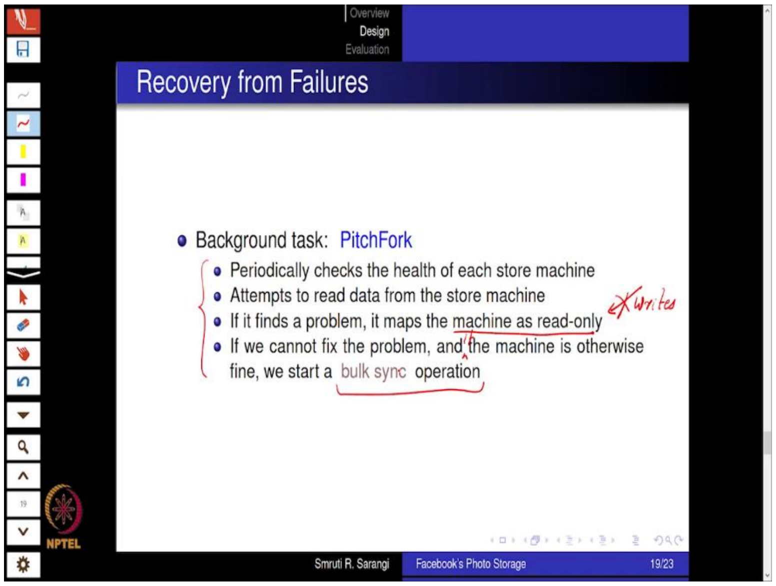
So, the file system underlying file system, well store machine should use a file system that allows them to perform a quick random seek on a large file. So, note that these are not sequential accesses, these are random accesses random seeks. So, the store machine uses a variant of, it is a it is a little known Unix file system called the extent file system.

So, in the extent file system in a normal file system, what happens is that we allocate at the level allocate space at the level of disk blocks. And a disk blocked could be 512 bytes could be half a KB, 1 KB, 2 KB, not more than that. But in the case of XFS, we typically allocate, chunks blocks fare, in I mean essentially in a much larger chunk in a much larger chunk and so maybe at level 1 GB and create a block map for it, a block map is the index corresponding to the chunk that has been allocated.

So, what we do is maybe we allocate 1 GB chunk, and then we create create a small block map for it, which again, can be cached in main memory. So, this means that we have all the indexes over here. And then we let it run out of space on the data file. We give it a new extent. And a new extent, let us say is not 1 GB extent, we create a small block map for it, which again, extends the index. And this small block map kind of stores pointers to all the needles in the newly allocated storage.

So, this also allows us for efficient preallocation of files if we are kind of aware of their sizes before the data comes. Also, what we can also do is that if we have an approximate idea the size of a photo. Let us say that they are 100 KB in a more or less, then we can kind of allocate 120 KB needles over here such that there will be some amount of internal fragmentation but then we can quickly reserve space for them and quickly insert them at the relevant needles. So, these things can be done. They are kind of optimization through the baseline Facebook photo storage file system. So, they have not been discussed in very great detail in the paper.

(Refer Slide Time: 50:11)



Overview
Design
Evaluation

Recovery from Failures

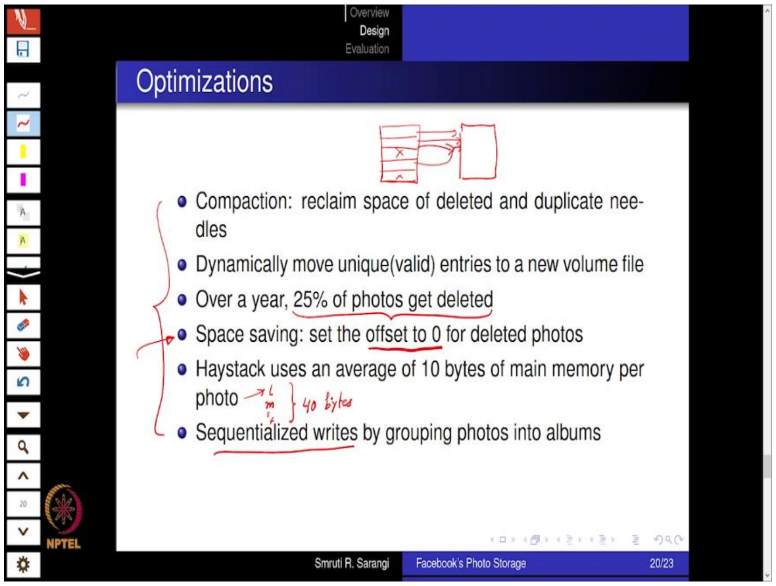
- Background task: PitchFork
  - Periodically checks the health of each store machine
  - Attempts to read data from the store machine
  - If it finds a problem, it maps the machine as read-only
  - If we cannot fix the problem, and the machine is otherwise fine, we start a bulk sync operation

Smruti R. Sarangi Facebook's Photo Storage 19/23

Recovery from failures, so a background task for pitchfork runs, this periodically checks the health of each stored machine. So, how does it do that? It attempts to read data from the stored machine. If it finds a problem or thinks that the disk is not accessible problem in the disk problem, the network it maps the machine is read only, which means the new writes are not sent to the machine.

And then we try to fix the problem. And the (ma) and if the machine is an if, and if the machine is otherwise fine, which means that it is possible to fix the problems, either via rebooting or by doing some kind of a configuration change. If we can fix the machine, then we start a bulk sync operation from another replica to synchronize both the data as well as the index files.

(Refer Slide Time: 51:16)



Optimizations

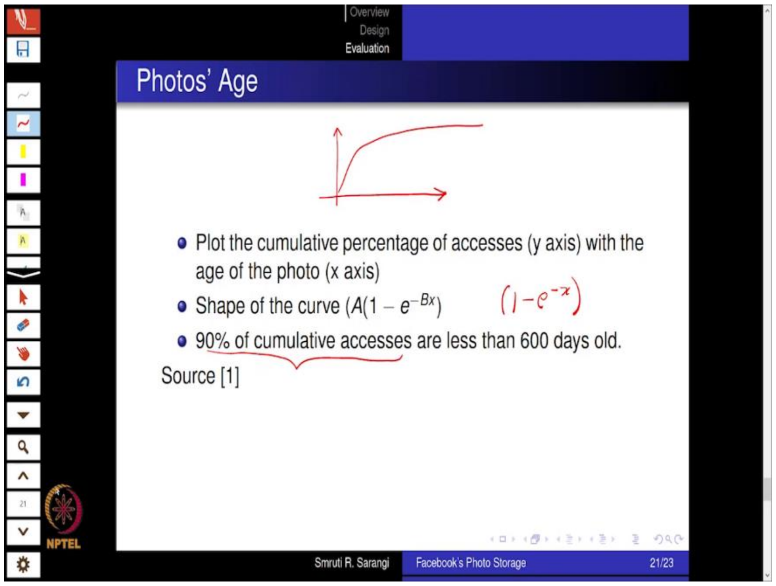
- Compaction: reclaim space of deleted and duplicate needles
- Dynamically move unique(valid) entries to a new volume file
- Over a year, 25% of photos get deleted
- Space saving: set the offset to 0 for deleted photos
- Haystack uses an average of 10 bytes of main memory per photo
- Sequentialized writes by grouping photos into albums

Some optimizations over here which are due, compaction, we reclaim the space of deleted needles or duplicate needles how do we do that, well, so the only way is we do not create holes in an existing file, we take the old volume file, and we dynamically move all the valid entries to a new volume file. So, let us say a few of these entries are invalid does not matter. So, this entry goes here, this goes here, this goes here, so we dynamically kind of compact it.

So, over a year, roughly 25 % of photos get deleted. So, this is the amount of compaction that we can do. Along with that photos get modified because we make changes to them, like rotating them removing red eyes applies, applying filters, and so on. One more thing that we can do is that we can for a deleted photo, instead of setting the flag bit to 0, we can send it offset to 0, which is like a special code being given, which tells us that look, photo has been deleted.

So, on an average for each photo, so mind you, we have 4 photos, 1 photo actually produces 4 photos, a large, a medium, a small and a thumbnail, for each of them is spent roughly 10 bytes. So, we have a total of 40 bytes of main memory 40 bytes of metadata main memory that we save per photos. And also, we try to sequentialized the writes because traditional storage devices such as commodity hard disks, group photos into albums, and so we just try to sequentialized the writes as much as possible, increases the performance at the server side.

(Refer Slide Time: 53:14)



Photos' Age

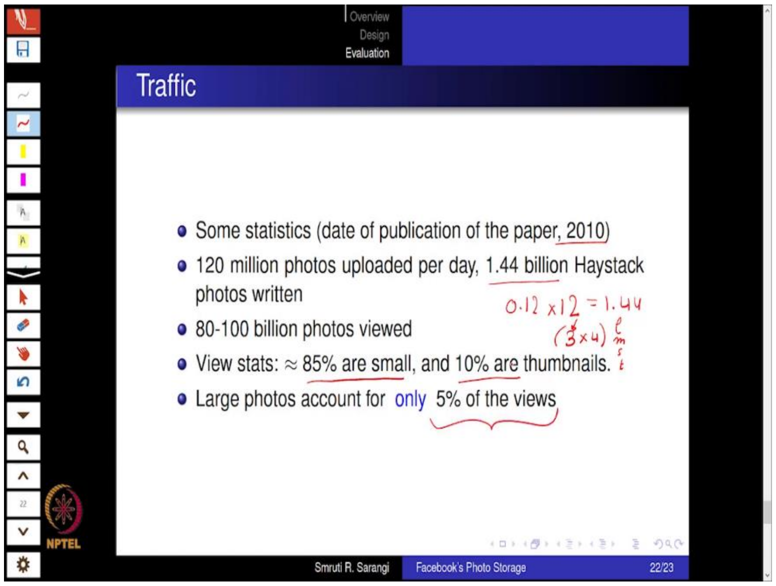
- Plot the cumulative percentage of accesses (y axis) with the age of the photo (x axis)
- Shape of the curve ($A(1 - e^{-Bx})$ $(1-e^{-x})$
- $\underbrace{\text{90\% of cumulative accesses}}$ are less than 600 days old.

Source [1]

So, if I were to plot the cumulative percentage of accesses the y axis with the age of the photo, which is a graph that we have been referring to, and this is also the kind of locality behavior, the shape of the curve is something like A(1 - e^{-Bx}) kind of a general curve of this type.

So, curve of this type is something which kind of has an asymptote, a strictly defined asymptote. And so, what we see is 90 % of the cumulative accesses, all the accesses are less than 600 days old. So, roughly within one and a half years, we access 90 % of accesses fall within that, which means we definitely do not access old photos. And if you see the chart in the paper, you will find that within 3 months or so, almost all photos are accessed otherwise, that access probability reduces to very, very low levels.

(Refer Slide Time: 54:17)



Traffic

- Some statistics (date of publication of the paper, 2010)
- 120 million photos uploaded per day, 1.44 billion Haystack photos written
- 80-100 billion photos viewed
- View stats: ≈ 85% are small, and 10% are thumbnails.
- Large photos account for only 5% of the views

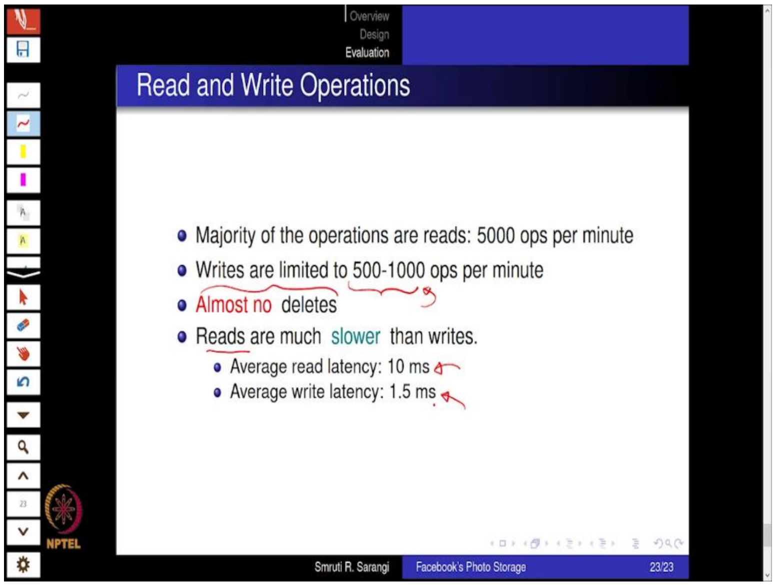
$0.12 \times 12 = 1.44$

(3×4) l m s t

So, given that some statistics, so of course data publication is 2010. So, in 2010, 120 million photos were uploaded per day. So, this means and 1.44 billion haystack photos were written. So, how does this come? So, 120 billion, this is which is point 12 billion into 12. So, how does 0.12 * 12 =1.44 the factor of 12 comes from two numbers. So, first is that there are 3 replicas of each logical volume. So, we write it thrice and 4 is because every photo is stored in 4 separate sizes, large, medium, small, and thumbnail.

So, that is the reason even if 120 million photos were uploaded per day, 1 point 44 billion historic photos were written and roughly during this time period and 100 billion photos were viewed. The view stats 85 percent, most of them are used as for small photos, 10 percent of thumbnails. So, these are the ones that you see on a typical Facebook wall. And in large photos, which are ones that we kind of expand and see are only 5 percent of the views. And so, so that kind of gives us some kind of a statistical estimate of how the photo store is internally optimized to deal with them.

(Refer Slide Time: 55:52)



Overview
Design
Evaluation

Read and Write Operations

- Majority of the operations are reads: 5000 ops per minute
- Writes are limited to 500-1000 ops per minute
- Almost no deletes
- Reads are much slower than writes.
  - Average read latency: 10 ms
  - Average write latency: 1.5 ms

Smruti R. Sarangi | Facebook's Photo Storage | 23/23

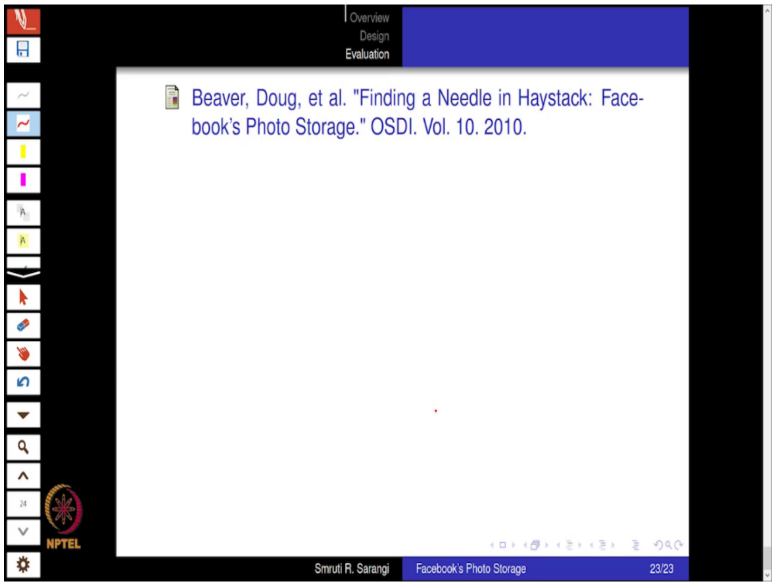
Read and write operations, well the read and write operations or the majority of operations are reads in a Facebook like system. So, one is that we are actively accessing a photo. The other is that let us say if I just visit my friends profile, I get to see hundreds of photos, even though I actually do not want to see them, the (Faces), the Facebook page is kind of full of photos, I see photos of friends, many things, he has been doing photos of friend's friends, all of that, but I am not really interested in all of them. Nevertheless, I see them.

The writes in comparison, so uploads are far fewer and as you have seen, an upload is also a more expensive operation, because we need to give it a logical volume and then multiple writes have to be done to the physical volume and given every photo, we have to generate 4 versions of it. So, all of that requires effort requires time. And but thankfully writes are not as frequent as reads. And there are almost no deletes in the sense deletes are few and infrequent we can deal with them.

One of the interesting observations is that reads are much slower than writes. So, the one of the reasons is that of course, writes are kind of off the critical path, we can quickly write something to a cache and come back and gradually the write will kind of percolate deeper into the system. In comparison reads are on the critical path. So, we directly need to go to the machine and we need to do the read writes in many cases that can be cached in-memory.

So, you can write to memory and say that look, later on the data from memory will get persisted to disk. But in read we will have to actually have to go and perform the discrete to actually get the data. And the read times are kind of high 10 milliseconds but the write times are small, 1.5 milliseconds.

(Refer Slide Time: 57:56)



Beaver, Doug, et al. "Finding a Needle in Haystack: Facebook's Photo Storage." OSDI. Vol. 10. 2010.

So, this paper was published in OSDI, in 2010. And of course, after that, Facebook itself has grown significantly. So, so Facebook is no more than Facebook of 2010. Since 2010, I still remember that a lot of people were still not using Facebook to that extent and so they, Facebook was kind of coming up those days.

But nowadays, of course, Facebook is ubiquitous. So, everybody has a Facebook page. And a lot of major updates are being sent on Facebook page, academic institutions have their Facebook pages, everybody has them. Celebrities have them, politicians have them policies are discussed on Facebook. So, in fact, if you are not on Facebook, it appears that you do not exist.

So, this was not there 10 years ago, so those 10 years ago itself, we are looking at so many photos, billion plus updates. So, nowadays in the current regime, things are, of course, an order of magnitude more. But many of these things are not documented. So, we will have to wait for Facebook to publish more papers such that we get an idea how they handle today's skill, and what modifications they have had to do to actually handle data and traffic in today's skill.