Storage Systems
Dr. K. Gopinath
Department of Computer Science and Engineering
Indian Institute of Science, Bangalore

Highly scalable Distributed Filesystems
Lecture - 43
Highly scalable Distributed Filesystems-Part 5: Lessens to learn from the design of highly scalable Distributed Filesystems like Hadoop, PNUTS, Azure, Cassandra, Haystack

Welcome again to the NPTEL course on Storage Systems. In the previous classes, we were looking at issues relating to the design of the Google file system and we also looked at some of the problems in that system, and how a newer system called the BigTable and resolve these issues.

(Refer Slide Time: 01:08)

BigTable Impl

- 3 main components: a library that is linked into every client, one master server, and many tablet servers
- tablet servers dynamically added (or removed) from a cluster to accommodate changes in workloads.
- master responsible for
  - assigning tablets to tablet servers
  - detecting addition and expiration of tablet servers
  - balancing tablet-server load
  - garbage collection of files in GFS.
  - also handles schema changes such as table and column family creations
- persistent state of a tablet stored in GFS thru memtable

So, just as a summary if you look at what we discussed previously a BigTable has three main components a library that is linked into every client a master server and many tablet servers. The tablet servers basically have multiple rows and that is the unit of distribution. And again if tablet servers they can fail or if you have more rows coming in you want to move it to a new machine that also can come in. So, it accommodate changes and workloads, this kind of dynamic activities with respect to tablet servers keep happening.

Again you have a master this actually does assigning of tablets to tablet servers, detecting which tablet servers are available which are no longer working, you need to do some balancing, and then you also need to do some garbage collection of files in the GFS. Basically it turns out that the GFS and BigTable together work similar to a what is called a log structured file system, you keep some information and then some older information becomes stale which has to be deleted and so that garbage collection is an important part of the system. And the BigTable actually manages the garbage collection of files in GFS. As you remember there is no when you do delete in GFS, it is just marked as deleted, it does not do something immediately, it is done later as part of this garbage collection service.

(Refer Slide Time: 02:58)

Interactions betw GFS and BigTable

- Persistent state of a tablet stored in GFS
- Incoming writes committed to a commit log that stores redo records
  - recently committed ones sorted and buffered in memory as memtables
  - older updates stored in a seq of SSTables (sorted string tables)
- Incoming reads checked in memtables and SSTables
  - sorted tables: efficient op
- To recover a tablet:
  - a tablet server reads its metadata: the list of SSTables that comprise a tablet and a set of a redo points (pointers into any commit logs that may contain data for the tablet)
  - server reads indices of SSTables into memory and reconstructs memtable by applying all committed updates since the redo points

Now, it also handles certain other things like if you change the schema. So, I think we will just briefly look at the interaction between GFS and BigTable one more time the important thing to remember is that the persistent state of a tablet they stored in GFS. So, basically then writes come in you commit to a commit log, and these once you get committed or sorted and buffered in memory, and those sorted and buffered records are called as memtables. So, all these stuff is in memory. And then just like a log structured file systems, once you are running out of space for example, memory for example, you can push out those full memtables presumably full memtables onto positional storage the SSTables and this is what is being stored in GFS.

So, in case that I mean reads coming in the information can be multiple places it can the memtables because that is where recently it got put in or it could be in older ones. And because its acts like let us say it does not really delete old information immediately stale information also can be there you need to figure out which is the most recent one. And of all the possible memtables and sstables you have to find the most current one. So, in a sense you have to look up all these tables. Now, because these things are sorted, you can do this fill easily because notice that all these tables are sorted; otherwise you are going to have very hard time finding out in each of these tables. So, because you are sorted it is easy to check each one of these things and figure out which is the most recent version that I am interested in and pick it up. This is some of different for a log structured file system where you need to have additional indexing information by which you can actually checkup and look up what you reading it. So, this is slight difference between the way this is done here and there. Here the sorting actually helps it to be much more it is certainly time consuming, but it is easier might comes for looking up to its.

Again if there are any failures the position state is sitting in the GFS, basically have to read the tablet and then basically the tablet server has to read its metadata which is basically a set of those updates stored in SSTables basically the ones which have been made persistent. You have to read all those things into memory. And you also have a set of redo points basically these are the points where but basically the commit logs tell you that this contains data for the tablet. So, and these data read into memory and you have to basically reconstruct the memtable, because I told you the memtable is the thing which keeps the current set off recently committed ones and then you can continue from here.

So, this all the system has been set up there is an interaction between GFS on BigTable GFS actually worries mostly about things like how to store it efficiently in 64 megabyte chunks. For example, and it also worries about reliability, it worries about how updates are done, I tries to do something about how to ensure that atomic appends take place. So, those are the kind of stuff at GFS is worries about. BigTable other hand tries to be more closer to application and you tries to figure out how to make record structures available to the application. So, you buffer multiple records as you come as things are coming in a some kind of a log structured manner and keep pushing it out to GFS once you have certain amount of size.

So, this all the interaction that is of the GFS and BigTable have been designed is an example of what you might call as a cross layer design. Each party tries to do certain things, but you try to ensure that the right set of functionalities of the different layers. So, that is the idea. And GFS came first and it was designed for handling mostly web search and crawling of data; and the BigTable came in because a lot of people are doing other things also with it, and you had an easier way to interact with this GFS system. So, it is likely closer to application I would say.

(Refer Slide Time: 08:07)

Hadoop Distributed File System

- Rack-aware filesystem
  - To help run work on the node where the data is, and, failing that, on the same rack/switch, to reduce backbone traffic.
- HDFS stores large files (an ideal file size is a multiple of 64 MB) across multiple machines.
- Reliability by replicating data across multiple hosts (no RAID storage on hosts)
- Above HDFS runs MapReduce Engine
  - Client applications submit MapReduce jobs to JobTracker

So, now we will take a look at slightly similar systems that have been designed hadoop is one which is closely modelled on the GFS slash BigTable kind of model. And this say it is a rack-aware file system actually most of the system that we talk about are actually rack-aware or sometimes you know what I call data center aware. They exactly know whether let us say if a data is present then which racket is present in, so that it would not replicate it they might want to keep it on a different rack. Or it can also be you want to the geographic distribution they also ensure that it can be replicated and places which are quite far away from the one of the data centers we are talk about. So, it can the same data can be in multiple data centers which are distributed neutrally. And this a important issue because the latency and bandwidth aspects or closely correlated with whether it is within rack or across racks, within a data center or geographically distributed.

So, essentially all these file systems are that, but hadoop also makes it specifically clean that is a rack our file system just like also I think Cassandra and other kind of file system also. And basically idea is to make sure that the work whatever has been done is actually closer to where the data is present there, but it is not that try to be near the rack same rack near switch, so that you try to reduce backbone traffic. Again HDFS also stores large files this is as a GFS does; and this is also 64 megabyte chunks. Similar to GFS they also do reliability by replicating data across multiple hosts they do not use RAID storage on hosts, because they upon it expensive and not giving as much reliability as if you were to distribute across multiple nodes.

Basically even if you have RAID storage the machine dies, then you do not have access to the data, so that is why the RAID storage is not using similar to again GFS. And above that can there are now this other application closer to application level entities like MapReduce engine etcetera. And some other entities like JobTracker which is doing the scheduling part which is modeling about scheduling. So, this is an open source system compared to GFS. So, it is being widely used and quite a bit of recent infrastructure from other major companies also use HDFS. So, there are all differences between this and GFS, but I just wanted to point out that there is no similarity also with respect to the kind of GFS kind of design we discussed.

(Refer Slide Time: 11:08)

Yahoo! PNUTS

- focuses on data serving for web applications
  - workloads mostly of queries that read and write single records or small groups of records
  - not for complex queries, e.g., offline analysis of web crawls
- data storage organized as hashed or ordered tables
- designed for low latency for large numbers of concurrent requests including updates and queries
  - all high latency operations asynchronous
  - support record-level mastering (local ops afap)
- per-record consistency guarantees: all replicas of a given record apply all updates to the record in the same order

We will also look at briefly look at another design. Yahoo has come up with some system called PNUTS, which is a large distributed geographically distributed data serving system for web applications. So, basically what they are trying to look for is how to support workloads mostly of queries that read and write single record the small groups of records. So, not for complex queries. Again because if the workloads are mostly dealing with single records or small groups of records then it is slightly easier to manage consistency also, because notice that GFS and other systems usually provide atomicity for only a single rows. So, if it is single records or groups of records possibly it is there in the single node and probably you can give you lot more guarantees then otherwise possible.

The data in this particular system is organized as hashed or ordered tables. We will talk about this hashing soon, this is one other different model of storing data. And we briefly looked into it sometime back, but I think we will look at it in some additional detail with respect to what is called consistent hashing we look at that part of it soon. And the most important thing that they are trying to do is to design it for low latency for large numbers of concurrent requests including updates and queries. So, because of this anything that is high latency there they usually do it asynchronously. And other thing they do is they are able to they want to support a system by which if you are modifying any records you can actually have master for it, so that it becomes a local operation for even you update those think. So, essentially you can cache it and update it in some sense is something like a session semantics, you copy it keep it with you updated as much as you want and then put it out back. So, you want to do as many local operations as far as possible.

Other thing that this system does is give you per-record consistency guarantees all the replicas of a given record apply all updates to the record in the same order. So, we will look at this a bit more. Basically you notice that in GFS etcetera also we will find that the updates happen in decided by the master; only thing is that different let us say depend a master are does one particular set of auditing of updates if it fails the new master can different order. So, there are some issues with respect to a kind of updates that are present in if you are doing for example, attends right, we will find that they can be in different orders and different machines.

So, what is attempted here is to give additional control to the application, so that you can say that with respect to a record at least, all replicas of given record apply all the updates

recording a same order. So, you want to know instead of doing it at the slightly higher levels, you want to do it at slightly record-level and similar to what is given in BigTable with respect to rows.

(Refer Slide Time: 15:25)

For example, they have something called timeline consistency model. So, the basic idea is the particular object has a timeline, in a sense it has got, for example, it got RAID here, it got modified here etcetera, there are various versions of the object available. So, reads are served using a local copy, but this may be stale with respect to the more current versions somewhere else. But the application API can specifically ask for a particular version or any version greater than a particular number. For example, I can say give me the most current version or better than, sorry, I want version, which is more than the fifth version. So, basically while copies one may lag master record every copy goes to the same sequence of changes. So, in a sense there is a timeline for objects of all places.

So, you can essentially say the following that I am interested in a particular version or any version later than that one. And because of this it is essentially consistent across all the systems, across all the clients it should all be consistent across. You also have a system, which is test-and-set writes. You can essentially say what is the current version and if it is this current version, I will allow the write to go through; otherwise I will not go I will it will become no op. So, it turns out this can be used to also facilitate per record transactions. Essentially you can roll back you can try to write it if it is not

possible to control back. So, far this means that you will be retrying certain operations multiple times before you can you may succeed.

(Refer Slide Time: 17:41)

PNUTS design

- uses a guaranteed message-delivery service rather than a persistent log
- trigger-like notifications
  - imp for some apps that must invalidate cached copies after some time (eg. ad serving with a time contract)
  - users subscribe to stream of updates on a table
    - asynch publish-subscribe message system
      - can be optimized for geographically distant replicas and replicas do not need to know locs of other replicas
      - contrast to gossip protocols

One of the difference between this system and previous systems we looked at is that they say they use what is called a guaranteed message-delivery service rather than a persistent log. You will notice that in the case of GFS and BigTable, we had certain logs and the primary kept those logs and then you had to use that for recovery etcetera. Here they are depending on a guaranteed message-delivery service. So, the various methods by which sorry messages can be delivered and guaranteed that actually that delivered. And they use this kind of systems for example that I use something called publish-subscribe message system that is what is used. So, another thing that they have done is provide support or what I called triggers that means, that if some data changes, applications can register themselves in the system saying that if this particular change takes place I want to be notified and this is important for some applications that must invalidate cache copies of sometime.

So, for example, if you are serving ads in the system, and there is a contract, it says you after some period of time, the contract expires, you want to actually remove all that ads corresponding to that. Or you might have a system which is trying to track certain websites as soon as some modifications take place, I want to be modified because I want to reindex it. So, if you are looking for slightly real time kind of models, you need to do

this. Again GFS also has proceeded in this direction if you look at Google they also have a service which is similar to this, they have something called a mega stored which is something similar to this. So, you want to essentially provide a trigger like capability, so that you get notified whenever things happen and that is done by users subscribing to stream updates on a table. And this is asynchronous publish-subscribe message system.

So, the good thing about this is that anybody can publish it and anybody can subscribe to it, they do not have to each parties do not know the who the other parties. So, basically what I am trying to say is that if there is a because everything goes through this particular message system, this particular system keeps track of who the parties are who subscribing and who are publishing, who subscribing. So, for example, if you are trying to update something, one replica does not need to know they locates others replicas, you just have to publish it and then the system and actually make sure that the other subscribers who are a looking for that updates they can get to it.

So, those kind of things yes in a in the sense what is happening is that, there is a broker in between, there is actually trying to figure out what people are interested in and they make sure that it happens. And because of the system it turns out that they can also optimize it for geographically distant replicas, you can figure out what is the best way to push the updates. So, this is in contrast to gossip like protocols which might not really be tune for geographical distribution, they might not be really taking care of this kind of things. So, this is one other design.

(Refer Slide Time: 21:50)

Windows Azure

- Designed as a scalable cloud storage system
  - cloud storage in the form of blobs (user files), tables (structured storage) and queues (msg delivery)
- Blobs for incoming and outgoing data, Queues for overall workflow for processing the Blobs, intermediate service state and final results in Tables or Blobs
- Publically searchable content (via Bing) within 15 secs of a Facebook/Twitter user's posting or status update
- "Strong consistency": same as others (within a "stamp")
  - Intra-stamp synch repl
  - Inter-stamp asynch repl
- Global and Scalable Namespace/Storage

Again, we will quickly look at another design, which is the windows azure model this is slightly different system, because it is designed as a scalable cloud storage system used both by end users as well as by companies like Microsoft themselves. For example, they want to use the system for search just like GFS was used by Google, they also wanted to export it out as a cloud storage system which other parties anybody who is interested can use it. So, their basic design is keep this data for example in the form of what you call blocks these are files or in tables, structure storage or queues message delivery. Again this queues is similar to what we saw in the case of PNUTS also. There also you have some kind of message model right guaranteed message delivery model same thing also is being done here in this case instead of depending on persistent logs.

So, basically in the system you have a blocks for incoming and outgoing data, again this similar to in the case of BigTable, you will notice that incoming tables coming to both memtables then stored persistently in sstables etcetera. Similarly, I can get somewhere log structure is going on there. So, here what is happening is that the blocks are being used for incoming and outgoing data. And the queues are used for basically this has got the message delivery. So, the message delivery is the message is essentially tell you the kind of workload that has to happen, so that is why this queues are for they are used for overall workload processing corresponding to the blocks, and any intermediate data is stored in tables of blocks.

So, they say that just like PNUTS was using some triggers, they also provide publicly searchable content within a very short time of something being published. For example, they claim that for example, 15 seconds of somebody posting something in Facebook or twitter, it will show up in their search let us say engines for example, Bing. So, basically because only way they can do it is by having some way in which somebody is pushing the updates to them also. So, they can subscribe to certain APIs which Facebook or Twitter is making available, so that as soon as anybody post something, they get the data and they actually push it through their system, the cloud storage this system out here and then that is made available to the Bing's search system, again it will reindex it. So, that new (Refer Time: 25:18) will actually can pick it up. So, this is another thing that they are talking about.

This windows azure system says that they provide strong consistency, but if you look at it too closely it basically the same as what other people are providing. For example, if you look at what PNUTS gives you or GFS gives you sorry the BigTable gives you, it is basically within a tablet or within basically a data center or a rack for example, those kind of things. And the name which windows azure gives is within stamp, stamp is basically one thing which is all available in a single place it is not essentially multiple racks, they might have for example, a stamp might have something like about q tens of terabytes of data in one single place

So, they provides strong consistency within a stand and across they usually do not provide it. And they say that most of the times if you do intelligent partitioning it turns out and the upper load protocols are properly set up. Almost all the things updates actually happen within a or whatever you are concerned about happens actually within a stand. So, that essentially you can say that you can get strong consistency that is the claim they make.

Again this like the previous system if you are looking for the thinnest stamp within a let us say data center or within multiple racks, they usually do synchronous replication. And across stamps, they do asynchronous replication. Again as you know the asynchronous thing is a thing that comes going to picture anything you have the cap theorem in to picture. So, as long as you are doing synchronous replication, then there is no serious issue that is why they can this is say that they provide strong consistency. And because it is a cloud storage system they also provide user accessible global and scalable

namespace in storage because this is in contrast to other systems which do not have a user set of users also using system. So, they have to provide a global and scalable namespace.

So, I am not going to too much in detail this is actually each of the systems is quite complex. And there is considerable amount of description that is available for example, if you want to look up windows azure in the recent SOSP 2011, they have described it and in some detail. And you should look up those details if you want further details.

(Refer Slide Time: 28:23)

Another Model for Storage: Consistent Hashing

- Hash both objects and devices using same hash function
- Map each obj to a point on the edge of a circle
  - Equivalently, to a specific angle
- Map each device (eg. storage bucket) also pseudo-randomly mapped on to a series of points around circle
- An obj stored by selecting the closest mapped device on the circle
- Each device contains the resources mapped to an angle between it and the next smallest angle

If a device added or removed, only nearby objs remapped

Used by Amazon and Facebook

Now, if you look at the previous systems we talked about GFS and the PNUTS etcetera, they basically have the model of one particular system with the master for example, which actually keeps track of their the actual tablets are or the various let us say 64 megabyte chunks are etcetera. They look (Refer Time: 28:53) they actually manage all those things. There is another model for storage and which is basically what is called consistent hashing. What we do in this thing is that you try to avoid having a master in the picture, you want to directly use the name and hash it and then from there figure out where to get the data. So, you want to avoid having somebody like a master in the picture and this also is used in some other systems. For example, if you look at a parallel file system called GPFS which IBM designed some time back they also use something similar to this.

So, basically the idea is to in a parallel file system, it turns out that if you want to parallelize all the activities, you have to parallelize also then you have look up part of it also. The pathname can be quite long, if you become sequential then your throughput also can be again going by (Refer Time: 29:55) laws and another things of that kind the minute a pathname lookup is serialized then the chances of you are being able to scale is also that much smaller. So, they had to incorporate abilities by which you can do name lookup as soon as possible without any sequentiality, and the way to do this by hashing area.

Similarly there is another file system called CEPH which also does something similar it may not is consistent hashing, but it also uses some hashing to may take a name and map it to a hash value and that hash value is used as a way to look it up. Here what happens here is you are looking for highly scalable storage. So, basically what is going on is people are adding more and more storage and sometimes some storage also is a little because of failures. So, you want a way in which you can add this storage without having to move things or load balancing. What is that in the case of GFS and other kinds of models, there is a explicit web balancer which actually has to move things are all which can be which is done in the background, but it is an issue, it can actually cost you quite a bit of stock in terms of time and energy.

So, here the idea is to find a way in which your some more civilian to additions and deletions that is the basic idea here. So, here the basic model is the following. You have various objects in the system and you have various devices storing objects. You basically hash both objects and devices using same hash function map each object to a point on the edge of a circle. So, basically you have some kind of circle on which all these things are residing and you mapped it to various points on the circle. You can have call it a point or a specific angle either one is fine. So, you map this objects other way you also map each device becomes storage buckets. For example, we talked about S 3, which has solution on something called storage bucket, you map those storage buckets also onto a series of points on the circle.

Now, the object is stored by selecting the closest mapped device on the circle. And therefore, each device contains the resources mapped an angle between it in the next smallest angle. So, you have to search it only those places. Again there is going to be a the device that is going to be some data structures kept here so that you can do the look

up. So, basic intuition behind this kind of model is that if a device is added or removed only nearby objects remapped, you do not have to move anything else. So, this is the basic insight about using this kind of system, and you try to avoid depending on masters like what was done in the case of GFS etcetera. In this kind of systems have been used by some major companies for example, Amazon has used it as well as Facebook has used it. Both of these parties are uses it. And also used in PNUTS also uses it, they also say that you can do storage also in this particular model, they give you the option.

(Refer Slide Time: 33:48)

Distr Hash Table (DHT)

- Uses some variant of consistent hashing to map keys to nodes
- A node with ID i_x owns all keys k_m for which i_x the closest ID, measured according to $\delta(k_m, i_x)$
- To store a file with filename *fn* and *data* in DHT
  - Calculate, say 160-bit hash key, *k* = SHA-1(*fn*)
  - A message *put(k,data)* sent to any node in DHT
  - msg forwarded from node to node thru overlay network (connecting nodes) until it reaches single node responsible for key *k* as specified by the keyspace partitioning
- To retrieve file *fn*, *get*(SHA-1(*fn*))

So, again just to look at it in some detail what is being done. This hashing technique users basically what are called distributed hashing tables DHTs. So, it goes as follows. So, basically what we are doing is we are trying to hash the keys to nodes. So, what do we do here you have a node with ID that is i x it owns all keys came for which i x is the closest ID and there is some kind of a nearness measure which gives you that information. So, example you have a key k m, and then you can see how close is it to a particular node by ID i x.

So, what you do is basically hash both of them and then from there you can figure out which is the one which is closest which you have to search. You can get and that closeness is going to be given by summation. For example, if you want to store a file with the filename fn and data in a DHT you first hash a the file name using some hashing function, the one people often uses SHA-1 because it has got very good what is called

the collision resistant properties. So, these are cryptographic hashing functions which are good collision resistance that means, that even if one small bit is changed it actually mapped to something quite different, they use some techniques called confusion and so called in what is other thing called they use some analog techniques by which this happens.

So, you want to store it you basically get the hash and you say put k comma data is sent any node in the DHT. You can send it anywhere. And then that node in turn what it does is it forwards it from to any other some other node through a sum overlay network which connects a nodes and until it reaches a single node responsible key k as specified by which the key space partitioning. So, essentially things are forwarded till it looks a right place, so that is what it is stored. And then if you want to get it you basically send this get again take the file name and then compute the key, and ask or you to get it. And any node can be given to any node and then it is actually again forwarded and then it is picked up from wherever it is, and then it is come through the overlay network and it comes back to the client. So, this is roughly is the model of distribute hash tables.

And as I mentioned this is what is used by some of these major companies or storing large amounts of data so that you do not have to keep on moving things around when something are deleted or added. Here also there are some hotspots and other kinds of things, but we will not discuss it here.

(Refer Slide Time: 37:48)

So, I will just briefly mention one example of another system. We choose has DHT. Basically it is similar to BigTable except that they use this DHT kind of models. So, it is also as a distributed multi-dimensional map indexed by a key. And similar to those kind of other models the key is basically a byte string and value is a highly structured object just like BigTable, BigTable also has got columns etcetera, so same thing here also. And just like BigTable, they also guarantee that any operation on a single row key right is atomic per replica, no matter how many columns are being read or written. So, they guarantee that particular aspect and they also had similar to a BigTable in terms of column families etcetera.

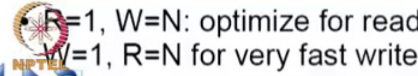
So, the data partitioned across cluster using consistent hashing and they have a small little tweak they use some what is called order preserving hash function. So, the API is quite simple again we are not going to too much into detail as I mentioned this there are lot of there is considerable amount of literature available on each of these systems. Basically they have three simple methods inserting in change, row mutation is a change into a particular row in a particular table with a particular key. We can get given a key a particular table, you can get a particular column, and you can also delete something because they have only three methods by which you can interact with the system. So, I am not going to too much detail about some of these things maybe I am going to stop with respect to details of this kind.

So, let us just briefly look at what we seen so far, we looked at GFS, we briefly looked at yahoos PNUTS. We also looked at windows very briefly window azure and then again very briefly look at Cassandra. So, basically the consensus you can have when you look at all the systems is that they are trying to provide a model of consistency which is manageable with respect to the kind of throughput. And let us say now the kind of transactions per second you need to support in the system the kind of operations you need to support high rates of operations. So, I think you can quickly look at summarize some other kinds of models basically you can have either server-side consistency models.

(Refer Slide Time: 41:08)

Server-side Consistency Models

- N = number of nodes that store replicas of data
- W = number of replicas that need to ack receipt of update before update completes
- R = number of replicas that are contacted when a data object is accessed through a read op
- W+R > N: write set & read set always overlap and one can guarantee strong consistency
- W+R <= N: Weak/eventual consistency



- R=1, W=N: optimize for read
- W=1, R=N for very fast write

or what are called client-side consistency models.

(Refer Slide Time: 41:09)

Client-side Consistency Models

- Strong: After an update completes, any subsequent access will return the updated val
- Weak: May return stale value. *Special case*:
- Eventual: storage system guarantees that if no new updates made to the object, eventually all accesses will return last updated value eg. DNS
  - *Variations*:
    - Causal consistency (CC)
    - Read-your-writes consistency (RyWC)
    - Session consistency (SnC)
    - Monotonic read consistency (MRC)
    - Monotonic write consistency (MWC)

  - If client connects to a server only, RyWC/MRC easy

That is what does for example, a client-side consistency models are those what the client sees, and the server-side is basically if it has got any replicas how does it ensure that anything updated how does it manage it one is coming from what server has to do another thing is what the client finally sees. For example if it is server-side, you will in somebody what we can see what is going to happen is the following. Let us say that there are N is a number of nodes that store replicas of data. W is the number of replicas that

need to acknowledge the receipt of updates before update completes. For example, if there are five replicas, do you wait for all the ack from five replicas before you say that the write is finished or you are willing to take only three writes has been completed before you, so that write is completed. So, W is basically a number of replicas that need to ack receipt before we say that right is completed.

And R is the number replicas that are contacted when a data object is accessed through a read op. Because in these kind of systems some of these replicas can be may not be current. So, it might be your bad luck that you whatever you contacted may be out of date. So, you can control that by having R also. For example, if N is the number of routes that store replicas then if you set R equal to N that means, you have to contact all of them and then you can see easily the fact that some things are out of let us say they are not current compared to other ones. Then you can decide what to do.

But as if R is one for example are very small then we can with high chance that you will see some stale version. So, you can basically what is going to happen is that if N is the number of replicas, if you might call W the write quorum and read R quorum is call the write read quorum is the write quorum and the read quorum together is greater than N that means, that there is some overlapping thing between the write set and the read set. And because of this you can guarantee strong consistency.

Because this is the same trick what paxos does they ensure that there is always somebody you print two it is a ballots. There has to be some guy who is the common between two ballot that is all they are able to say that if something has been committed in one ballot and it is nobody that is not good it has not been not everybody still. Then other party will there will be a way in which information flows from one of the guys in the previous balloting, where think out committed it becomes available to the next ballot with if it gets started. And therefore, the whatever decision that was committed becomes known to the second ballot image and the same thing will be followed that is why you get strong consistency there, here also we seems to be.

If you have W plus R less than equal to N then it is weak or eventual consistency nothing to show less than N, it cannot be less than equal to N is it Z equal to? It has to be less than this is correct sorry it has to be less than equal to it can be disjoint then there is no question this correct W plus R less than equal to N. So, then it in this case what can

happen is that there could be a disjoint set of replicas and therefore, they might have different versions they can be some steal values that could be written

Now, finally, if you think of it more costly if R equal to 1, and W equal to N. What is this saying they are saying that I am do not mind reading only one thing, but I am going to make sure that W equal to N that means, that everybody has to be written every replica has to be written before it is declared of success. And therefore, it will be slow, but reads will be fast because you do not have to contact multiple bits. Whereas if W equal to 1, and R equal to N, then they are basically saying that the readers take the trouble of figuring out contacting all the replicas and deciding what to do with it, if there are out of work they might have to do some protocol by ways they synchronize them etcetera or let them consistent whatever.

So, typically in web kind of applications this is what is a typical situation. Let us say while notice that in the case of GFS there is any kind anything is written you have to get acts some all the secondaries, then only the master proceeds or else the (Refer Time: 46:36). So, this is what most of the large scale web kind of system will follow. And this is not very common, I am not aware of any major application that requires very fast writes and readers are got to be realized, so that the writes can be very fast. At least some web scale system systems, you do not find this much that is the server-side.

On the client-side, I think we already and again summarizing what all you have done so far. If it is strong after update completes any subsequent access will return updated value. In a weak case, may return a stale value. The special case eventual model storage system guarantees that if no new updates made to the object eventually all accesses will return last updated value. A class example is DNS which has an information about the mappings, it will eventually converge to the write values, but it there might be some intermediate times when it does not come us it, it can give a wrong information.

The various variations we looked at causal consistency similar to when discussing group communicating systems we talked about consistence causal models. The same thing out here that means if there is a causal reason why certain updates have to have happened before some other update, you need to (Refer Time: 48:19). For example, you may have some information kept in some tables, and you may have a requirement that that it should not be seen by some people once I make certain kinds of updates. Then it should

be a case that if I updated it, then the removing of permission for somebody to look at it should have happened before, the new updates that is I want that guarantee to be given. If there is not given then it can I may not be do it I will not process the system.

So, in some cases causal constancy required. Read-your-write consistency basically means you can cache your stuff you can keep updating your own copy, and then any time you read you get back your copy. Whatever you write, but it is not synchronous with rest of it rest of it. This is also in architecture word called processor consistency basically you can have processor you have a something called write buffer, you keep on writing to it whenever you read it, it actually it looks of the read buffer and gives the most recent version. But it may not be consistent with other parties who have cached it they are also writing there.

Another model is session consistency basically this is closer to what many file systems provide. What you are doing is you open a session and then till the close you are guaranteed that you are the reads and writes happen the way you expect. And only when the session is completed do you when you explicitly say close the session, then your updates are uploaded to rest of everybody else sees it, this session. There is also some other slightly simpler models monotonic read consistency and monotonic write consistency. In monotonic read consistency, basically if you read it multiple times, they are all in a sense, their consistent basically what it means is that I read something, if I will get five; if I read it again I cannot get six, if there is no other intervening write for example.

In this can happen this may not be this may be violated in internet kind of scale systems. For example, you read a score of a cricket match, one six is it is some number 320, next time you read it can become 350. So, what is because you are getting updates from multiple different paths and that will not have monotonic read consistency. This is also moronic read consistency. So, basically what it means is that if you write something A for example, and you write B later then the value A a is seen by everybody before you see the value of B by everybody. In a sense, you are essentially ensuring that there is some kind of barrier between each write, we will first read A first completely then only you write B. Again if a client connects only to a single server then certain things are trivial for example, this read-your-writes consistency or monotonic write consistency becomes very simple.

(Refer Slide Time: 52:09)

Haystack: Facebook's photo storage

- Std solutions (eg. NFS): excessive number of disk operations because of metadata lookups
  - to read a single photo: 1+ disk ops to translate filename to an inode #, 1 to read inode from disk, and one to read file itself
- reduce per photo metadata so that all metadata lookups in main memory
  - rwx perms not needed; 128-256B inode size too big
  - now disk ops only for reading *actual data:* increases overall throughput
  - high throughput and low latency: at most one disk op per read
- Why does caching not work?

Again I just want to conclude with a slightly different kind of model. We are looking at consistency management as a critical one. There are the kind of solutions where a slightly different set of things are optimized on when you thinking about scalability. Facebook has a photo storage system they called haystack. And for some time they are using NFS, but they found that to read a single photo, it was requiring too many excessive disk operations, basically you need one plus disk operations to translate filename to an inode number because finally, NFS uses it back end file systems. So, it requires one plus disk operations to translate file name to an inode number, you have to read the inode number and then read the file basically this is required to look up the directory then after directory you have to look up the inode and then you have to look up the file. So, there are three things will required.

So, the basic idea here is to see if you can avoid multiple disk accesses. One example of it is to somehow make reduce the amount of metadata, produce a metadata then it turns out that all metadata can be kept in main memory itself you do not have to go to disk at all. So, basically here the problem was that you are going to look up the metadata from disk. So, just like GFS also did, but is a completely different with a completely redesigned file system, what we are talking about here is they are using mostly is similar kind of file system not too grammatical different, but you want to remove certain things are not useful.

For example it turns out in there system you do not need rwx kind of permissions, it is not needed, because there is no it is not a shade system that they are providing to users. There is this providing photo service it is that application level there is no other user other than this particular photo service that Facebook is providing. So, you do not need this permission information. So, you remove this kind of thinks when it turns out that typical inodes are 128, 256 bytes to 512 bytes this size is too big. You reduce all those unnecessary things it turns out you inode size becomes smaller sufficiently small that the metadata for keeping billions of photos is not possible to be kept in memory itself. I think GFS is try to do it by making very big chunks 64 megabyte chunks, so that is why they reduced amount of metadata required for keeping it.

Here what they are doing is instead of going for very big chunks, you reduce the size of the metadata itself. And if you do that you should carefully look at your system and see what the design, what you using the system for then you can essentially limit lots of unnecessary things. And they say that by looking at all this kind of stuff the high throughput and low latency and at most one disk operation per read. Again and basically you need to scale and the way to scale it is by keeping on metadata in memory, you cannot keep all metadata in memory if inode is very big. So, we way to do it is to figure out how to look at your application see what is really critical information throw out all the useless stuff and then you can able to put it in memory.

The question for this is why does caching not work. Typically often they say that if you want one disk operation per read, people often say caching will work, of course, as we notice one issue is that inodes have to becomes too big. This is another major reason why it actually does not work also and that reason is because of.

(Refer Slide Time: 56:02)

Haystack Design

- Content Distr Networks (CDN) effective for serving "hot photos"
  - recently uploaded & popular
- But social networking sites also see a large # of requests for less popular (often older) content: long tail
- Requests from long tail account for a significant amount of Facebook traffic
  - almost all access the backing photo storage hosts as these requests typically miss in the CDN
- Haystack: each usable TB costs approx 28% less and processes 4x more reads per sec than an equiv TB on a NAS appliance

It turns out that Facebook etcetera I use something called content distribution networks. And these are used for serving hot photos recently upload and popular. Basically if you there are systems like akamai which are used by company Facebook to distribute their content. So, that users geographically distributed can look at the most closest server to pick up that focus for example. But a problem with these networking sites, social networking sites is that it seems that it they also see a large number of requests for less popular and often older content, this is basically what is called a long tail of the requests. They are unpopular things, but they are often used no wonder.

It seems that these requests are account for a significant amount of Facebook traffic and this will miss in the CDM. And basically the by reducing the amount of metadata they can essentially get most of the lookups quite fast and they are that is what being served by Facebook. The rest of this being handled through content distribution networks. They basically say that they can do quite well because of this they are able to do four times the reads per second than an equally that system NAS for example.

(Refer Slide Time: 57:34)

Summary

- Scalability inducing many cross-layer designs
  - Have to pay attention to overheads and remove them
  - Handling failures imp, so repl (or erasure coding) critical in design
    - Distribution of data a necessity
    - Coord of updates a necessity

So, basically you have this one is to summarize what I will we looked at with respective scalability. So, essentially scalability is a critical issue for web scale kind of systems, you really have to systematically look into cross layer designs. This cross layer designs are complicated basically because you are not instead of the time tested layering principles you are going to throw them away and do something new and typical that means, that it requires lot more careful attention to details and that is one part of the thing.

We saw it how consistency has been taken as one area to do cross layer optimizations. The other thing that we looked at also was to take care of inode, how to design inodes. Other issue is handling failures is very important and because distribution of data is a necessity and we also have to coordinate updates that also is critical.