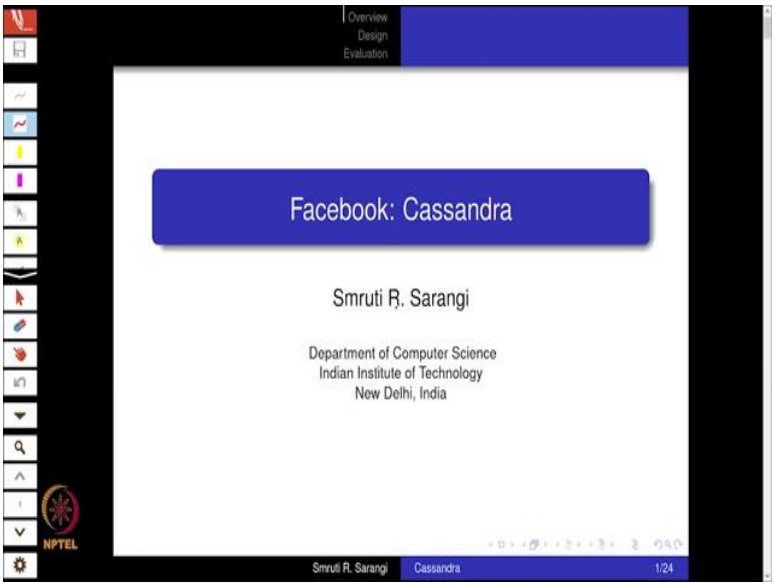
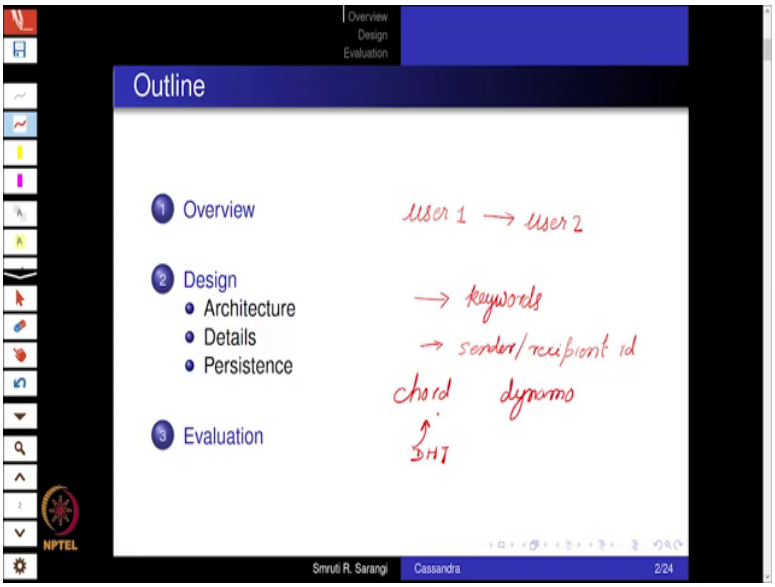
Advanced Distributed Systems
Professor Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture 21
Cassandra: Distributed Storage Engine

(Refer Slide Time: 0:17)



In this lecture, we will discuss the Facebook Cassandra system. The Cassandra system is a key component of Facebook that is used in the inbox search. So, we will discuss, what is inbox search? Over the next few slides.

(Refer Slide Time: 0:35)

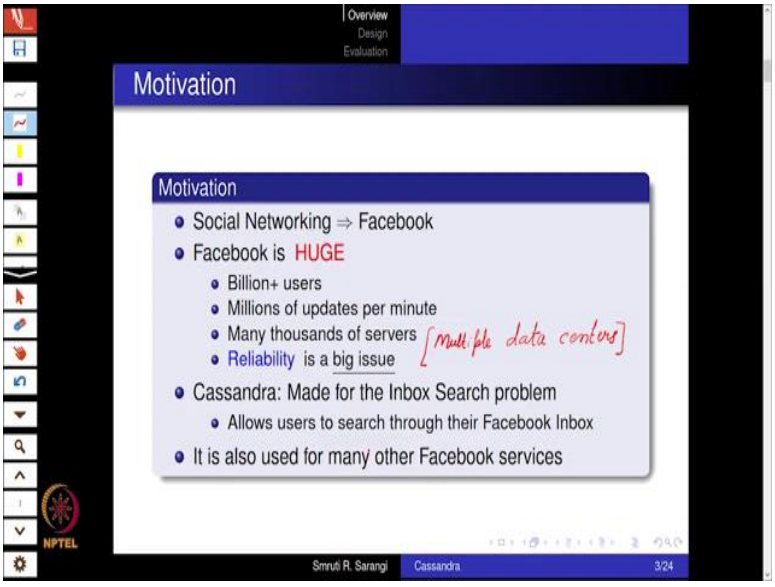


Of course, this does not need any introduction, because almost all of us are very active Facebook users. So, all of us know what an inbox means. Facebook has a messaging system between users. So, it is possible for a user 1 to send a message to user 2 that appears on user 2's dashboard.

So, active users can also get hundreds of messages per day. And it is the job of Facebook to store all of this data in a database of sorts. And furthermore, it should be possible to query these messages based on keywords. And based on the recipient, and basically based on either who the message is being sent to, or who the message is being, or either the id of the sender or the id of the recipient, let us call it the sender or the recipient id.

So, actually maintaining an inbox search for a billion plus users is a rather difficult challenge. So, that is why the Cassandra system was developed. To solve this particular problem, we can think of it as a logical extension of the chord and Dynamo projects. So, that is why it is important to have at least an idea of chord, the chord DHT before proceeding through this lecture. And also, if the viewers have an idea of the Amazon Dynamo project, it would help. So, Amazon Dynamo is described in one of the previous lectures of this lecture series.

(Refer Slide Time: 02:33)

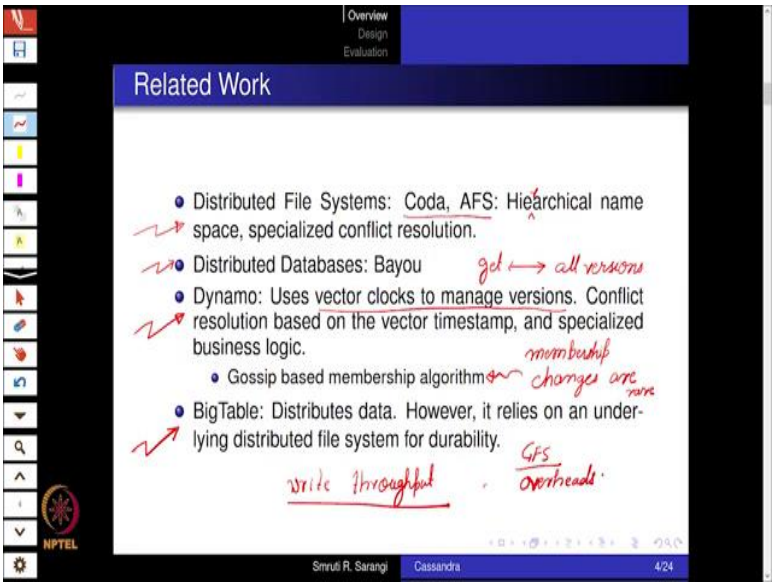


So, nowadays, of course, social networking means implies Facebook, Facebook is the de facto social networking site. There used to be others like Google Plus and orkut, but they are not there anymore. Facebook, of course, is huge. It has billion plus users, millions of updates per minute. In this case, millions of messages are sent between users every minute, users post photographs, users write posts.

So, that is the reason internally, similar to other large systems that we have seen. Facebook will have 1000s of servers that are physically partitioned across multiple data centers. And we will need to say, in such a large distributed system, reliability is a big issue, massive issue.

So, Cassandra, was made for the inbox search problem. It allows users to search through their Facebook inbox. And it has also internally been used to operate many other Facebook services. So, as we have been kind of seeing with Google percolator, and Amazon Dynamo that many of these systems are designed to be generic. So, I said at least the company that is designing them, it can use this software for other things as well to implement other services as well.

(Refer Slide Time: 04:13)



So, a bit of related work, we have already looked at distributed file systems, we have looked at Coda and AFS. So, a distributed file system can definitely be used to implement an inbox, where pretty much every message can be written to a file, or a set of messages can return to a file. And then, so, even though this does appear kind of appealing, the main problem over here is that inbox is not really a file system. And this causes an issue.

Furthermore, file systems have a hierarchical namespace, and they have specialized conflict resolution. So, we do not need all that these things in an inbox search, all these things are not really required. And so that is the reason using a traditional file system to implement an inbox is not really a good idea.

On similar lines using a distributed database like Bayou, that is also not a good idea. Because a database has actually far more structure. So, it is okay to store airline tickets, for example.

But it is not really a good idea to implement an inbox using a database. We have seen 2 other projects we have seen Dynamo, Amazon Dynamo.

So, this is the system that Amazon uses to store its shopping carts. So, in this case, Amazon uses vector clocks to manage the versions. And whenever there is a get, we actually retrieve the entire context, which is pretty much what we get from a get message is actually that we get all the versions and then either the system or the user, they need to perform an automatic merge. The conflict resolution is based on the vector timestamp, specialized business logic. And of course, as we had discussed in Amazon, it will be the same in Facebook also.

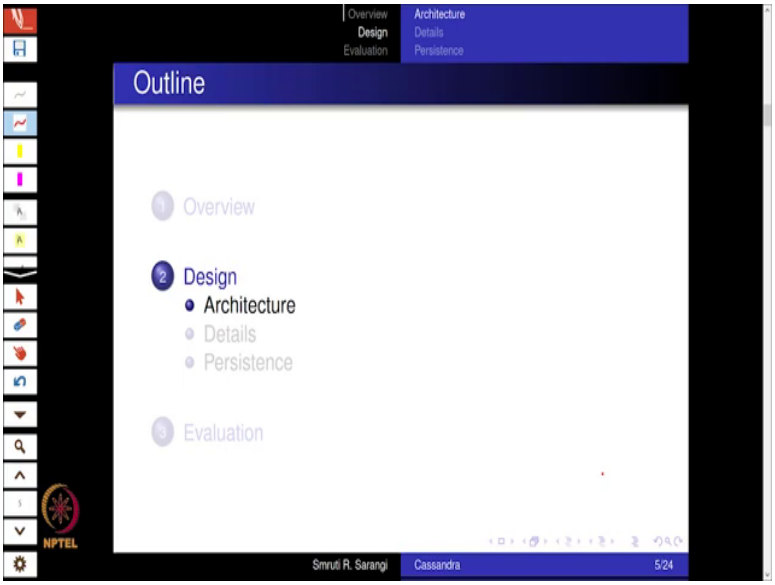
Membership changes to a ring, are kind of infrequent. So, this is the same in Facebook, that whenever so we do not regularly add or remove servers. These are relatively infrequent operations, some membership changes are rare. So, that is a reason even in Dynamo we had a console based system. And even in Cassandra, also, we have a slow console base system.

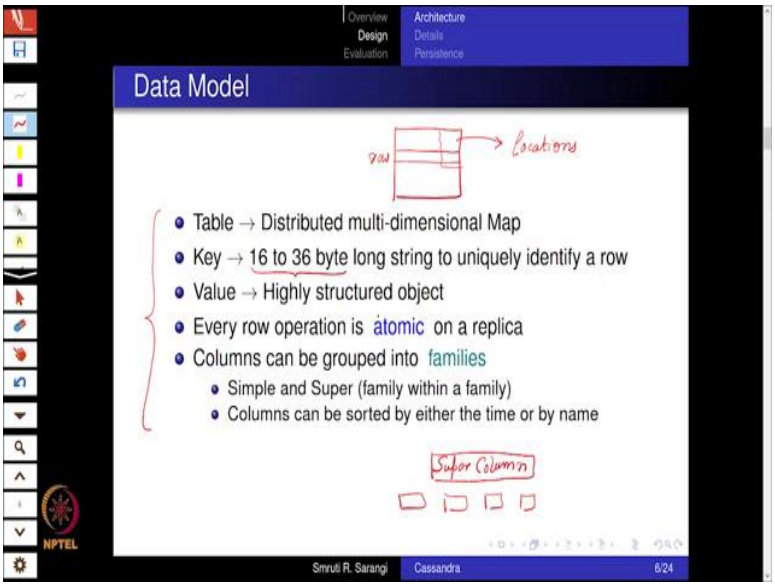
But the main problem with Dynamo is that it is great for shopping carts is great for kind of very structured objects. But inbox is a very simple thing. And it is and for this, what we need actually, is we need a very high write throughput. Of course, latency does not matter that much for light for writes.

But we need a very, very high write throughput. And so basically, that is a prime goal. And Dynamo is not really designed for a very high write throughput, but it is designed for actually very good write latency, and also for aggregating multiple versions of an object. So, it never loses a write. We have also looked at a little bit of Google Big Table in the percolator project. So, big table is pretty pretty much a multi dimensional table where every row is indexed by the key. So, basically, it is a think of it as a key value store in that sense.

So, one problem is that percolator sorry, Bigtable relies on GFS Google File System for the underlying implementation. And the Google File System, what it does is that it divides the large table into chunks, and stores these chunks. So, we do not want structure such a strong relationship with the underlying file system, because that also adds its own overheads. So, that is the reason there is a need to create a bespoke system over here, a custom system over here.

(Refer Slide Time: 08:35)





And of course, Facebook is large enough that it can afford to create a separate, fully designed system from scratch to store in Facebook inbox messages. So, we will define a couple of terms over here that will be used pretty regularly over the next 18 slides or so. So, first, the table, again to Google big table, it says distributed multi dimensional map, which means that it is a very, very large table. So, we can think of let us consider a 2d table to start with. So, every row can actually have a large number of columns. So, in a certain sense this table is partitioned can be row wise can be column wires can be block wise, and they are physically stored at different locations.

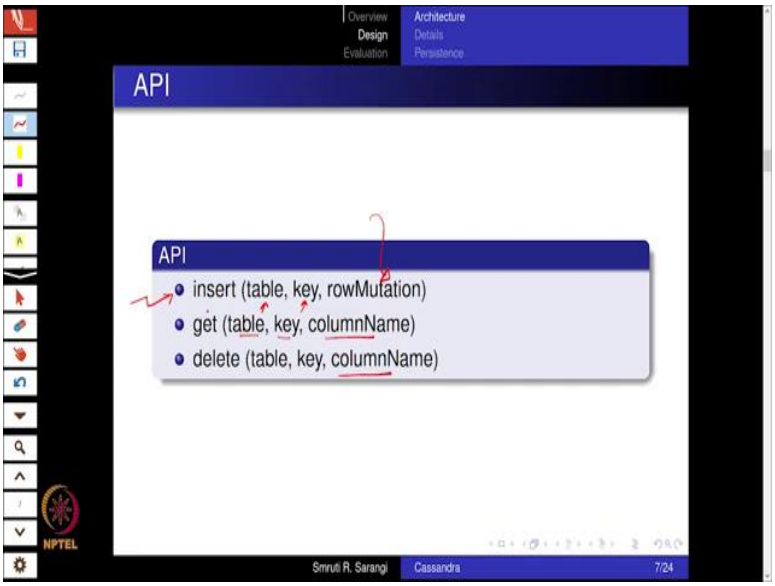
So, the physical storage is at different locations. And so, it is so this table is essentially stored in a distributed fashion. And of course, each slide a key value store for every row has a unique key, which in this case is a 16 to 36-byte long string quantity to identify a row. And a value, of

course, in this case is a highly structured object, we will see how and why. And furthermore, similar to big table, we assume that every row operation is atomic on a given replica.

So, this is important, every new operation is not atomic on like all replicas. But for a given replica, it is atomic. So, a given replica will never have a mixed row state. And for the columns, which we can have many, the columns are grouped into families. So, we can define a super column that essentially contains a lot of child columns.

So, this is possible to do, and a column themselves can either be sorted by time, time of modification time of creation, or by name. So, it is important that the columns are not just one hierarchy, but in the columns have a hierarchy within them. And so, we can group columns into families and we then we can create a family within a family. And so that would pretty much be a super coloumn.

(Refer Slide Time: 11:23)

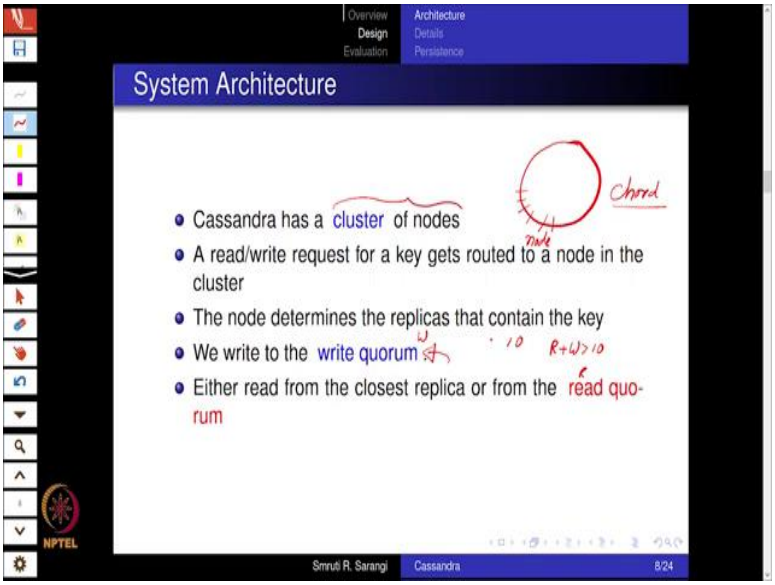


So, the API is a rather simple, the API is pretty much as simple as it gets. So, it is similar to broadly what we have seen in percolator, which is that inserting into the table is rather easy. We give the table id the key which uniquely identifies the row. And in this case, as we have been seeing in distributed systems up till now, anytime we create a row, we actually create a new version of it.

So, the row mutation basically means that we are creating a new version of the row. And get, well it is the same thing, but we get a certain column name. And we can also delete a certain column or we can delete the contents of a certain column. So, insert get and delete is a very

standard way that almost all of these tables operate. That includes percolator dynamo, big table, everything we have seen up till now. And the interface in that sense is rather standardized.

(Refer Slide Time: 12:30)



And one reason for having a standardized interface is that it is useful both for inbox search, as well as Facebook can use it for other applications as well and at some point of time. Well, so some versions of Cassandra are available publicly. But let us say at some point of time, if companies decide to open source such kind of systems, developers can use them to build a wide variety of applications, which can benefit the host company as well, that is also one reason why open sourcing internal projects is typically a good idea.

So, in this case, Cassandra has a cluster of nodes, it is a distributed system, a read write request for a key will get routed to a node in a cluster. So, this cluster is of course, organized as a DHT. And this is the DHT that they use as called. So, that is why I said that chord is a understanding chord knowing chord is a prerequisite for this lecture.

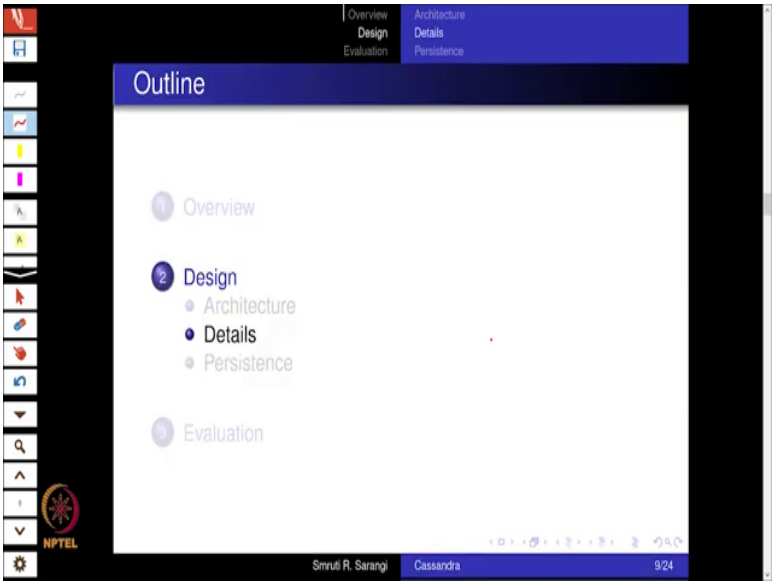
And so similar to chord, what we do is that given a key, we map it to the hash space, then we traverse the hash space clockwise till we find the nearest node. And then of course, we maintain replicas of the key value in the next n nodes subject to the fact that they are in different data centers.

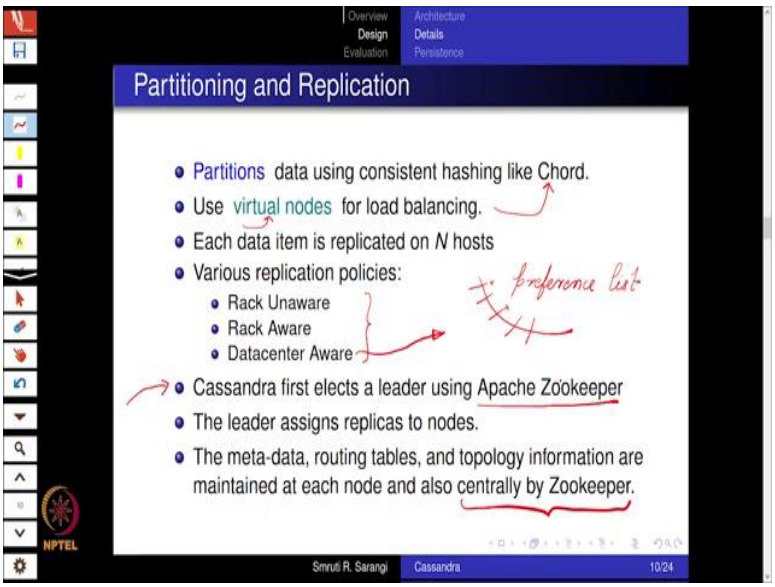
So, then, of course, similar to chord we determine the replicas that contain the key. And among the replicas, we define the write quorum, and then we define the read quorum. So, write quorum, if you recall, let us say that there are 10 replicas, and the size of the quorum is W size

of the read quorum is R, then we had always said that r + w > 10. And which will basically ensure that there is at least an overlap between the read and write quorums. And so, in this case, it will mean that we will not miss an update.

So, for a read either we read from the entire read quorum and somehow merge what we were doing in Dynamo. But of course, Cassandra does not discuss the merging aspect. Or we read from the closest replica, the geographically closest replica, or the one that we find the one that is available.

(Refer Slide Time: 14:59)





The details, so, well, as we discussed, we partition data using a consistent hashing algorithm like chord. And chord, if you would recall was using virtual nodes for load balancing, we do

the same over here. So, this kind of ensures that the load per physical node is kind of balanced. And there were some formulae in code that discussed virtual nodes using some results.

So, I would refer you to this paper. So, in this case, each data item is replicated on N hosts, along the thing, of course, if these are the end hosts, but of course, we will skip a few subject to different to the replication policy. The first is it will be it will be rack unaware. So, in the rack unaware policy, we just take the end successors.

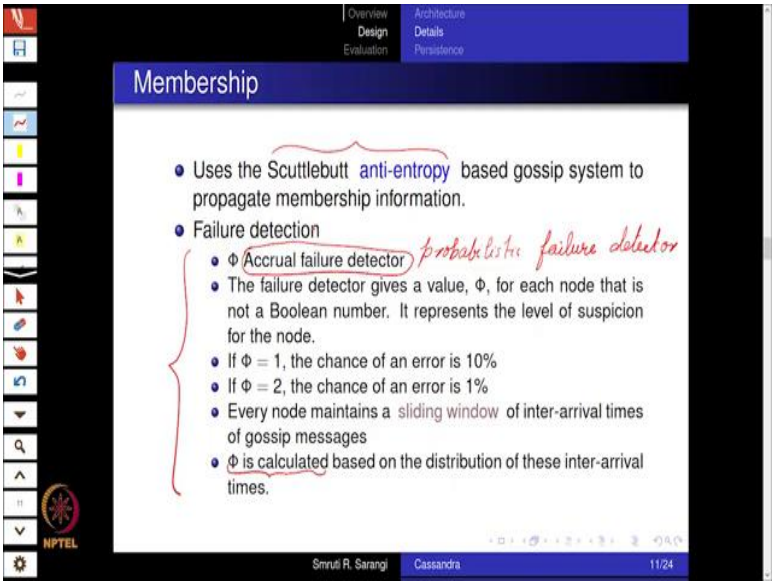
But in rack away, we skip those successors that are in the same rack. So, at least they will be across different racks of the same data center. What is a rack, well a rack is like cabinet of servers. So, it is possible that let us see the network connection to a rack can get cut. In this case, the entire rack will go offline.

So, a rack aware replication policy will ensure that at least the replicas are in different racks. And what Facebook actually does is that it is more of data center aware in this case, you want at least in the ideal case, all n of the replicas to be in n different data centers. If that is not possible, you would at least like to maximize the span of data such that it is in different data centers.

So, just in case an entire data center gets cut off, we can Facebook will still continue to work. Cassandra will still continue to work. So, that is the broad idea over here. So the Cassandra what it does, is among the set of replicas, so similar to Dynamo they also call this the preference list. And so, Cassandra, first elects a leader using the Apache zookeeper service and Apache zookeeper service is a distributed coordination engine.

The job of the distributed coordination engine is to ensure is to provide certain key services, certain basic services. And the basic services include electing a leader and we will see it provides an implementation of a few more distributed algorithms as well. So, you do not elect a leader or do the preference list but you elected leader of all the nodes. So, leader will assign the replicas to the nodes. Along with that the meta data routing tables and topology information are maintained, of course at each node for ought to work, and also centrally by a zookeeper system to initiate repair and maintenance messages of the array. So, party Zookeeper, I would encourage viewers to take a look at this open source project, which has a wide array of distributed coordination algorithms.

(Refer Slide Time: 18:37)



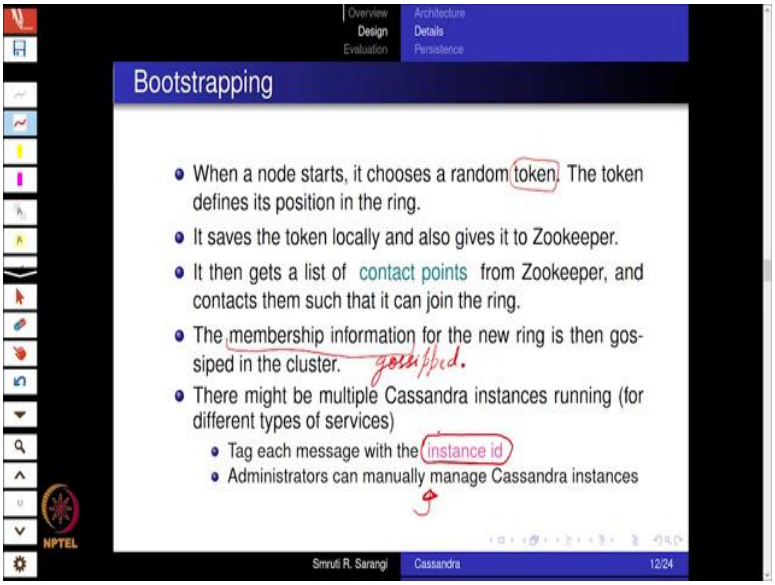
So, for membership well, so membership information is typically gossiped using a gossip protocol. And gossip protocols were discussed in the second or third lecture of this course. So, in this case, Cassandra uses the scuttlebutt anti-entropy-based gossip system to propagate membership information. And this is propagated relatively quickly.

And failure detection is a problem. The reason it is a problem is that we want to detect failures quickly. But to detect failures quickly, we need to actually send a lot of messages. So, what Cassandra does is that it uses a probabilistic failure detector for the accrual failure detector which kind of gives a confidence bound on the failure.

So, it gives a value ϕ for each node that is not a Boolean number, which represents a level of suspicion level of suspicion that the node has actually failed. If $\phi = 1$ the chance of an error in whatever is being predicted is 10 %. Which in this case, if you see it has failed and $\phi = 1$ well the chance of mice being wrong is 10 %.

If $\phi = 2$ it is 1 %. If $\phi = 3$ it is 0.1 percent and so on. It varies exponentially, and on the log scale, every node maintains a sliding window of inter arrival times have gossip messages. And based on this sliding window of inter arrival times of gossip messages, Phi is calculated, and weavers are welcome to take a look at the details of the actual failure detected on the web.

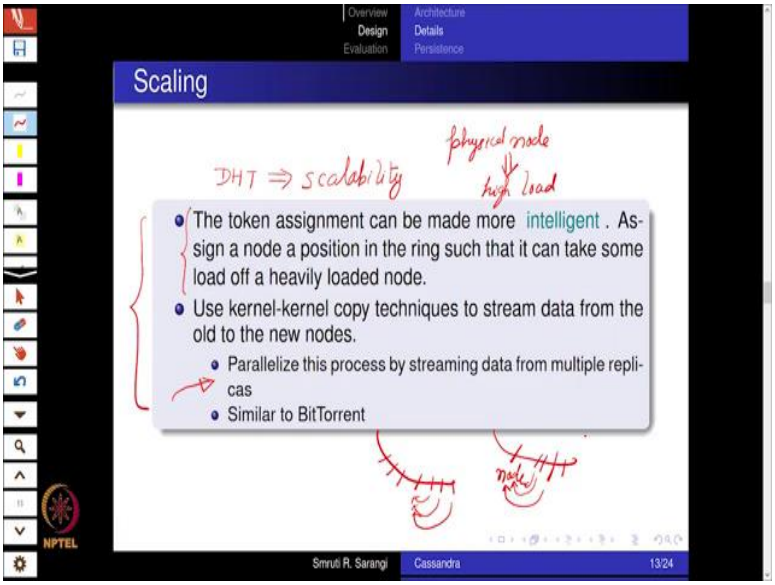
(Refer Slide Time: 20:37)



So, bootstrapping, how does the system starts well, when a node starts, it chooses a random token. So, we have used the same terminology as Dynamo. So, the token is the hash value of its IP address for example, it defines its position in the ring. The token is stored locally and also it is given to zookeeper which is acting as a global coordinator. Zookeeper in return gives it a set of contact points and known set of servers which are already on the ring.

One of the servers while other contexts, servers can be contacted, so that it will help the node join the ring. And now given that a new node has joined the DHT ring, the membership information is gossip. So, of course, there might be multiple Cassandra instances running for different types of services. Cassandra is a generic system. So, you can tag each message with the Instance ID of which Cassandra message it is. And furthermore, administrators are given a console to manually manage the instances and manually manage the process of node addition and node deletion.

(Refer Slide Time: 22:02)



Does the system scale well? Yes, of course. So, using a DHT was the best thing to do in this case, because the DHT automatically implies huge scalability. And of course, we can make so what will stop what is the main problem with DHT? Well, the main problem with a DHT is that it is possible that load balancing maybe non-uniform. So, for a given physical node, we might have a high load.

So, this is kind of taken care of the virtual nodes. This is kind of taken care of by changing the replica placement strategy. Also, the token unlike Dynamo will not assign the physical node to different points in the ring. But the initial token assignment, of course, can be made much more intelligent. In a sense, a node can be assigned a position on the ring such that it is beside a heavily loaded node, such that it can take away some of its load.

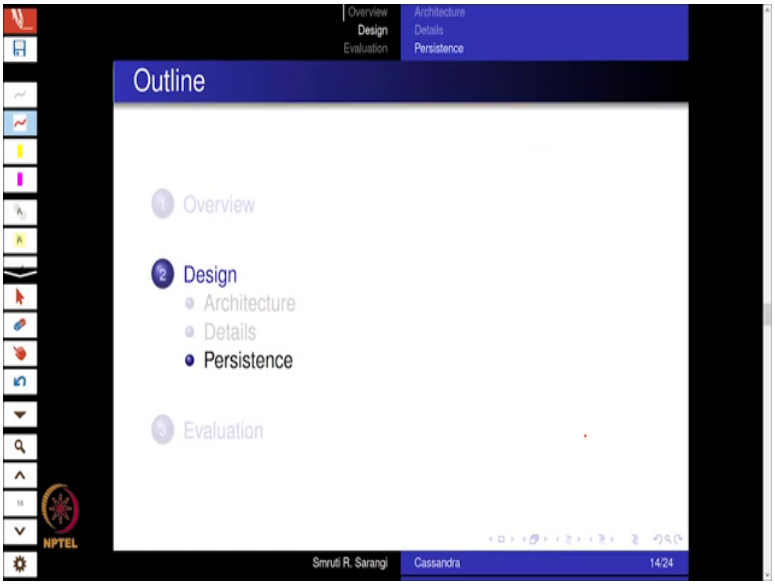
So, different combinations are possible. So, it is a job of Apache zookeeper along with a customized Cassandra engine to ensure that this indeed does happen. Once that has happened, once a node has been placed on the ring, so it will place it needs to get data. So, for all the rest of the nodes, they will be storing, they will be the owner of some key and recall that the owner is a clockwise successor of the key.

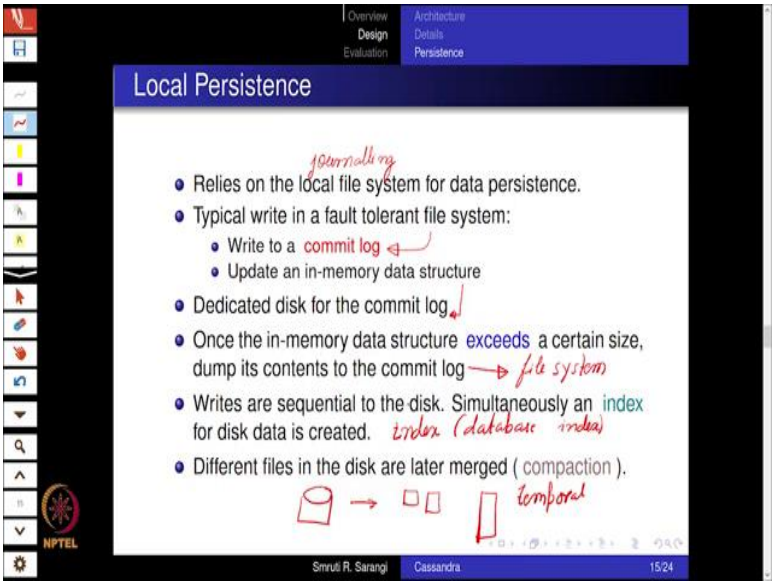
So, it will basically get replicas from a few of the other nodes which are on the ring. So, replica transfer will be a fast kernel kind of transfer, where data is streamed into the new nodes. This process, of course, can be personalized. Similar to BitTorrent in the sense, we can take the replica data, break it into small chunks. And if let us say the node has multiple network ports, then via each network port, we can sort of streaming the data in parallel, similar to what BitTorrent actually does.

So, this is again, kind of like at a lower level, it is kind of lower level in the sense, it is more at the network layer. And so, so it is very started really at the network layer, but it is pretty much at a level that is lower than the DHT. So, in this case, what essentially needs to be done is that if, let us say, there is this node over here that we just added, and another node wants to send a replica to this node.

So, clearly, there are many other nodes that also have the same replica. So, all of them can sort of divide the replica into small chunks and send the chunks to this node in parallel. So, this will speed up the process significantly if the node has multiple network interfaces and can absorb the parallel transfers.

(Refer Slide Time: 25:09)





So, outline a little bit of persistence will persistence basically means storing the results to disk such that if there is a memory crash, the data does not go away. So, this durable storage, so we rely on the local file system for data persistence. So, recall that this is a, this is a journaling file system, I will not describe what a journaling file system is, the expectation is that viewers will go to the web and read about it.

So, a typical and the fault tolerant file system, which uses journaling is that we write to a commit logs instead of directly writing it to the file on the disk, we write it to a commit log. This is used to update an in-memory data structure. Furthermore, it is possible that this commit log will be regularly be persisted to the disk. And we can have a dedicated disk or a dedicated non-volatile memory for the commit log.

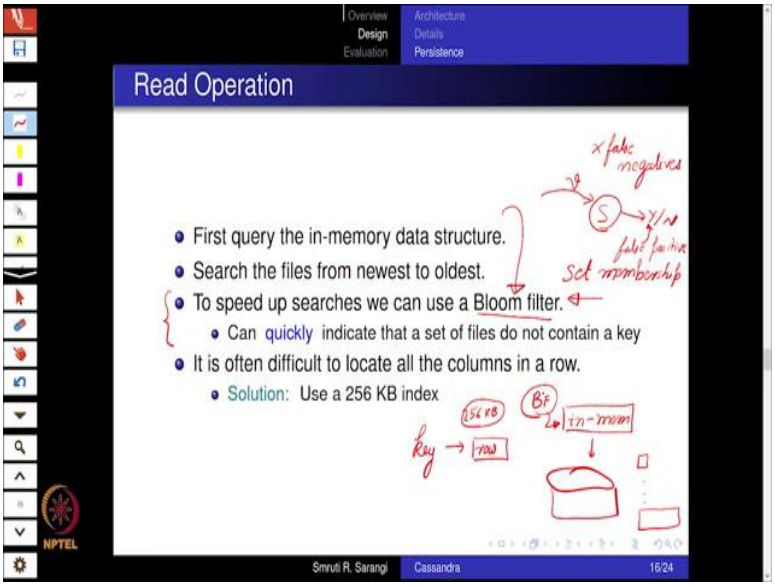
And once this in memory data structure exceeds a certain size, we simply dump the contents to the commit log, which means that from the commit log, we actually change the underlying file system, so the commit log is kind of like a cache in the file system. And it is ordered sequentially.

So, writes are sequentially made to the disk. And, and simultaneously, what we do is that we create or update an index for this data. So, such that if we need some data on a disk, since we are we do not have a regular file system that is organized into directories, we need to create a special index similar to a database index. So, again, a database index, I am not explaining over here, because I am assuming that readers would have viewers will either already know or will kind of look it up.

So, the important point over here is that even if we have lots of data on the disk, it is important to have an index, it is important to index it. And it is important to create a database like index for doing that. And once we have a lot of files on the disk, Cassandra actually has files a variable size. So, the recent files kind of have like small sizes. And gradually, what happens is that older files are much larger sizes.

So, they are kind of merged. So, we do not need them, we create large files, and we kind of keep them. So, this is mainly because inbox messages show a high degree of temporal locality. And because of the high degree of temporal locality, well, most of the time, we will get them in the in-memory structure. If not, we will get them on some small files and the disk and sell them do we have to actually access the large files.

(Refer Slide Time: 28:11)



So, what is what does the read operation look like? We first query the in-memory data structures. We search the files from newest to oldest. And of course, we can speed up this search process by using a Bloom filter. Again, I am not explaining but the Bloom filter, let me say, is this specific kind of a data structure that essentially indicates set membership.

So, given a set of values, and now if it is presented with and this is also listed as a set, it is a set of values. So, now it is presented with a new value, it will essentially indicate a yes or no answer. Yes, means that it is there in the set and no means that it is not there. So, bloom filter. If it says yes, it is there, it does not mean that if it is actually there, it might be there it might not be there.

So, essentially it allows for a false positive. However, if it says that the data item is not there means it is definitely not there, which means that a Bloom filter does not have any false negatives. So, false positives are there but false negatives are not there. And how exactly a Bloom filter works well, again, this is a exercise to the readers if they do not already know. Just go to Wikipedia and read about Bloom filters.

So, the advantage is that the Bloom filter can very quickly indicate that a set of files do not contain a key, which is the key for the row. Alright, so why are we doing all of this? Well, so the reason why we are doing all of this is given a key, we want to fetch the contents of the row.

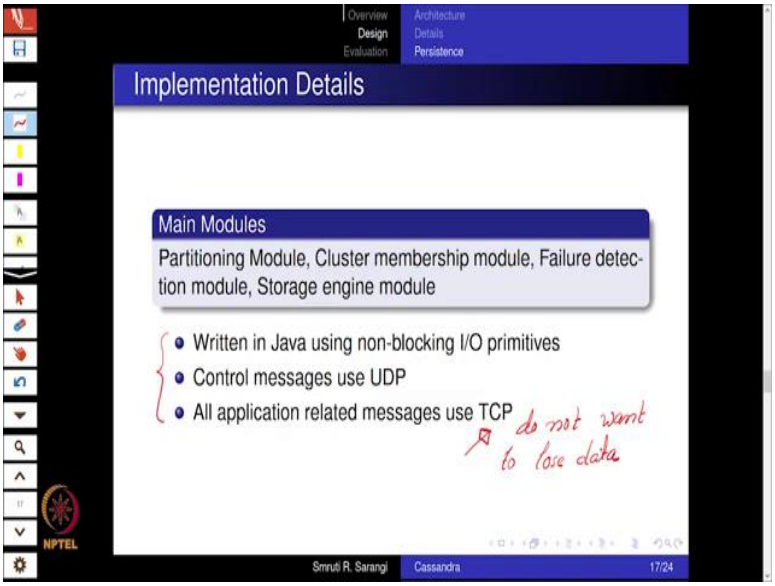
So, this is required for both a get operation and an update operation. And so, for either, you know, it is like get and put in a dynamo, either for getting data or updating data. And so that is

the reason we first access the in-memory data store, and if that is not there, then we access the disk. And the disk, of course, has a range of files of different sizes.

The recent entries are stored in smaller files, and later entries are stored in very large files, hoping that they will, we will have temporal locality. And we will typically not access later entries, and just in case we are users will be willing to tolerate the additional latency. And to speed up the search, well, how do we organize it, we have to create some kind of an index.

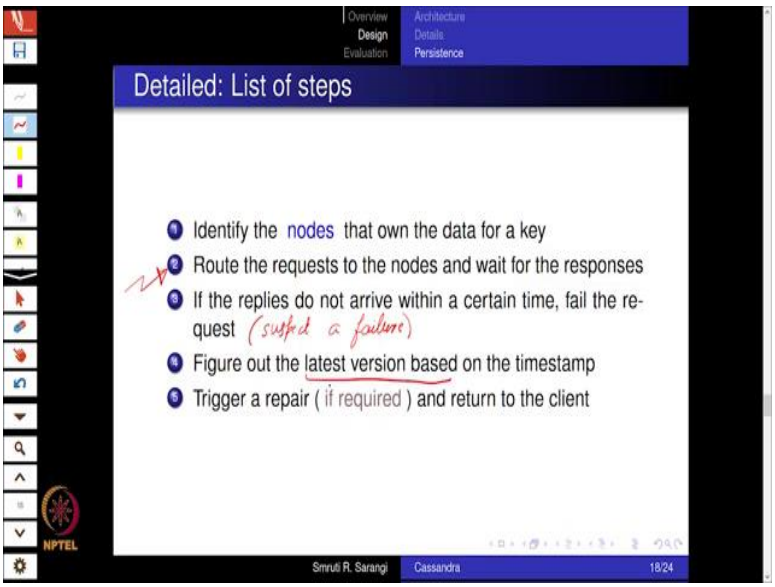
So, we can go through a Bloom filter, which will tell you quickly whether it even makes sense to search here or not. If it is not there, we directly go to the disk. And furthermore, a row itself can have a lot of columns. So, what we do is we use a 256-kilobyte column index in the row, which allows us to exactly find out the byte offset of a given column.

(Refer Slide Time: 31:34)



The implementation details well, we need to look, we need to consider several things we need to consider the partitioning module for partitioning the keys and the replicas a clustered membership how do we take care of membership failure detection and storage. So, the entire system has been written in Java using non-blocking I, O primitives. The control messages to manage Cassandra use UDP and all the application related messages use TCP Of course, this is required. Because this is where we need reliable communication, we do not want to lose data.

(Refer Slide Time: 32:22)

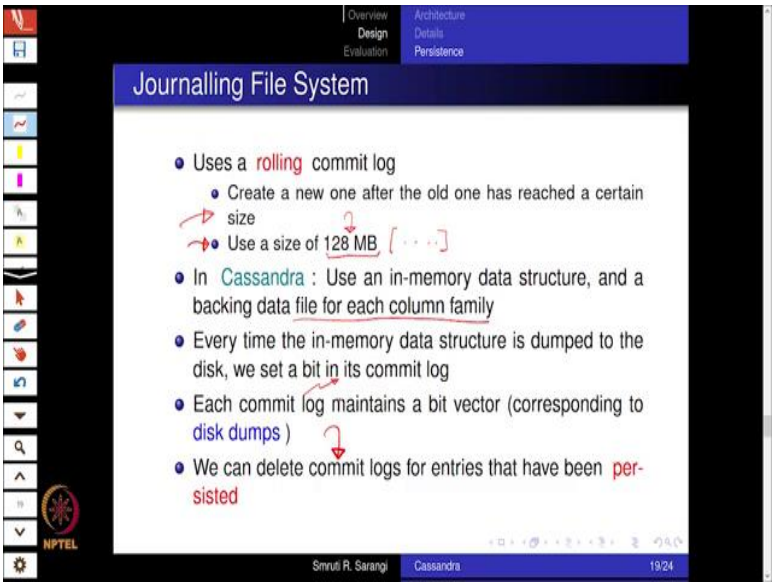


So, we identify the nodes that own the data for a key, we route the request to the nodes and wait for the responses. If of course the replies do not arrive within a certain time the request

fails. And then of course, we tend to suspect a failure. So, the failure approval mechanism that we use will pretty much take such inputs, but of course, it uses far more inputs also like the inter arrival time of WhatsApp messages and so on.

However, if we do get a request, we find out a latest version based on the timestamp and just in case the repair is needed or emerge as needed, well, then we again, return the values to the client. But of course, the Cassandra paper as such, does not broach this issue in great detail. Hence, I will also not try to speculate on how exactly the versioning is done in Cassandra. Because this aspect is not really well covered in the paper.

(Refer Slide Time: 33:37)

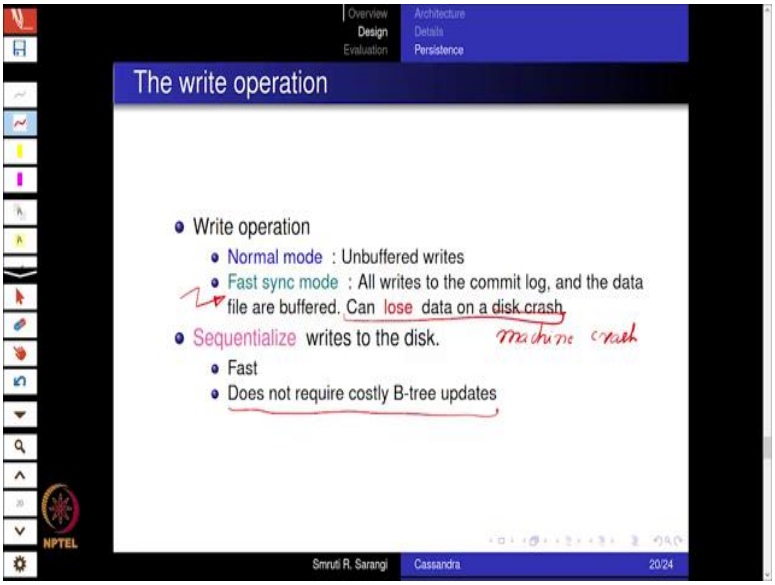


So, it uses a rolling commit log. Well what does that mean? I mean, it is essentially a large log where data can be added and removed at any point in time. So, once a given commit log has reached a certain size, we create a new one and the old one is persisted to disk and the size threshold that is used is 128 megabytes. So, this recall that is an old paper roughly 2010 so, those days even 128 megabytes because you know we would like ideally like to keep the commit log in memory. And those days 128 megabytes was quite a bit of memory for an off the shelf processor.

So, it was 128 MB was not quite a bit of memory, but of course, recall that we are many virtual nodes and then we are storing the return for each virtual node. So, and also the additional overhead. So, keeping all of that in mind, 128 MB was the maximum that designers thought they could afford. So, I stand corrected in my previous statement.

So, in Cassandra, we use an in-memory data structure that we have seen and a backing data file for each column family. Every time the in-memory data structure is dumped to the disk, we set a bit in its commit log. And this basically says that the data has been persisted. And each commit log will maintain a bit vector corresponding to the disk dumps that have happened, which tells you what has been persisted, and what has not been persisted. And for entries that have been persisted, we can kind of delete the logs. And we do not need them. So, that will free up space for us.

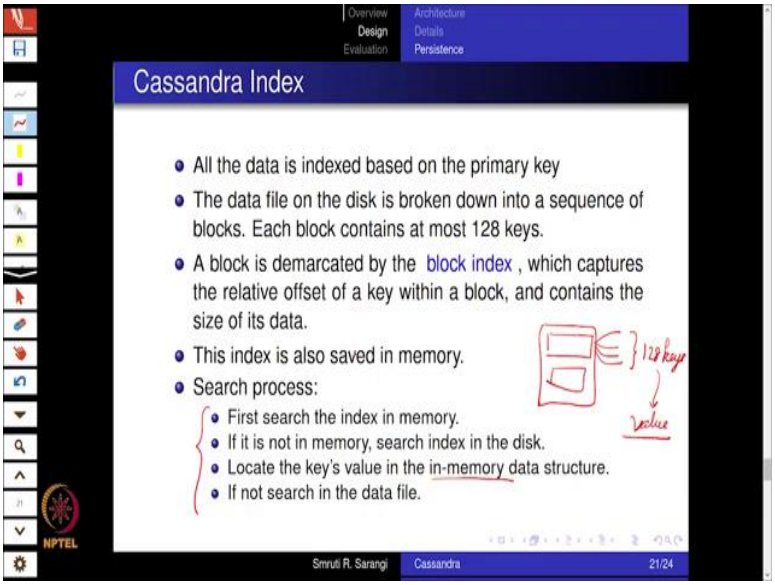
(Refer Slide Time: 35:32)



So, write operations are again of 2 types, we can have a normal, write which is unbuffered, which means that we directly persist and or we can have a fast sync mode. In fast sync modes, all the rights to the commit log and the data file are essentially buffered in memory. So, this makes the rights kind of fast, but just in case there is a crash, we will lose data. And so that is a problem.

So, we sequentialized writes to the disk, of course, which is a fast, fast operation. And unlike databases, in this case, our data structures are very simple. They are essentially like linked lists. So, we do not have costly B-tree updates. I need to make a small correction here. This is actually not a disk crash, it as a machine crash. And the machine crash is something that wipes off the state of main memory. So, that is the reason. If the machine has crashed, its main memory is gone. So, but the data on the disk still remains. And even if he ca not get the machine up, you can always move the disk from one machine to the other.

(Refer Slide Time: 36:51)

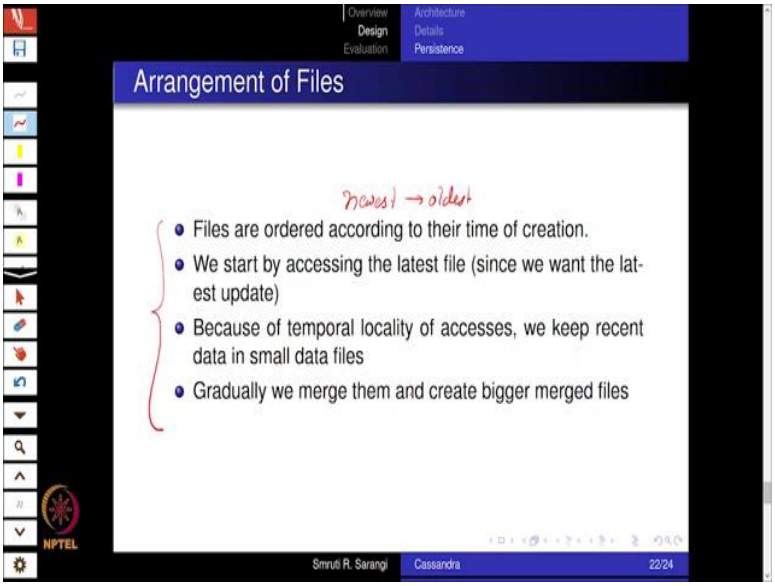


So, let us now look at the structure of a Cassandra index. So, all the data, of course is indexed based on the primary key. So, the primary key is the index of the row. And within the row, as I said, we have columns, families of columns, super columns. So, the data file on the disk is broken down into a sequence of blocks. So, basically, what we do is if you consider the data file on the disk, we break it down into a sequence of blocks, and each block has 128 keys.

So, a block, of course, is demarcated by the block index, which captures the relative offset of a key within a block and contains the size of its data. So, basically, what we see over here is that the search process would look like this that we search, first search the index and memory. If it is not in memory, we search the index on the disk.

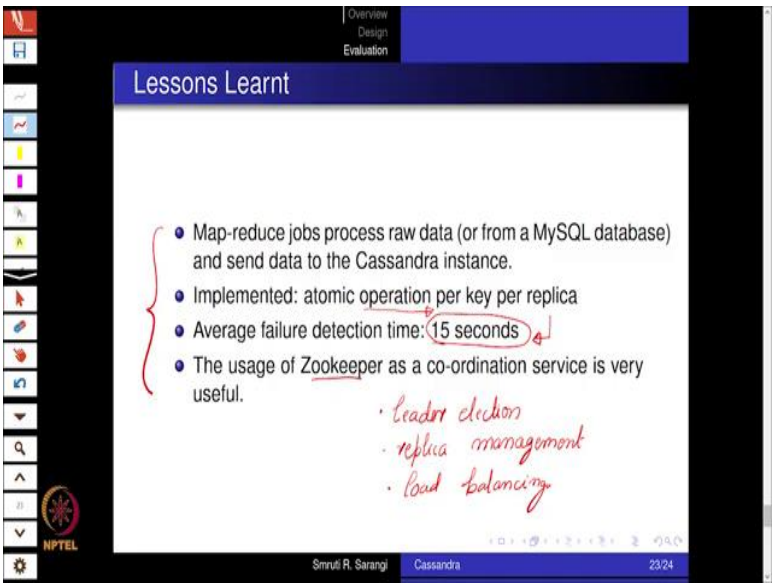
So, given a key, we can quickly locate it in within the index. And using this structure, of course, here we can find the block that contains the key. And from there, we locate the value of the key so, every key will have a pointer to its value. So, value is the location where the data for all of its columns is stored. So, again, the value we search for the value in memory data structure, if we do not find it, we then we search for the value in that data file. So, of course, most indices are stored in memory for quick access. Because in this case, we would like to minimize disk accesses as much and as far as possible.

(Refer Slide Time: 38:38)



The arrangement of files of course, files are ordered according to the time of creation. So, I start with traverse the file from the newest files from the newest to the oldest. So, since you are on the latest update, we access the latest file and because of temporal locality of accesses, we as I said we keep recent data in small files. And gradually we keep on merging the older files to create bigger and bigger files, which of course take more time to access. But we are also assuming that accesses to such files will be kind of infrequent.

(Refer Slide Time: 39:18)

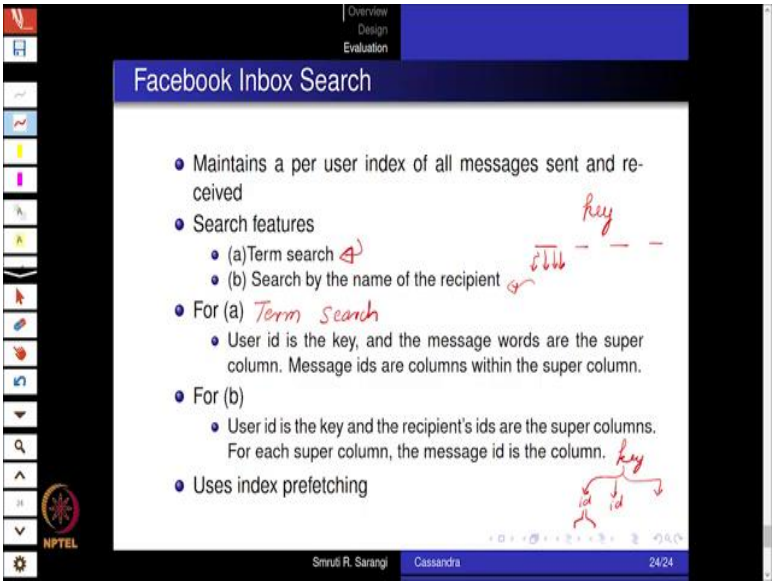


So, what are the lessons learned by the developers in this. So, of course, they use Map-reduce jobs to process all the raw data or from a MySQL database and send it to the Cassandra

instance. They did implement atomic operations per key per replica that was not hard to implement. And the average failure detection time was actually high, it was order of several minutes with a regular failure detector.

But with the accrual-based failure detection mechanism, they were able to bring it down to 15 seconds. And the usage of zookeeper as a coordination service was very useful. So, in general, in a distributed system, at least for the leader election, replica management and load balancing, so let us write it down. You will need some kind of a distributed coordination service.

(Refer Slide Time: 40:47)



As the inbox search, so we maintain a per user index messages sent and received. So, we can search it as we had said in the first slide based on the terms and based on the name of the recipient. If we are to if you are doing term search, term based search. In this case, the user id is the key. And the message words are the super column.

The different message was a different message words are the super columns. And for each and then the message id is that contain that term are the columns. Similarly, if let us say I were to search for the recipient, well, the key would be the root kind of the identifier. The recipients ids are the super columns.

And for each super column, the message ids are the columns. And to speed up the execution, we use index prefetching, which means that we also prefetch the indexes and keep them in memory for very, very fast access, because we need to locate the key and given the key we need to quickly fetch its value.

(Refer Slide Time: 42:03)

So, this was a 2010 paper published by 2 Facebook engineers. And so the Cassandra system, henceforth has been used very heavily. It is very widely documented, as well, in different papers and in the technical press. So, I would ask the viewers to look at other sources of information and documentation on Cassandra.