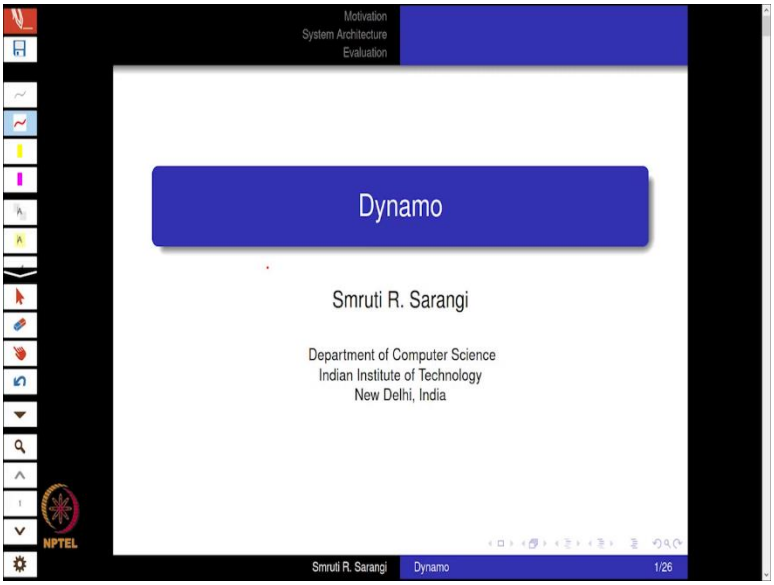
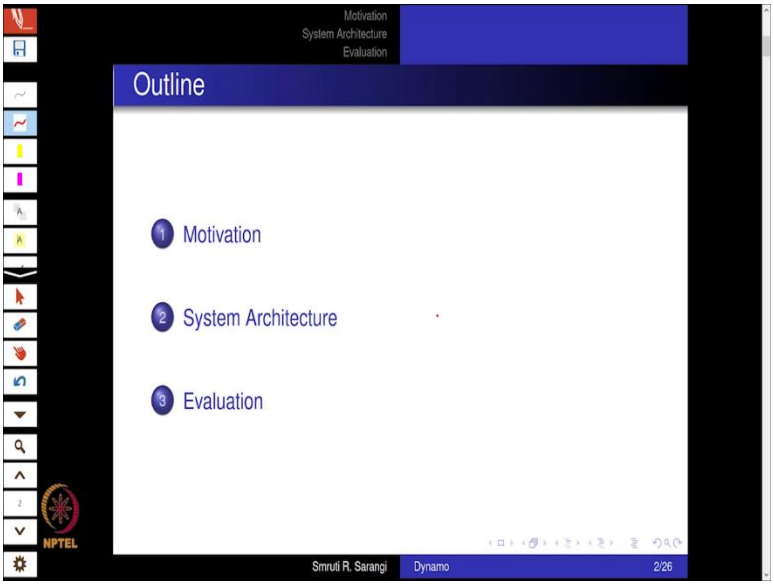
Advanced Distributed Systems
Professor Smruti R. Sarangi,
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture 18
Dynamo (Amazon's key value store)

(Refer Slide Time: 00:17)



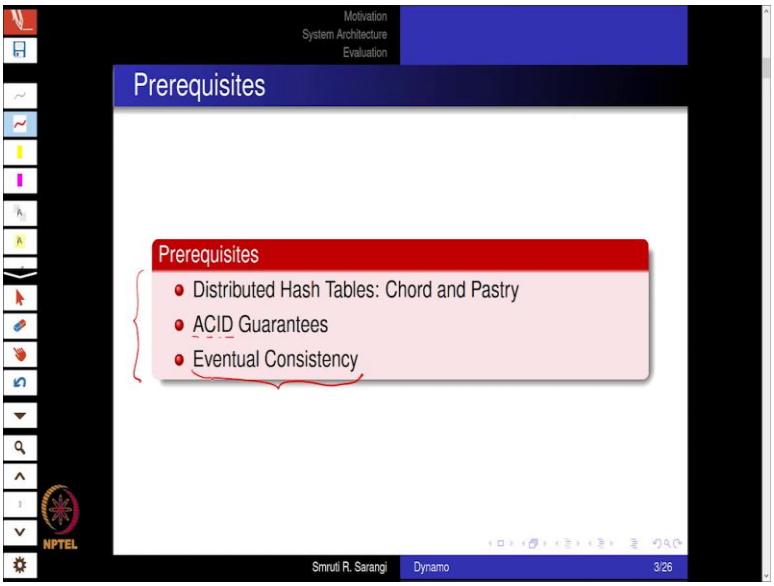
In this lecture, we will discuss the Dynamo system. The dynamo system powers many of the services within Amazon. So, this has been documented very well in their paper. So, we will discuss the dynamo system, which is essentially a key value store, it is a DHT, in this lecture.

(Refer Slide Time: 00:39)



So, we will have three main sections in these slides, the motivation, the system architecture, and the evaluation.

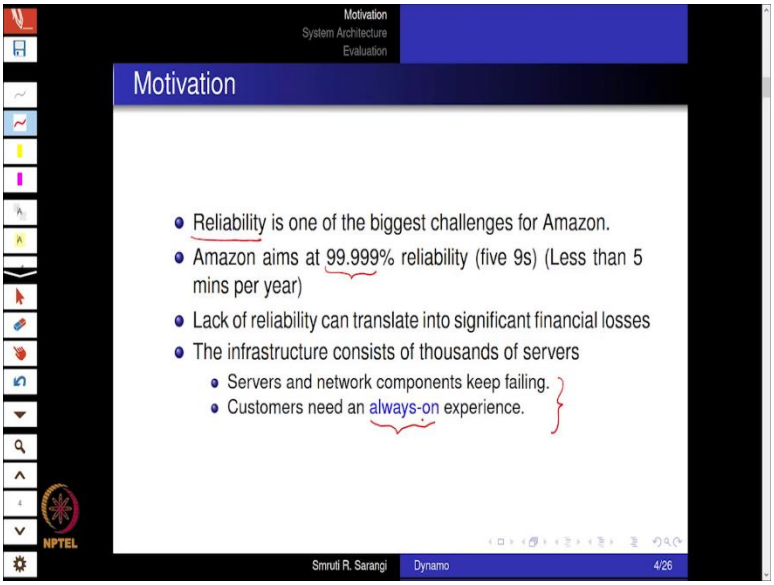
(Refer Slide Time: 00:52)



So, we have a couple of prerequisites before actually leading or viewing this lecture, or reading the slides or viewing the lecture. The prerequisites are that the viewer should have an idea of distributed hash tables, in particular, the chord and pastry algorithms. Without knowing these algorithms and without understanding how a distributed hash table works, it will be very difficult to follow the concepts in this paper.

Second, the viewer has to be aware of the ACID consistency, the ACI guarantees. So, ACID stands for atomicity, consistency, isolation and durability. So, these four concepts in the world of databases, they should be known to the viewer. And finally, the viewer should have some idea of consistency models in distributed systems, notably eventual consistency. Eventual consistency refers to a model where a right is ultimately visible. So, all rights are ultimately visible. This is eventual consistency. So, these three concepts should be known before proceeding forward.

(Refer Slide Time: 02:10)

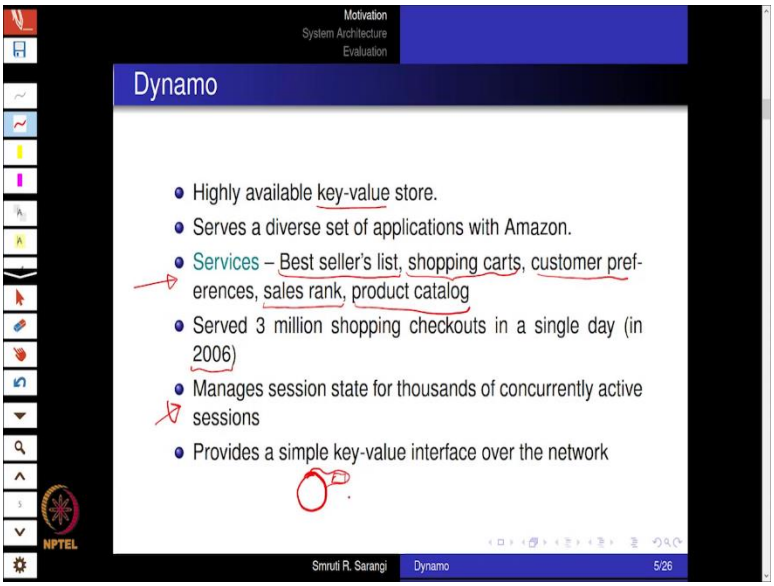


So, the background is like this that reliability was one of the biggest challenges for a large e-tailer like Amazon. So, in particular, Amazon aims at 99.999, so as you can see there are five nines over here, two before the decimal point and three after the decimal point. So, Amazon aims for such a high reliability. Well, so there are many reasons for it.

The first is that even a small downtime, particularly in the peak season, which is Christmas in the U.S., it is Diwali in India, so in such cases even a small downtime will essentially take away users. This will lead to a large loss in net revenue. And additionally, also the e-commerce site is not the only site that you actually, that actually is in Amazon. Amazon does many more things. So, Amazon has sites for web services, Amazon has a lot of sites for all the vendors that keep uploading the details of their products, Amazon has internal sites for payment to the vendors, for ads, for a lot of things.

So, in this case, given that Amazon is involved in so many things, there is a need to have a very high reliability. And so the infrastructure which consists of thousands of servers which are commercial of the self-servers, the server components, network components keep failing. So, the customers basically need an always on experience, which means that the platform that they are using should be always on, should we always active and always responsive.

(Refer Slide Time: 04:09)



So, for this purpose, Amazon needed a highly available key value store. And of course we have seen pastry and chord, we have seen the reliability guarantees that come along with pastry and chord, so they are great, but of course they are not that great that they can be used in a commercial system of the size of Amazon. So, what are some of the services that Amazon would like to provide? Well, some of the services are a best sellers list.

So, for example, I was just doing it right now that currently when this video is being recorded, we are in a lockdown in India. And so there was some news that e-commerce might open today, which it has not, but at least what I was trying to do is I was trying to see what are the new best sellers in the world of electronic gadgets. And thankfully, many sites like Amazon have a best sellers list which I can check.

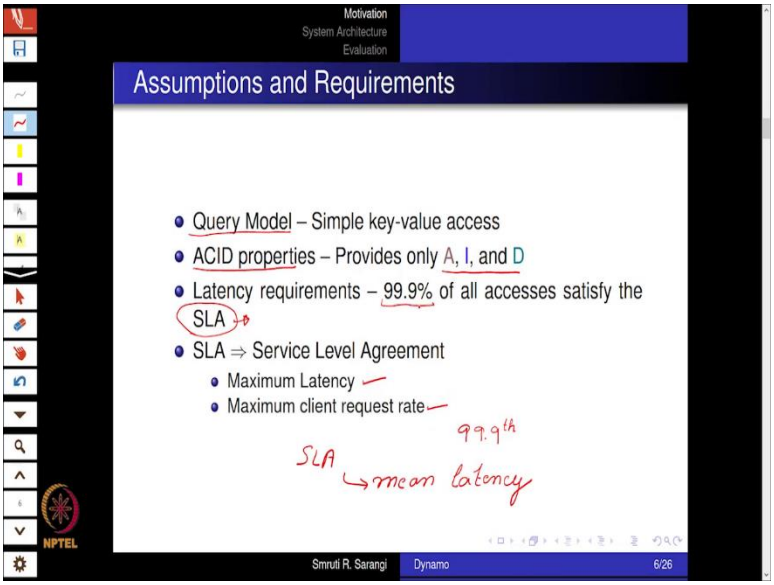
Then, of course, shopping carts, which are very important, so these are digital shopping carts, where whatever I want to buy, I add them to a shopping cart. Then we can have customer preferences. So, customer preferences are things that are bought before. And also you would have

seen that Amazon suggests a lot of products that I can buy based on my sales, based on things that I have bought before those kinds of things.

Sales rank, so the ranking of products, if let us say I want to buy a DVD player, it will show me the rank of the products and the product catalog, of course. This is something that many of the vendors see when they are uploading products. So, as I said, across users, across vendors, there are a lot of things, a lot of services that Amazon typically provides. And also these services keep on increasing. So, this paper is an old paper published in 2006. And after that Amazon has grown many, many times.

So, even in 2006, Amazon was still very popular. It served 3 million shopping checkouts in a single day. And it was managing thousands of sessions. So, the way that it was doing it was that it just had DHT. So, if you would recall from our discussion on pastry and chord, the DHT is a structure which can expand, which basically means that at the time of peak demand, let us say at the time of Christmas, what can happen is that we can take an additional data center and the DHT can sort of expand and the keys can get redistributed. So, the DHT can grow and shrink based on the demand, which is one of the key advantages of the DHT.

(Refer Slide Time: 07:06)



So, of course, here the assumption and requirements are that it is a large system with a simple semantics. So, we will provide a simple key value access. ACID properties, well, in a large system,

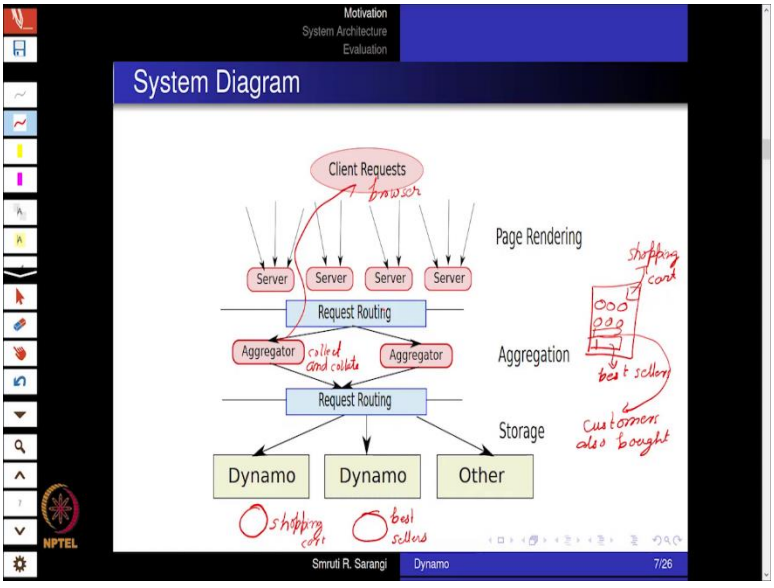
A, I and D, which is atomicity, isolation, and durability, these three properties have to be provided. So, Amazon does kind of relax the consistency requirement we will see how.

And furthermore there has to be some service level agreement, which means that, so here it is important for me to mention the dynamo from the point of view of Amazon is an internal service. It is not an external service, but it is a service which is internal, which means that other services within Amazon they use what dynamo provides. So, service level agreement is required even between Amazon and third parties and between one branch of Amazon and other branch of Amazon.

So, in this case, so the typical service level agreement, the way that it looks, talks about, let us say, the mean latency, that as a typical thing or maybe the maximum. So, Amazon, so what they went for, very interesting reasons are given in the paper. So, they looked at the 99.9th percentile, which to them is important, which clearly tells them that look 99.9%, which is one in thousand, so they look at 99.9th percentile, and 99.9% of all accesses have to satisfy the SLA.

For example, the SLA prescribes a certain maximum latency. Then it is clear that 999 are of 1000 requests need to satisfy that. And if there is a maximum client request rate, then that also needs to satisfy the SLA with the same 99.9% criteria.

(Refer Slide Time: 09:29)



So, as we have discussed in a single Amazon page that we will see, if you search for something, is actually a very, very complicated process. So, the way that Amazon actually works is that it calculates, it gets the client requests. So, they go to, of course, a multitude of servers. The servers, what they do is, once they get the client requests, so the client requests can be anything. It can be a search query and the client might want to see the list of products that Amazon has or the client might want to just update his or her shopping cart.

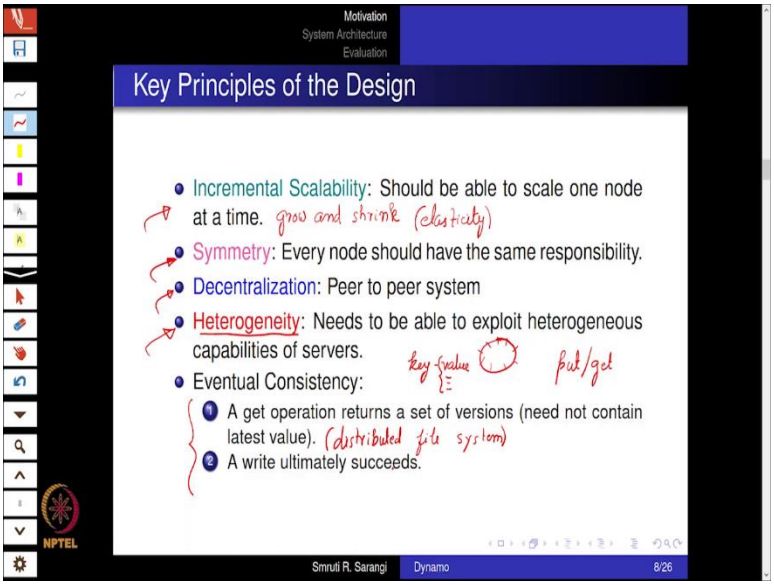
So, there are a set of servers called aggregators. What they do is that they collect and collate requests. So, they, in a sense, collect and collate and aggregate these requests. So, they actually route the request to multiple dynamo instances. And each of these dynamo instances manage different things. Maybe one manages the shopping cart. One instance of dynamo might manage list of best sellers, might work as a recommender system. So, they have different instances. So, the different instances give back the results. The aggregator aggregates them.

And so then what happens is if we consider a typical search page, so Amazon might have a list of search items over here. It might have this spot which is like best sellers in this category. It might have another part of the page, which is this, which could also be customers also bought. So, we also see that, that customers also bought similar items. And then of course here there will be details of the shopping cart.

So, pretty much each of these pieces of information are being produced created by different dynamo instances and these dynamo instances produce this information, which is subsequently aggregated by these aggregator servers and then a web page is created. And the webpage again comes back to the client. And the client can see the structure of the web page on the browser.

So, it is important to understand that a single, so whenever we click a button on Amazon, we see a web page. And this web page is actually being produced by a very, very complicated process, where a lot of information is being collected from different sources within Amazon. Everything is brought into the same webpage and shown to the client.

(Refer Slide Time: 12:46)



So, what are some key important broad principles that we would like to have? The first is incremental scalability, which means it should be possible for us to increase the size of the DHT, one node at a time. And so this is needed. So, the key property of the DHT is that it can grow and shrink. So, the reason we say it is a very good property is because essentially, when at the time of high load, we can take cloud computing resources, and we can add it to the DHT. And at the time of low load, these resources can be released.

So, then Amazon can, for example, give its compute power off to individual users, where they can use it for, something like Amazon Web Services. They can use it to do their assignments, run projects, and so on. So, a DHT is gives us a certain amount of elasticity. And the elasticity is a key, a very important property of DHTs. Then of course symmetry, which means all the nodes do the same thing. We have already seen in pastry and chord that this is the case. This is indeed a peer to peer system. So, this is a type two DHT decentralization.

Heterogeneity, which means that the heterogeneous capabilities of servers that also needs to be exploited, because one thing you need to understand that whenever such the DHTs are created, which are very large, which can consist of tens of thousands of nodes distributed across many, many data centers in the world, all the nodes will not have the same compute capability. Some might be older processors, some might be newer processors, or might be really small ones, big ones, even CPUs, GPUs, FPGAs, all kinds of nodes will be there on the DHT.

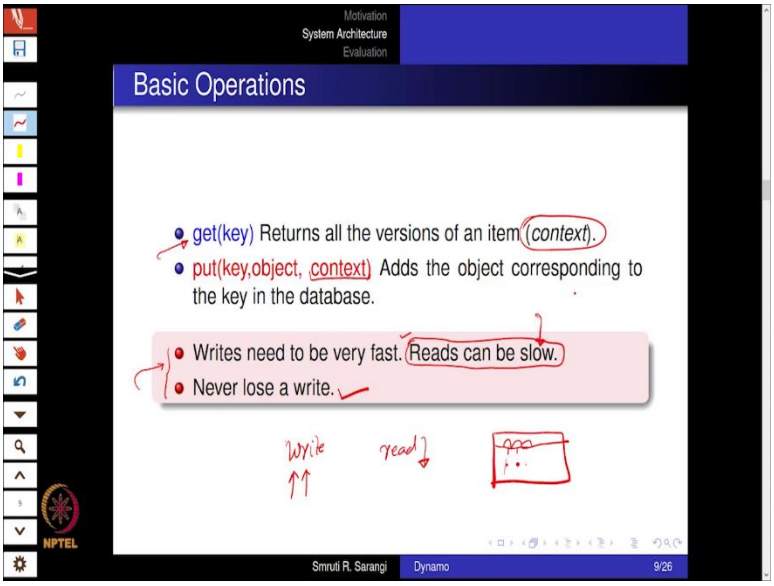
So, essentially ensuring that all of them would actually execute at the same rate is, at the same rate with the same service level guarantees is not a wise idea. So, we should be aware that heterogeneity exists. And finally, an eventual consistency, which means two things, in this case, particularly that, so recall that in a DHT we have two operations put and get. So, a get operation returns a set of versions of the same variable.

So, put, of course, as a key and a value. But given we have eventual consistency, which means it is possible that different nodes will be seeing different values because of the weak consistency, which means that there will be multiple ones of these with their separate versions. So, the get operation will fetch all of those versions.

So, if you would recall, we were doing something very similar when we discussed distributed file systems, like coda. So, in that case, also we were fetching the different disparate versions that are there, and then there was a merge. And this is something very similar to a distributed file system. So, I would request the viewers to take a look at the coda distributed file system, which is a part of the same playlist. So, that discusses the versions issue in more detail.

And the other is, of course, as we have discussed, a write ultimately succeeds. So, it does not take infinite time for a write to actually happen. So, it ultimately happens. But then of course multiple versions have to be created. And we are explicitly aware of that.

(Refer Slide Time: 16:48)



So, there are, as I just mentioned, there are two key operations get and put. So, get returns all the values associated with a key with additional information. All of that information is referred to as the context. And similarly put inserts a key value with a given context into the database. So, the context thing will be clear. It is actually a set of vector clocks. But this will be clearer as we keep on discussing.

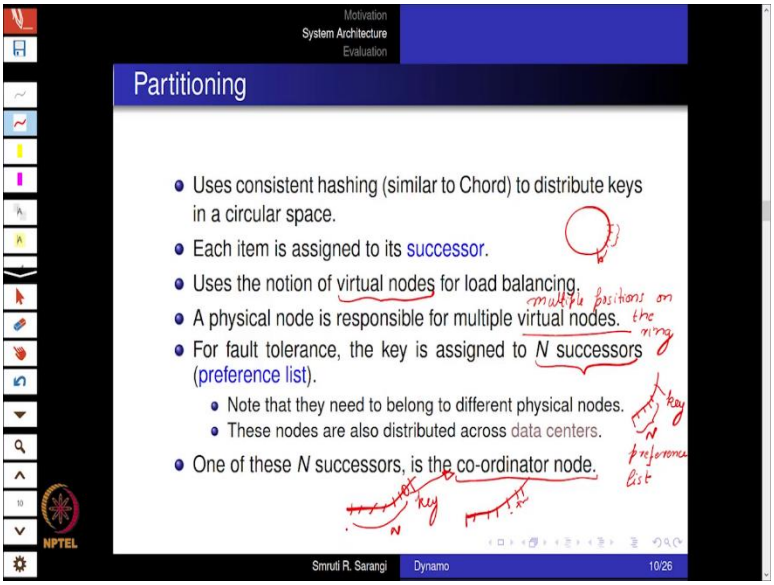
So, the important conceptual things over here is that writes need to be very fast. Writes need to be very, very fast. So, this is a design decision. And we will see why. And reads can be slow. Writes have to be fast, reads can be slow. And other is that we never lose a write. So, in the priority order, we have essentially increased the priorities of writes significantly. And for reads, we have kind of deprioritized them. So, the reason why most e-commerce sites do it, in fact, most customer-centric, customer-friendly, client-friendly sites actually do it, the reason is rather clear.

The reason is that, so consider a shopping cart. In this case, we want to get the customer clicks customer preferences as soon and as quickly as possible and just record them. So, which basically means that, let us say, I have a browser of open over here and I have multiple tabs. This is tab one, this is tab two, and kind of Amazon is open in all of those tabs. So, here I click something in this tab, then I click something in this tab, then I click something in this tab.

So, I am basically adding things to the shopping cart. But of course they are happening via different tabs. So, we never want to lose or overwrite any of my preferences. And I might close the browser in the middle, I might then open up my Amazon site on my mobile on my mobile phone. So, whatever I am doing, I never want to lose any write, which means I never want to add something to the shopping cart or I never want to change my, forget my preferences.

And the other is that always we want to record a write. So, writes are not on the critical path. But that is the reason we want to quickly record them. And when we do a read, we will be well aware that there are many different versions floating around, because we just quickly recorded writes without synchronizing. So, at that point, some version management and enforcing some synchronization and consistency for all the writes that we have gathered that is important. That is why in this case reads are slow and writes are fast.

(Refer Slide Time: 19:44)



So, how does this work? Well, the way that this works is that this uses consistent hashing, which is similar to chord to distribute keys a circular space. Similar to chord each item is assigned to its clockwise successor on the ring. Furthermore, what chord did was that on each physical node, it actually kept multiple virtual nodes. In the sense what it actually did is that it created, it assigned a host of positions on the ring. And each physical node was given a set of contiguous virtual nodes for the sake of load balancing.

So, we will see that the virtual nodes are used for in a conceptually similar manner over here, but of course not in this way, in a different way. But we just need to keep in mind that a physical node is responsible for multiple virtual nodes, which means multiple positions on the ring. So, they were contiguous in chord, but they need not be contiguous over here. So we will see.

So, a fault tolerance what we do is that the key is assigned to N successors called the preference list. So, if this is the part of the ring and these are all our successor nodes. And in a sense if our key is mapping over here, so we assign the key to N successors, also known as the reference list. So, in this case what happens is that there is a degree of redundancy in our system, where N successors all have the key and the value such that if any one of them is down, then of course the other ones can supply their data.

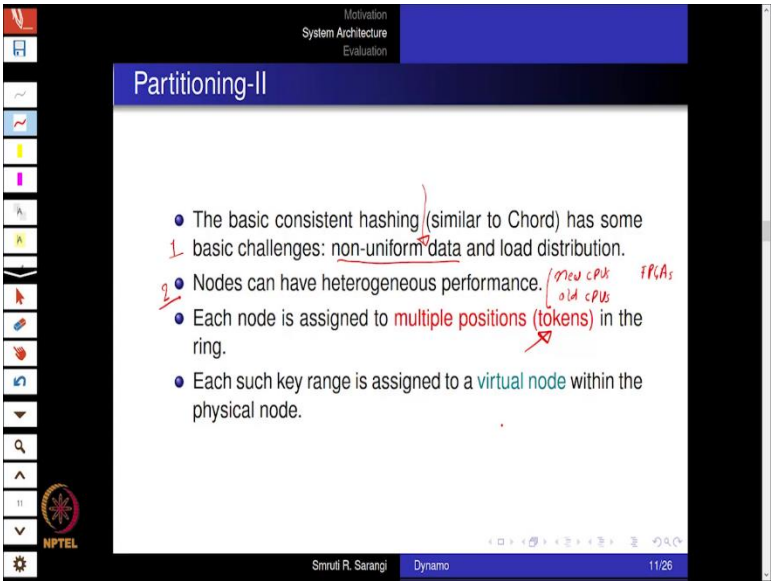
So, furthermore, what dynamo does is that it tries to distribute them across different physical locations where these physical locations are on different data centers. So, of course, now if you

actually look at the key space, I am just drawing a part of it, so it is pretty much possible that maybe these two nodes are on the same data center. But then what dynamo would do is that it would skip the second position and instead consider this as the second successors, the first successor. The next one, which is a different data center, is the second successor.

So, essentially, when we are talking of N successors, that N successor is located in different physical nodes, which are preferably in different data centers. So, N successors should not be interpreted as N successors on the ring, rather they are successors on the ring, the closer successors on the ring. But of course, if any two of them are in the same data center, we kind of skip the second, the second, third and so on such that we want to ensure that all N of them are physically located at different places. So, this is what gives us the fault tolerance.

And one among these N successors is the coordinator node, which is primarily responsible for the key and value. So, one among them in this preference list, so this is where the key maps, this is the preference list, one among them might be the closest one also is the coordinator. And the rest just contain replicas of the key value data.

(Refer Slide Time: 23:48)

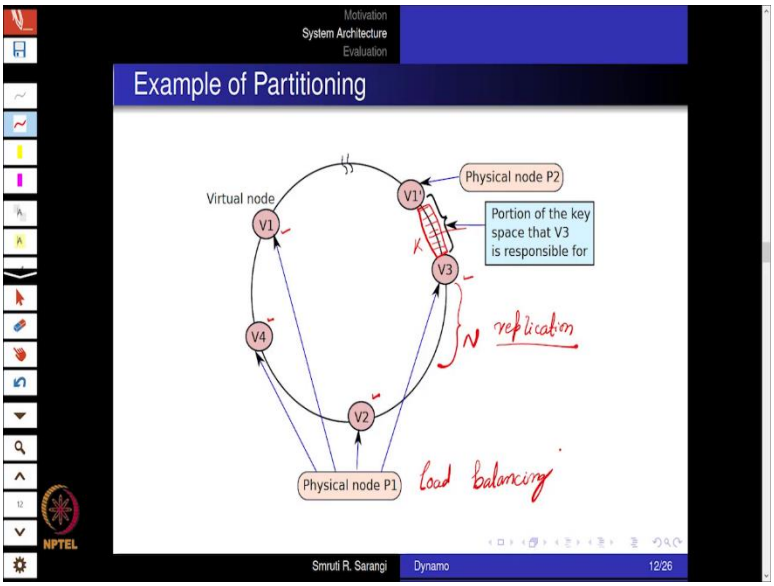


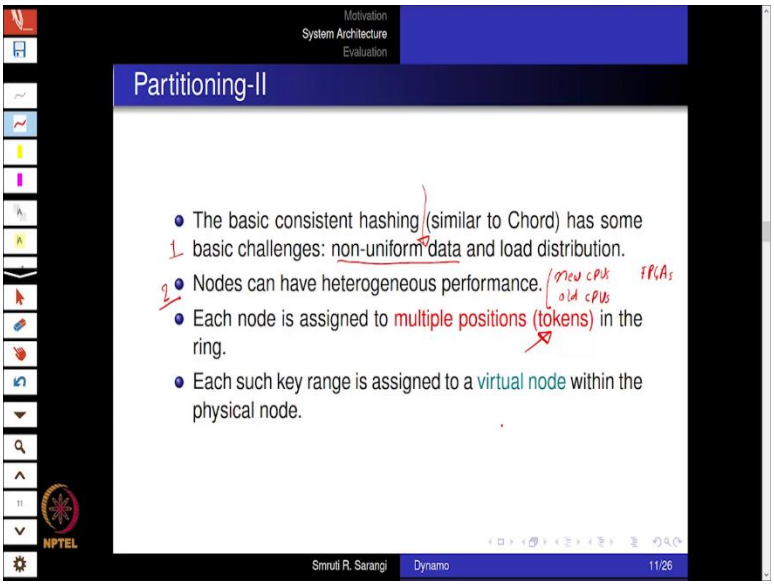
So, what is the idea? Well, the idea is that the consistent hashing algorithm in chord has, still has some issues with regards to non-uniformity of data. And why is this? Well, the reason is that on the ring you will have, even though you are using a very good hashing algorithm, you will have regions which are more populated with keys and regions which are less populated. This is reason

one. Along with this there is a reason two that the nodes themselves have heterogeneous performance. They are not of the same type.

So, some are new CPUs which are fast. Some are old CPUs. We can have ultra-fast FPGAs with very high throughput. So, of course, latency might be low, but throughput will be very high. We can have other kinds of futuristic hardware, custom accelerators, and so on. So, what we actually do is that each node is assigned to multiple positions. These multiple positions are also called as tokens on the ring. And each range of keys that position is responsible for, which means it is a successor off, is assigned to a virtual node within the physical node.

(Refer Slide Time: 25:20)





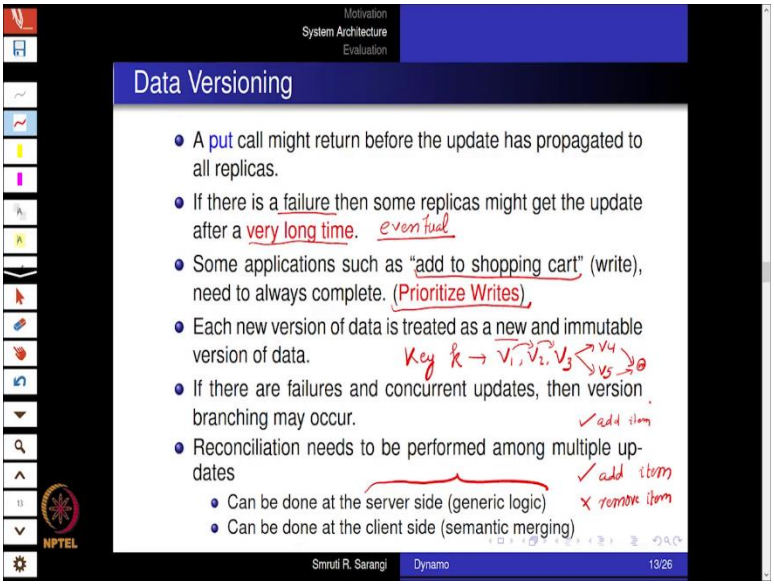
So, let me see I have a small figure over here that explains the same concept. So, what we have here is we have a ring. We have a ring where all our hash functions are distributed. So, given the physical node P1, as we have just described, we assign it to multiple positions, also called tokens in the ring, which means that P1 is assigned to different virtual nodes, virtual node V1, V2, V3 and V4.

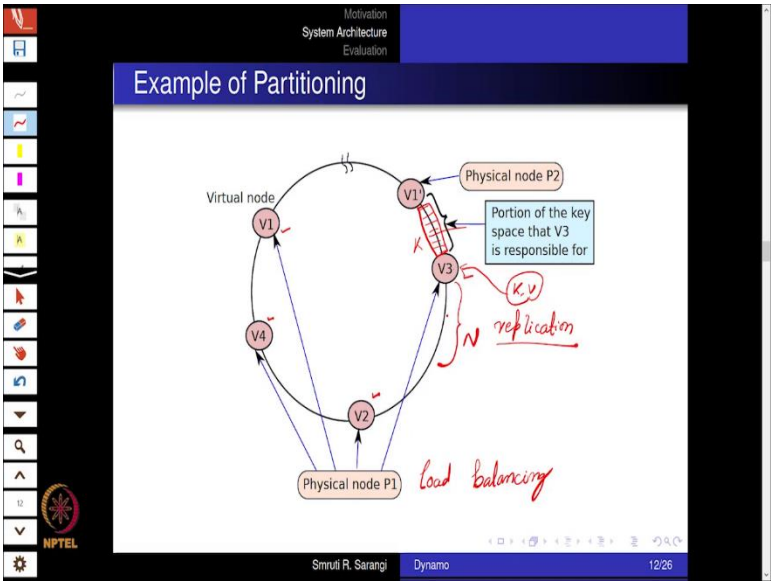
So, let us now assume that for V3 the node that is closest to it, the virtual node that is closest to it on the ring is V1 dash, which corresponds to a physical node P2. So, for this region that you see over here, this region of the hash space, this is the portion of the hash space, the key space that node V3 is responsible for. So, what we have essentially done is that we have taken the key space, which is a very, very large circle, and in the key space we have assigned multiple positions, multiple tokens to each physical node P1, each of these positions is a virtual node.

Furthermore, what has been done in this case is that the key space has been implicitly partitioned, which is a chord like partitioning, because any key that maps to this region over here is logical successor in this case is V3. And of course, since we store replicas of this key, any key let us say over here, we store replicas, they will be stored in the N replicas that succeed V3 and subject to the fact that these replicas are physically stored in different places.

And so then this will give us a degree of replication. And also the fact that we are actually mapping each physical node to random positions on the ring will also give us a degree of uniformity and load balancing. So, this is the key difference between dynamo and chord.

(Refer Slide Time: 27:40)





Let me just, data versioning. So, it is possible that a put call, when I am trying to put key or value on the key space on the ring, it might return before the update has propagated to all the replicas. So, this is of course possible that we have N replicas which are physically in different places. If I send, and let us say that this is the coordinator for a certain key, so I send it the key and value, so it needs to be sent to the rest of the replicas. So, of course, note that this N includes V3 also. So, it needs to be sent to the rest of the N - 1 replicas. And this takes time, because you are talking of physical network messages that need to be sent by V3 to the remaining N - 1 replicas.

So, in this process if there is a failure, some network failure, node failure, then some replicas might get the update after a very long time. So, this is where your eventual consistency plays a big role. So, we consider a put to be successful if at least one of the servers within the preference list records it. It is the job of this server, which is typically the coordinator, to broadcast this to the rest of the servers on the preference list. And this, there is no time limit, there is no associated timing guarantee associated with this. Hence, we see it is eventually consistent system.

So, some applications such as add to shopping cart is where a write, which means that I want to buy some item, that needs to always complete and always needs to complete quickly. So, we have discussed this before and this is exactly why we prioritize writes. Each new version of data that we create, so let us say consider a certain key k and then it can take a succession of values, so it is a general key value store, it can take a succession of values, where if k is the shopping cart as we are adding or deleting new entries, and V1 is the contents of the entire shopping cart, it is taking a succession of values. Each one of them is considered a new and immutable, unchangeable version of data.

So, what we do is that we do not actually overwrite. We just create a new version and record the fact that V1 was the first version, then V2 and then V3. We will see that this simplifies our life significantly. Now, what happens is that if there are failures and concurrent updates then a certain version branching may occur, which means after V3 some might record a V4, some might record a V5. So, of course, this should not happen, but this will happen if there is a failure.

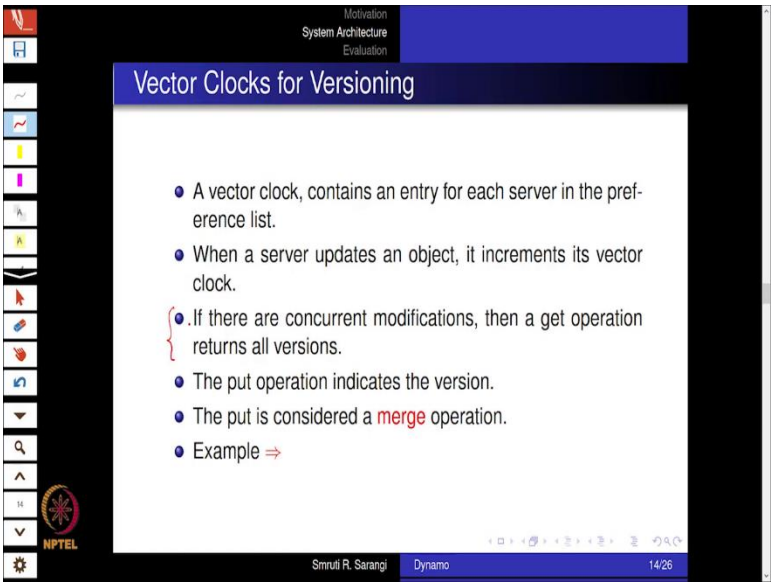
So, at the time of reading data or at the time of final checkout, there is a need to actually merge V4 and V5. And this merging can be done in two ways either generic logging which means at the server side, is a general way of merging. So, one way of merging shopping carts is that look, if I added two items at different points in time and I have not removed them, then maybe I want to buy both, so I just can merge them.

However, as we have seen, so this is very similar to what we had discussed in coda. So, in coda what we had seen is some of these updates are not possible to merge. So, it is possible that in one server records, let us say add an item, another one records remove the same item. What do we do? If they are roughly in a same window of time, we are never sure customer actually wanted to add it or add it and then remove it or what was the intension.

And so basically this can further get complicated. Of course, let us assume, previously there was one add item and then we add one more, so the relative timing of add and remove can make it complicated because we would not know which one came first, which one came later.

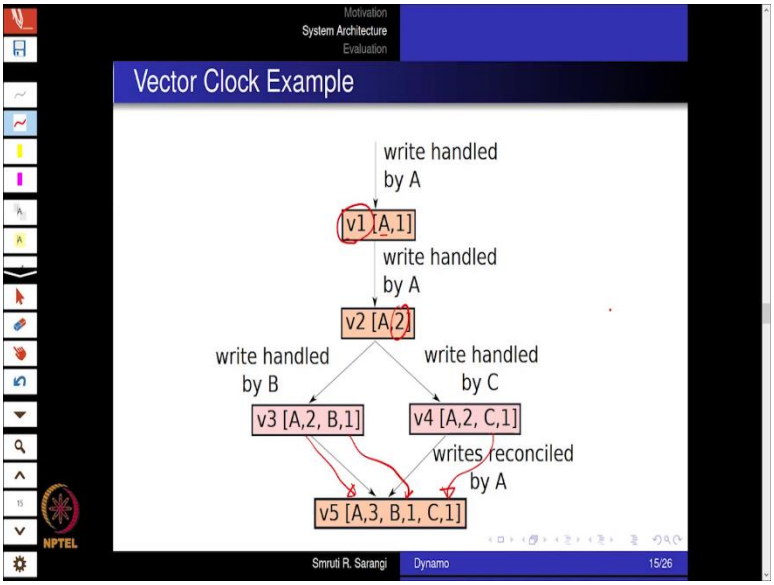
So, in some cases, it can matter. In some cases it will not matter. But in these cases, it might be a good idea at the client side browser actually tries to resolve the issue on its own, particularly if there is an issue that there is conflicting information and this cannot be resolved automatically on the server side.

(Refer Slide Time: 32:56)



So, the way that we manage versioning is that we use a vector clock. A vector clock contains an entry for each server in the preference list. So, whenever a server updates the object, it increments its vector clock. So, this is very similar to CVV in coda. And if there are concurrent modifications, then what will happen is that a get operation will return all the versions which it should and after that some kind of a version merging reconciliation is attempted.

(Refer Slide Time: 33:32)

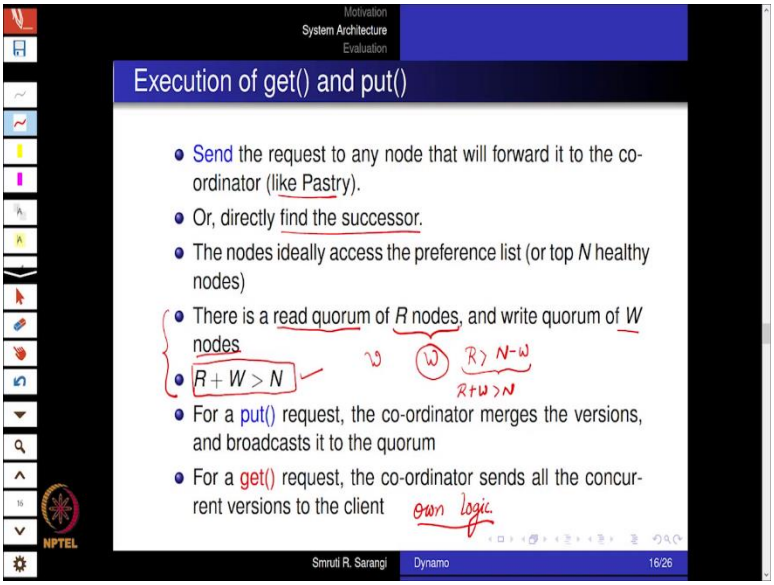


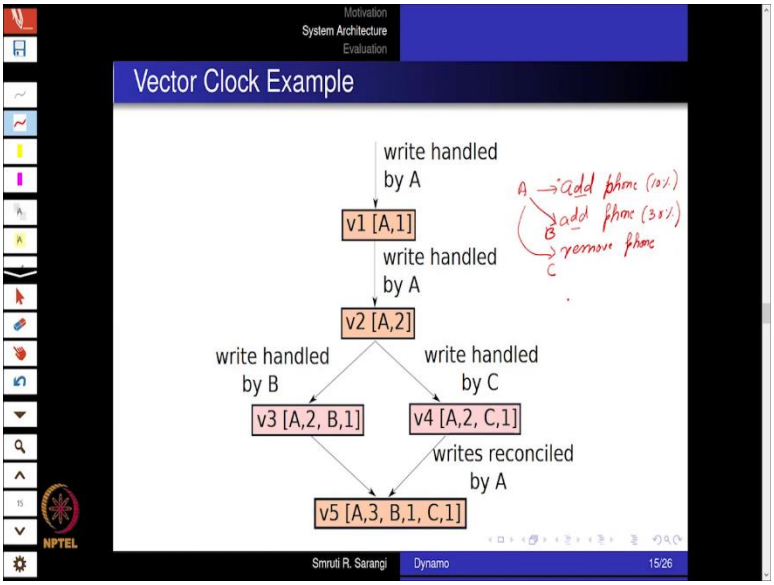
So, this is how an example would look that let us say there is a write operation which is handled by server A. So, the first vector clock v1 would look something like this that this is version 1 created by server A. Then again if there is another write that its handled by server A, it will just increment the vector clock and make it A2. After this it is possible that some failures or some branching may occur in the network. So, it is possible that a write is handled by B.

So, in this case, the vector clock would be A2, B1 and we can have a simultaneous branching where the write is handled by C. In this case, it will be A2 and C1. Then of course, there is a need to reconcile the writes. So, when we reconcile the writes, we reconciles the vector clocks as well. And if A is doing it, it will increment A2 to A3, this we can see over here, and the rest of the entries just get copied. So, this basically means that any subsequent version will have to emirate from here and this version has essentially reconcile all the parts.

So, using vector clocks for reconciling conflicting data, for reconciling divergent data, is a common pattern in distributed systems and this is almost, so we will see that this is a common pattern, it is a recurring pattern and almost all such systems use it to varying degrees.

(Refer Slide Time: 35:13)





So, how do we exactly do a get and put? Well, so similar to pastry, we send a request to any node that will forward it to the coordinator. So, basically if we do not know the coordinator, we try to find the coordinator or given the key we try to directly find the successor of the key. Then the nodes would ideally access the entire preference list and send or receive data from entire preference list. So, the preference list in itself is organized in a kind of special way.

So, we define a read quorum and a write quorum. So, quorum is like an extension of the notion of a majority. So, let us say that if you consider any voting system, if a majority of a nodes agree for doing something, then we say that that is a decision of entire group. So, in this case, when I am

talking of a read quorum of R nodes, this basically means that I read the data from R nodes and that is sufficient to tell me what the data is. And if there is a write quorum of W nodes, it means whenever I do a write, I need to send it to W nodes.

So, the classical equation that we hold in all, that we, that holds in all quorum systems, is R + W > N. So, we will see why this is the case. So, in this case what happens is that, if let us say consider the fact that I am writing and I write it to W nodes. If I write it to W nodes, when I am reading, I should at least reach one of these nodes, and so if I reach one of these nodes, I will get the update.

So, what is the probability, so when will I not get the update? I will get the update only when I reach one of the other nodes, which is N - W. But if R > N - W, then it automatically means that in my read quorum there is at least one server, which is also there in the write quorum. And since I am reading from it, I am guaranteed to get the most up to date version of the data. So, given R > N - W, we can automatically infer R + W > N.

So, dynamo does use a quorum system in the sense that all N servers of the preference list are not all used in the same way. So, we can designate some as read nodes and some as write nodes as long as this condition holds. So, for a put request the coordinator merges the versions, if it can, and broadcast it to the quorum, the write quorum. For a get request, the coordinator sends all the concurrent versions to the client, and then the client needs to do a merge. So, it can be the client or it can be some other Amazon node on behalf of the client.

So, there are many ways of actually looking and thinking about this. So, but, I mean, as I said, there are many different kinds of complicated logics that can be implemented on a dynamo system. And so we do not want to constrain it in any way. So, of course, as I said, if the server or somebody else can automatically merge the conflicts, it is the best. Otherwise, the client needs to do it.

So, let me come back to the example that I had in the previous slide. It is kind of a tricky example. So, a little bit of re-explanation might be necessary. So, the one of the examples that I had said over here is this add and remove item thing, which I would like to maybe just use some of the free space over here and issue a quick re-explanation or quick clarification so to speak. So, let us construct an example which need not be all that practical, but just kind of will give you an idea of the kind of things that we need to deal with.

So, consider the shopping cart and let us say that I added an item. So, let us say, I add a phone. So, this phone is the same model. It has a 10% discount, and then I add another phone same model with a 30% discount or it is, so let me do that, and then I just say remove phone and I do not specify which one it is.

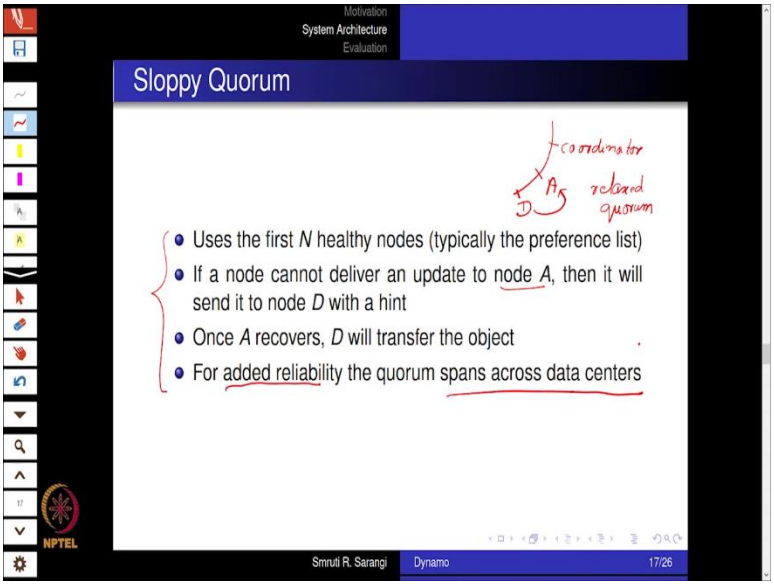
So, if I do that, so if you were to actually see we can assuming we have this logic, so of course there are many ways of ensuring that this does not happen in a sense we can give both the phones different internal IDs, and the remove would refer to one of the specific internal IDs. But let us assume that we have an implementation, where we are just saying that I had phones. And then I just want to like to remove one of the phones with this model number.

And so, this can happen not from a client side, but maybe from an internal Amazon user site, maybe one of the phones suddenly went out of inventory so that would get removed. So, now the question is which one? So, let us say that this happened, this was version A and then a version branching took place, so then this became B and this became C. So, we would not exactly know the relative ordering between B and C. So, if C comes first maybe this B will be removed.

If C comes first A will be removed this one and if C comes later B will be removed. So, in many cases for concurrent events we are not sure of the ordering and that is why automatic merge followed by a manual merge is required. And so this merging can lead us to very tricky complicated scenarios. So, we have already seen how to handle them in coda. So, in Amazon, in dynamo, the philosophy is slightly different.

So, the philosophy of dynamo is that we send all the concurrent versions to the client. And the client can either be the actual user or some internal Amazon node which is between dynamo and the browser and then the browser can use its own logic or the internal node can use its own logic to resolve these conflicts in some way. So, that is not really the lookout of the DHT.

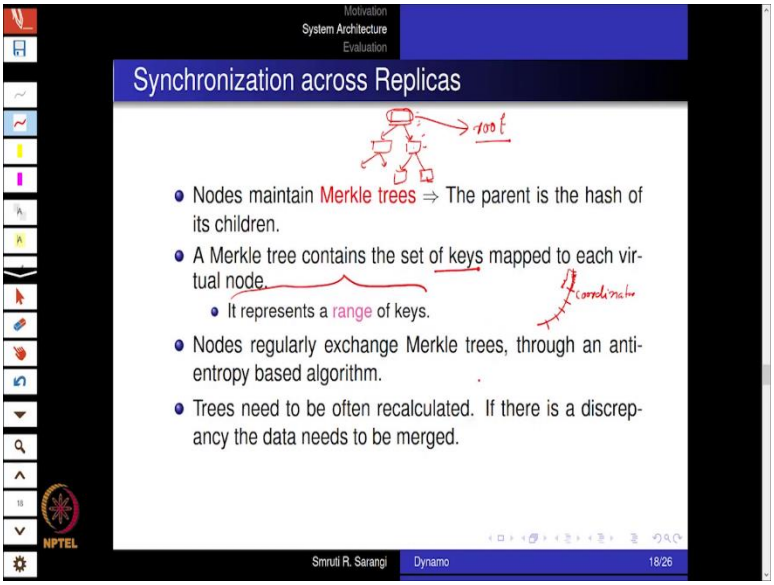
(Refer Slide Time: 42:45)



So, furthermore, dynamo defines a new concept called a sloppy quorum. So, the sloppy quorum, it uses the first N healthy nodes, which is typically the preference list. So, assume that a node cannot deliver an update to node A. So, then what it does, so let us say, we have this part of the ring and then this is the coordinator. It wants to send an update to node A, but it cannot deliver it.

So, in this case what it does is that it actually delivers the node to another node D and then it ask D, it drops a hint to D that once A recovers, D will transfer the update to A. So, this is known as a sloppy quorum, because this is not an exact quorum, because the coordinator will still record the fact that A got it even though A actually did not, but we can think of this as some form of a relaxed quorum. For A need not get it immediately, but at a later point in time some other node which has better access to A can transfer the object to A and kind of give the coordinator a promise back that it will do it. Of course, for additional added reliability, this quorum should span data centers, such that our system is as reliable as possible.

(Refer Slide Time: 44:21)



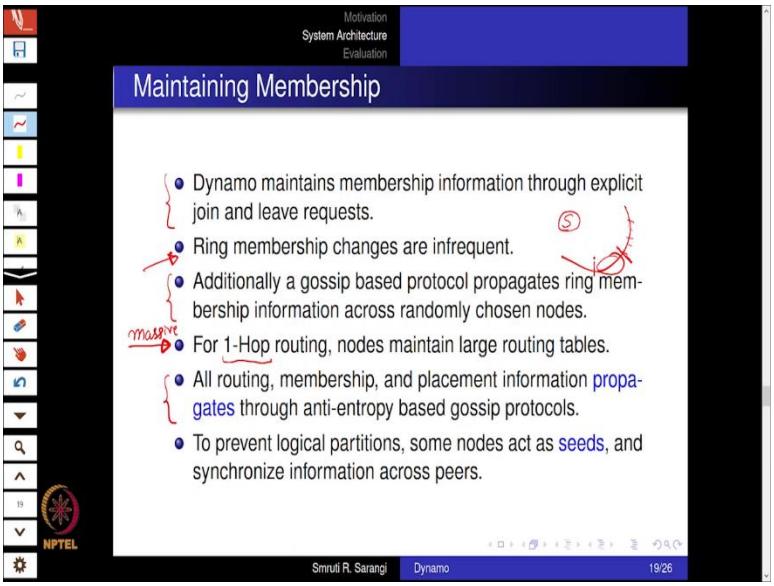
Now, let us discuss more about how we do synchronization across replicas. So, recall Merkle trees, so Merkle trees we had discussed in our lecture on Bitcoin and blockchain. So, just a quick recap of Merkle trees. So, this is a regular tree. So, the leafs are the ones that actually contain real data, then each internal node, each parent just contains the hashes of its children. So, each of these internal nodes just contains the hashes. And finally, the root of the tree contains the hashes of its immediate children.

So, if you think about it, if I change the value in any leaf, all of the hashes till the root will change. So, the root in a sense contains a gist of all the data that is contained in the entire system. So, if I just re-compute the Merkle tree and just compare the contents of the route to the Merkle tree, that is equivalent to containing, that is equivalent to comparing the contents of the entire data that each tree represents.

So, in this case, the Merkle tree contains the set of all the keys that are mapped to a virtual node. So, it essentially represents a range of keys that a virtual node stores. So, nodes regularly exchange Merkle trees via an anti-entropy based mechanism. And so why is that done? Well, the reason that is done is that let us see if I have the coordinator over here. And then I have a set of replicas, which also contain the data for this, the data for this zone. So, it is possible that over time because of lost messages in the network, the data can go out of sync.

So, in this case, if the data goes out of sync, then there is a problem. So, that is the reason for this zone, the entire data is sent via an anti-entropy kind of mechanism. So, nodes regularly exchange Merkle trees to ensure that the still, the replicas are consistent with the coordinator. So, trees often have to be recalculated. And if there is a discrepancy, the data has to be merged.

(Refer Slide Time: 46:58)



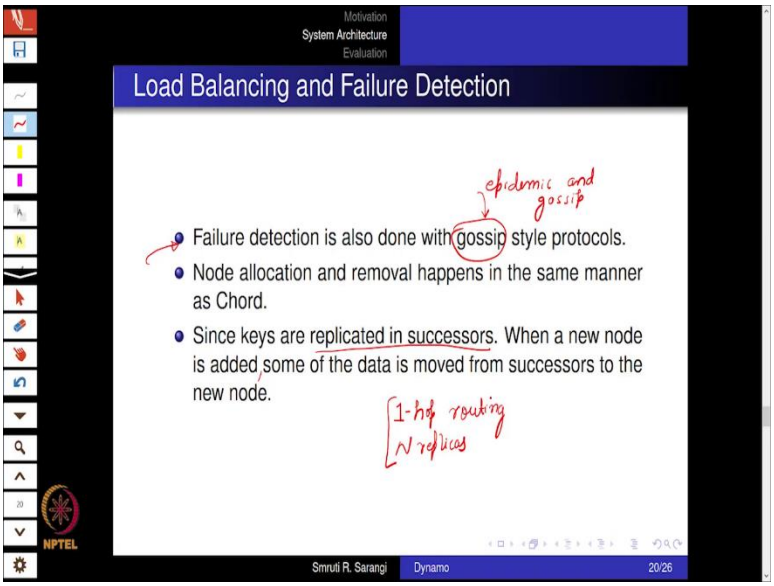
So, how does dynamo maintain membership? So, recall we had discussed virtual synchrony in a previous lecture, where we talked about view changes. So, we do have something here which is more or less conceptually similar. So, dynamo maintains membership information through explicit join and leave requests. So, as compared to other systems where it is far more common, in dynamo particularly this is a slow operation. So, ring membership changes are infrequent.

Additionally, a gossip based protocol, which we have seen different forms of gossip, exchange ring membership information across randomly chosen nodes. So, now, why is this important? Well, the reason it is important is that in dynamo the routing tables are massive, there are huge routing tables, where we aim for pretty much 1-Hop routing.

So, what I mean by 1-Hop routing is that unlike chord and pastry where it is a login step routing, here in dynamo it is a single hop. What that means is that I have very large routing tables and I try to reach the node immediately in one step. So, all routing membership and placement information propagates again via an anti-entropy based gossip protocol among all the nodes. And they try to keep the membership information roughly consistent.

Of course, it is possible that the logical partitions in a ring can get created because of lost messages, so which basically means that these nodes have no idea about nodes that are in another partition, so this information got lost. So, we have some seed nodes that act as global directories, which have exact information about each of the nodes in the network. So, you can think of the seed nodes as kind of like global aggregators of information of when a node joins and when a node leaves. So, these global seeds can be consulted if nodes feel that there is a partition, and that can be used to reconstruct the parts of the ring.

(Refer Slide Time: 49:30)



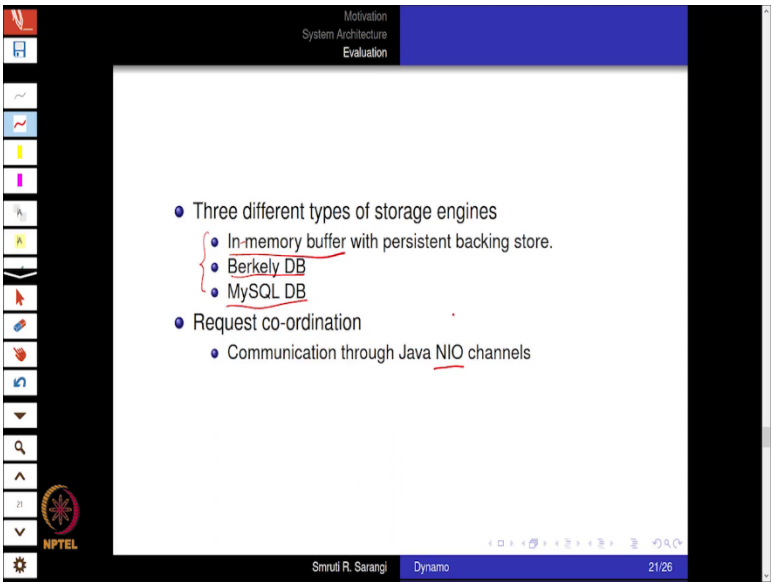
So, failure detection is also done via gossip style protocols where we have messages and acknowledgments and heartbeats. So, if you feel that a node is unresponsive for a long time, a node, the other nodes can declare it to be fail and exchange this information via gossip. So, of course, here, the assumption is that epidemic and gossip algorithms are known to the viewer. If the viewer does not know about them, then they can look up the course webpage and so they will find the slides and a video on this is also coming up.

So, node allocation and removal happens in the same manner as chord. So, I am not going over it. And so since it us very similar when a node fixes its own routing table and also adds itself to the routing table of other nodes and same is true for removal as well. So, here, since keys are replicated at the successors, when a new node is added, some of the data is moved from successors to the

new node. So, some data movement does indeed happen. So, that is why node addition and deletion, particularly in this setup with 1-Hop routing.

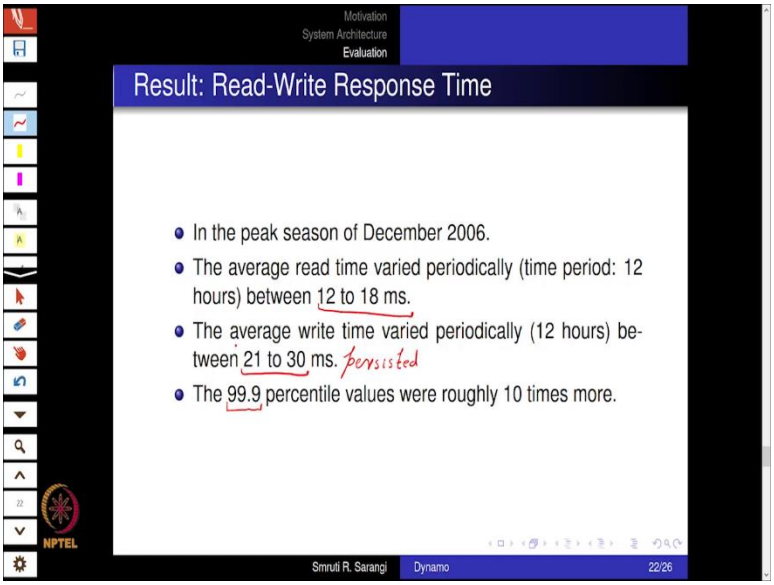
So, there are two things that actually make dynamo kind of heavy and expensive. One is 1-Hop routing. The main reason was that in Amazon when we are browsing it, when we are just searching for new products and we are adding it to our catalog, adding it to our shopping experience, we want a very quick response from the website. That is the reason 1-Hop routing is required in dynamo. And the other is of course that we maintain N replicas. So, both of these things make things slow and make the protocol heavy. And so that is why nodes need more storage.

(Refer Slide Time: 51:36)



And once when nodes need more storage, then what happens is that any addition or deletion means you have to move in a lot of data. So, given the data aspect in mind when dynamo was actually evaluated, they compare it with several kinds of, they actually implemented it with several kinds of storage in, several backing stores and in memory buffer, Berkeley DB which is meant for storing very short files, and a traditional MySQL database. And the entire system was written in Java with Java NIO channels, which are very fast IO channels, very fast implementation of IO channels in Java.

(Refer Slide Time: 52:23)

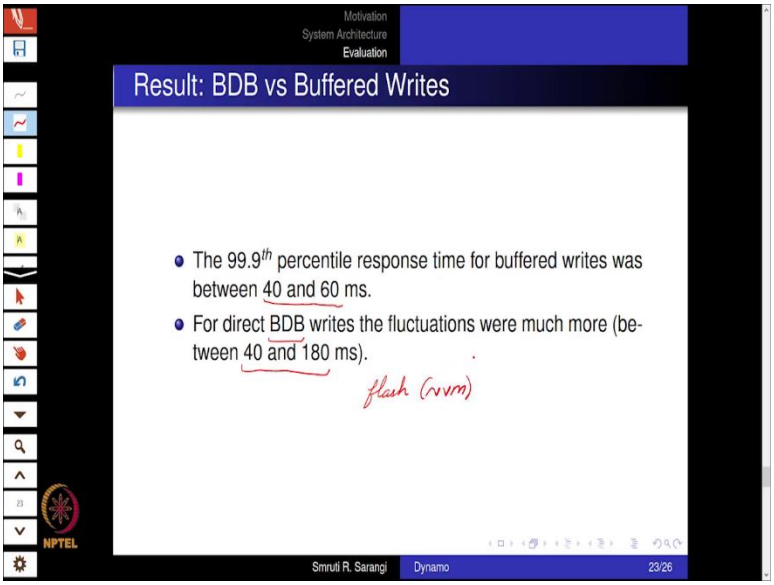


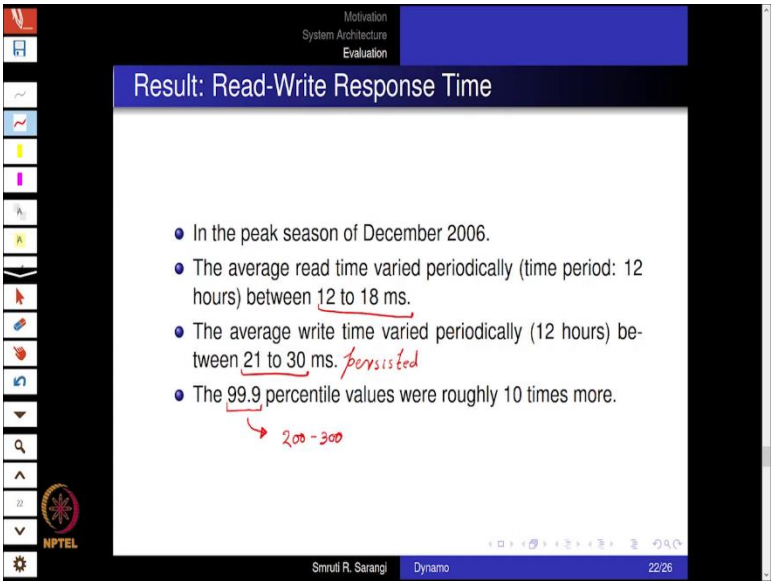
Let us now look at the response time for reads and writes. So, in the peak season of December 2006, the average read time had a diurnal variation. So, the time period a variation was roughly 12 hours. So, what was seen in this 12 hour period is that the read time varied from 12 to 18 milliseconds. So, an astute reader can ask a question that our system was designed to make writes fast, but how come the average write time in the same period is 21 to 30 milliseconds, but the average read time is just 12 to 18 milliseconds.

Well, the answer lies in the fact that the writes need to be persisted to durable storage such as the disk in at least one server, in at least one of the replicas in the preference list, the writes need to be persisted. Persisted means written to stable durable storage. And this takes time. In comparison, the read is much faster, because most of the data for a read can come into the cache. But the write needs some durability guarantees, because at least the A, I and D properties hold in our system. So, that is why writes were still slow, but not that much slower, because we do so much to speed up writes.

So, these were the mean values, the 50 percentile values, but the 99.9 percentile values were roughly 10 times more. So, this much of inherent variation was there in the system. Nevertheless, 100 milliseconds or 200 milliseconds is not much. And the user is willing to wait for seconds to get the answer. So, this is why the system works and it is reasonably fast and responsive.

(Refer Slide Time: 54:23)





So, another thing that was compared was buffered writes as compared to direct Berkeley database writes. So, buffered writes are writes where we write to the memory. So, recall that the numbers are 21 to 30 milliseconds, and 99.9 percentile values were 10 times more, which means roughly in the range of 200 to 300 milliseconds.

But if we were to consider in-memory writes, where we just write it to memory and gradually persisted to disk, and so in that case the 99.9 percentile values were far lower between 40 and 60 minutes seconds, which is a reasonable idea if we have some reliability guarantees where we know memory is not going to fail.

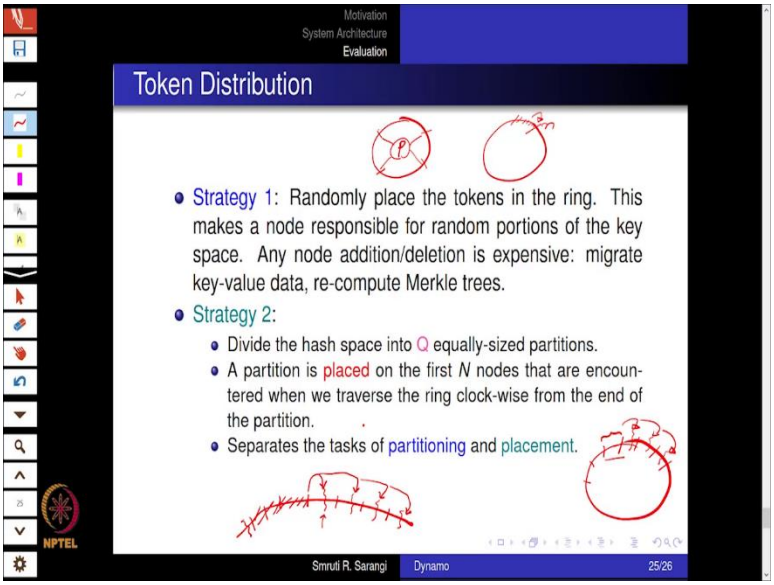
And for Berkeley database writes, were a Berkeley database is a database tuned for small files, so in that case also it was between 40 and 180 milliseconds. So, the write time was nevertheless still high if you are not using a full in-memory solution. But, of course, nowadays the technology has changed. So, those days flash memory was not there. But with flash memory, which is a non volatile storage, these numbers should have become much better.

(Refer Slide Time: 55:50)

So, as we had discussed, there are two kinds of reconciliation methods. One is we can have a business logic based reconciliation. So, in this case, like a shopping cart, well, we know exactly what is to be done. If we have multiple versions of the shopping cart, we merge them or for many other services, we can just have a timestamp based reconciliation, which was there. In, as we had seen in the coda file system, it was there with AFS.

So, in this case, the last write wins, which means that for the same object whoever is the last writer is the one who finally decides the state. So, of course, in this case, there is no merging, so something like a customer session management, where we are managing the session of a customer falls into this category.

(Refer Slide Time: 56:43)



So, the last point that we need to discuss is that in a circular ring, for a physical node there are multiple virtual positions. These are called tokens. So, how do we place the tokens? So, the first approach is that we randomly placed the tokens in the ring. So, then what happens is that if this is the ring, this makes a node responsible for random portions of the key space, which means that if a node is over here, then the key space between this and the previous node, all of this is mapped to this node.

So, any node or addition, of course, in this case, is expensive, because the key value data has to be migrated and we need to re-compute Merkle trees. So, this is the baseline vanilla scheme that we have been discussing, which is pretty much that, we just place tokens at random nodes for the same physical node. And then the regular chord like hashing scheme is used, the regular chord like mapping scheme where every node is mapped to its successor that is used.

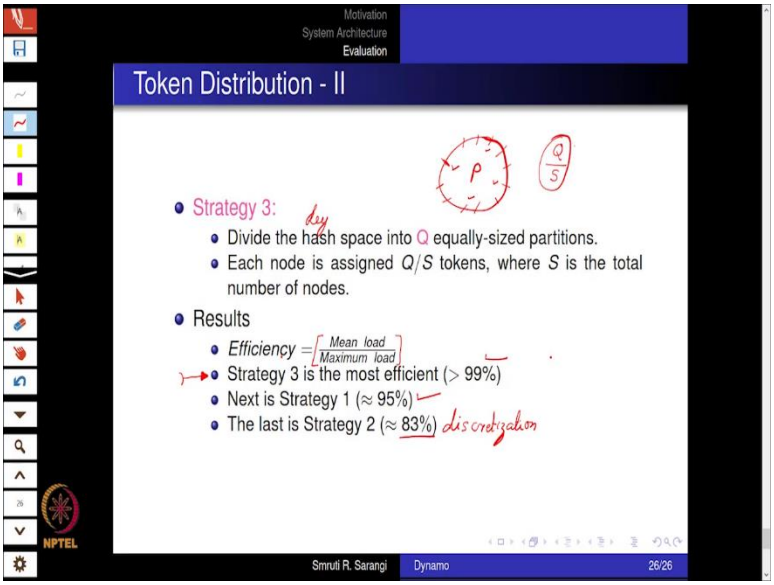
The other approach is something like this. We divide the hash space into Q equally sized partitions. So, then what happens is that, so the hash space is divided into two equally sized partitions. So, partition is placed on the first N nodes, which means that we split the job of partitioning this key space and assigning the keys to successes. So, in this case, if I were to consider this partition of the key space, and let us say if these are the positions of all the nodes, then for a given partition I will assign it to N successive nodes on the ring.

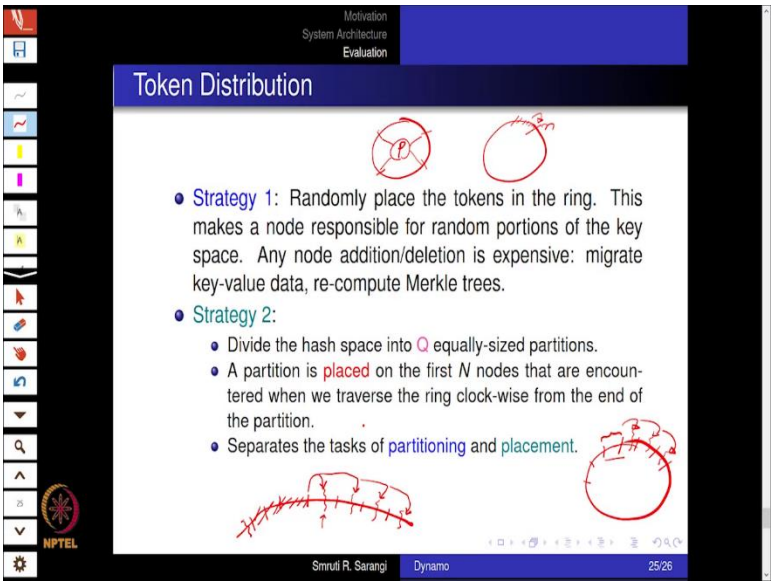
So, basically, if we have this partition over here, any key falling in this region is kept here, also here, also here, these are its reference lists. So, one advantage of separating the tasks of partition and placements, if I were to kind of draw a zoomed in view of the same figure, and let us see if this is my partitioned key space, so if this is how my key space is partitioned and let us say this is where my nodes are, my virtual nodes are, then for this region of the key space, it is assigned to the N successive nodes on the key space.

So, this is a better form of load balancing, where, of course, we have kind of discretized, we have quantized, the key space, and we have also ensured a certain amount of load balancing for each virtual node, which means that it is not the case that in a single virtual node is responsible for a very large portion of the space. So, that has not happened. Of course, there can be sparse regions, but then nevertheless this is still a reasonably effective idea of trying to balance the load out.

But of course the main problem with Strategy 2 as compared to let us say Strategy 1 is basically that it is possible that the distance between two nodes is actually a lot. So, there is some sparsity in the way that the nodes are mapped. In that case, all of these partitions, key space partitions between these will actually get mapped to this node and that is a problem. So, given that that is a problem, this will still cause some load balancing issues. So, that is the reason we have Strategy 3.

(Refer Slide Time: 60:59)





So, in this case, what we do is that we divide the hash space and the key space is the same, we divide it into Q equally sized partitions. So, then, so this is something that we were doing in Strategy 2 as well. So, here S is the total number of nodes. So, but in this case, our load balancing is explicit. So, this is a slow scheme, but this gives the best runtime performance. So, in this case, if there are Q tokens and S nodes, the average number of tokens per node is Q/S.

So, what we essentially do is that whenever a physical node joins, we assign it random tokens. So, that we do. So, let us say, this be one position, another position, another position, another position. And essentially, we steal tokens from other nodes such that this condition holds. On an average roughly this condition holds, which means that Q/S tokens are held by each node. So, this is a very, very explicit form of load balancing, where all the tokens are the same size. And the number of tokens held by each node is roughly exactly equal.

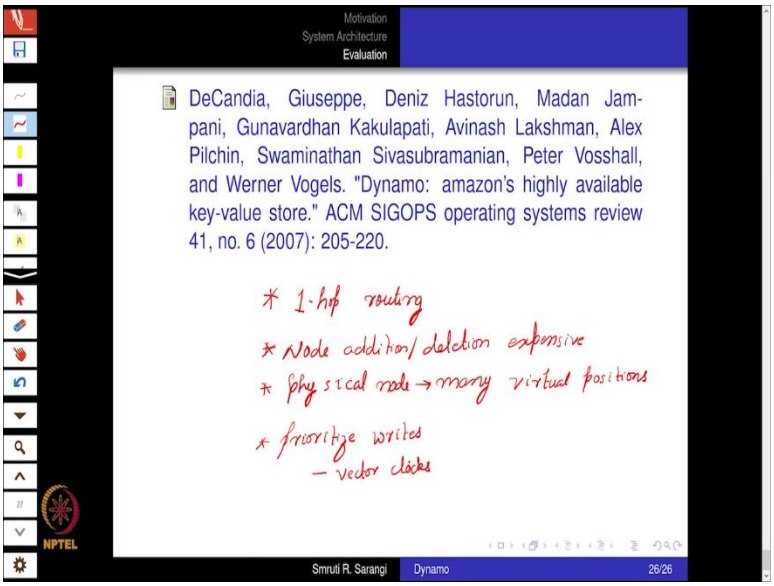
So, Strategy 3 is clearly expensive. But this is appears to be the best we will see when we look at the results. Strategy 2 is something that takes us to its strategy 3. And Strategy 1 is, of course, the random default scheme. So, if I were to define the efficiency as the mean load per, mean load is the mean number of keys stored per virtual node divided by the maximum, so Strategy 3 would be the most efficient with an efficiency of 99%.

The next would be Strategy 1, which is again purely random, which is 95%. And the last is Strategy 2 with 83%, because here we did discretize the key space. So, Strategy 2 was the one that introduced discretization in an aim of doing load balancing, but it did not really take care of sparsity

very well, which the random schemes actually did. So, in this case, essentially, Strategy 3, which does an explicit load balancing, is the best.

So, of course, Strategy 3 is also rather expensive when it comes to node addition and deletion. Because when we add a node, we have to steal tokens from other nodes and give it to the new one. And while deleting we have to follow the reverse process of taking out virtual nodes, taking out tokens and distributing them to other nodes. So, this is, of course, expensive. Strategy 1 is not that expensive and also Strategy 2. But so there is clearly a trade-off between the amount of effort we put in and the quality of the results.

(Refer Slide Time: 64:08)



So, this was the original paper Dynamo: Amazon's highly available key value store in operating systems review 2007. So, I would request the viewers to take a look at recent avatars of dynamo and read some of the recent papers from Amazon to get an idea of the improvements that have been done to the system.

So, one thing that we will appreciate in this course is that subsequent key value stores that were used by Facebook and LinkedIn and so on, have a design that follows a very similar philosophy, where essentially the idea is to create a large key value store for storing a lot of data.

So, if I were to summarize, the main points of this of course was 1-Hop routing, which meant we had to keep massive routing tables and this also made node addition and deletion expensive. And

the other was that, so basically every physical node was assigned many virtual positions, which of course as we saw were random, but kind of intelligent random where load balancing was explicitly taken care of. And the other thing is we did prioritize writes. And one way of doing this was by using vector clocks to maintain different versions and merge them, when, either automatically or using a business logic based merging where the client does the merge.