Week 7 - Video 06

Friday, 5 January 2024 5:40 PM

Hello friends, welcome again. So let's get started with the demo or practicals for cache, right? This will give you more clarity, right? Cache practicals. This might be more than one session, so that's why I have given this extra S, right? So let's start. Till now we have seen theory of it and all fine, but the moment you see practicals, you will be more comfortable, right? Now let me go to my labs and I have the labs here. It is PySpark 3, but as I mentioned, I will go with PySpark 2 because at this moment PySpark 3 version is not stable and the lab providers are saying to upgrade that, right? Because there is a bug in that version and it's an open source, right? So they have shown me the bug as well. By the time probably if you use the labs might be in one week time or two weeks time, it might be a different scenario, right? Okay, PySpark 2. I am selecting the PySpark 2 kernel and let's see. Let me run this particular Spark session. Now here if you see the boilerplate code, you know about it now because we have learned Spark session. We are getting the password, our username and fixing it here. So this is basically if we want to create a manage table in Spark, our data will be stored at this part, slash user, slash my username which is ITB005857 warehouse. And the Spark UI code does not make any sense because of Spark UI we know how we have given us, right? We will be able to access through that. Enable Hive support, master Yarn. We want our application to run on Yarn, so that's fine. Get or create. So this is fine. Now if I have run this, I should be able to see my application here and this is my application. This is my application and I will click on application master to go to my Spark UI. Right? Here it is. Okay, that's great. And now what I will do is I will start writing some code. Let's say I want to read some orders file. Let me just go to my terminal to see what file I should read. Let's say I will say hadoopfs-ls and we have this public folder, right? Where public data sets are there. Inside that, granditech. Right? Inside that, granditech. And then we have inside this orders if you can recall. Let's see what all we have in orders. We have orders 1gb.csv and this was a file which is slightly over 1gb. Slightly over 1gb. LS-H. Right? It shows 1gb but it is slightly more than it so that we get 9 partitions actually. Right? And this shows there are 3 replicas of this data. 3 replicas. So let me take this file which is a decent size file for this cluster and we will be using this. So how do I read this? I will say spark.read. Right? Spark.read and I can say .format. Format of this file, how do I see what it is? Let me just do a head on this. Hadoopfs-head and then inside this I want to just see the initial few lines. Initial few lines. And these are the initial few lines. First thing that you have to understand there is no header. Directly the data is starting up. Right? There is no header line and this is a comma separated CSV file. Okay. So format is CSV. Right? Space and then backslash. And then since I do not have a header I cannot say .infra-schema, header true all of that. So as a good practice we should put in our own schema. So let me define the schema. I will say orders schema equal to and let me say order ID which is of long type. Right? Order date which is let's say date type. So if you see this is timestamp. If I put date it will be casted to a date. Right? That will not be a problem. Let me do that. I could have mentioned timestamp also. That will also work. And apart from that I have customer ID. Any name I can give. Right? And I will say this is of let's say long type and we have order status which is of string type. Right? I will say .schema and I give the schema name. Schema name is what is the schema name? Orders schema. Remember do not give this in double quotes. It does not it has to be directly because you have defined a variable. Right? Okay. Dot and then I will load it. Load it from whatever path. What path is that? I will just see this one. Dot load. First I run this order schema and now I take this as orders tf equal to. Done. And to see if the results are fine I will say orders df dot show. That means I want to see whatever I have done is fine or not. I can see the column names. Let me see the data types also. Orders df dot print schema. Right? Long date long string. And you can see since timestamp was bigger and this is date is smaller so it has casted. Right? It has casted a timestamp into a date because this is feasible. If it could not cast then it will become null. Right? I hope this is there. So till now this is fine and this is the data frame which is slightly over 1 GB. Right? Now let me go to my spark UI and see if something has happened. So basically as soon as the application started we got basically one driver and two executors. By this time you should know what is a driver and what is executor. So one driver and two executors we got. Right? Now that's fine. I will go to this particular job. Right? And you can see. So right now we see only one job and this is totally fine. So I am saying this UI you can keep tracking as and when you run more and more things. That's the spark UI. Ok. This looks perfectly fine. Right now is there anything under storage tab? Basically if you cache something it will show under storage tab. Right? So nothing is showing up. All good. Now let me just say orders df.count to count the number of records. To count the number of records. Right? It's running and it took some time. Let me go to the jobs. There will be a job which will be created for count. You can see this. It took around 4 seconds. It took 4 seconds to do that. So let's see if I click on this. So basically we have two stages. Because count how it happens is. Let's say our data set is slightly over 1 GB. Let's say 1.1 GB. Right? Let's take an example. How many blocks are there in HDFS? It will be 9 blocks. It will be 9 blocks in HDFS. Now how count works is. First each machine or each. So basically now we should not use the term machine rather executor. Right? So there are 9 blocks. Definitely there are 9 partitions in your data frame. Right? 9 partitions. And definitely 9 executors will be working. Whether in parallel or one by one. Whatever. But 9 executors will come into play. Right? And each. There will be a task for each of it. Right? So there will be 9 tasks. And in each task. Right? One executor will be taking a partition and calculating or counting the number of records. So let's say the task 1 returned. Like okay. There are let's say 1000 records. Task 2 returned. There are let's say 2000. Count is 2000 for whatever block it was handling. Let's say. Right? Task 3 returned. 1500 like that. Right? So each. Each executor gave the total local count of it. Right? Each executor gave the local count. Right? And then this goes to one final executor so that we get the final count. And that's what has happened. If you see. Initially we have 9 blocks. So 9 executors would be working on it. There are 9 tasks. Right? You can see this. There are 9 tasks. 0, 1, 2, 3, 4, 5, 6, 7, 8. And you can even get to know which executor. Right? Which executor is working. So the executors are even executors. ID is 1, 2. Right? Like that. So you see this. Host is this. And on each host there can be multiple executors actually. It's not that on one host machine or on one worker machine there will be one executor. These are like containers. Right? So basically if you see there were 9 partitions to your data frame. Right? 9 executors have worked. But you see it's not that all in parallel. This executor worked and then again the same executor worked on other partition and so on. I hope this is clear. So basically if you see your matrix aggregated by executor only 2 different executors have come into play. So this 9 tasks were given to these 2 executors only. So it's like one executor will take one task and then complete it and then take another task like that. That's where if you see this particular executor ID 1 has spent 1 second for one of the tasks. Has spent again 0.8 seconds for another task. So 1.8 total. 0.8 for another task. So 2.6. Again 0.8 for another task. This becomes 3.4. And then again 0.3. So roughly it has taken around 4 seconds to execute 5 tasks which are allotted. Right? So this is based on the other executors might be busy or the cluster might. Many other people are using the cluster. Right? So it's like not happened too much in parallel. I hope this is clear. Right? And size of data each executor dealt with is 128 MP roughly. Like the size of the block. And how many output records are there? Input records are lot. You see that this particular executor dealt with what? How many? 81,000. So this is 1 million records. Input records were 3 million to this. And the output record is 1. What is that one record? The count of that. Right? The count. The count would be nothing but whatever. 3 million. Same way the second executor would have given the output record. Right? And all of these results would have gone to another executor. I will just go back. This would have gone to other executor which would have given you the final answer. So count is if you see transformation we are definitely one more stage will be. Count is an action where definitely it's not that it will not require shuffle. Shuffle will be required but very minimal. Right? Very minimal because everything is done kind of locally and it gave the final counts. So that's where the exchange is required. But it's exchanging one-one record from each executor. Right? So how many records this executor would have got? It would have got 9 records. Right? So this is the executor. If I click here you should be able to see that. You should be able to see that shuffle read is 9. Right? Shuffle read is 9 records. And then finally it will give one final result as the answer. Now let me try running this again and see what happens. How much time it takes. It's still taking decent amount of time. And it depends. Sometimes it takes less. Sometimes it takes more. Because again the resources you are getting are not guaranteed. Right? Now let me click on this. And this time it took 5 seconds. Slightly more. And if I click here let's see how many different executors we got. This time we got 3 different executors to do our work. Right? If we would have got more executors and they would have done it parallel. It would have been completed much faster. Right? So you see this. This time 5 tasks went to this particular executor with executor ID 1. 3 tasks went to executor ID 4. Both these executors are on the same machine. You can see this. Both these executors 1 and 4 are on the same machine which is worker node 2. The third executor is also on the same machine. So all these 3 containers are there on the same machine and can get parallelism. Right? So 5 tasks, 3 tasks, 1 task. That's fine. Perfect. Let me try running it once again. This is quite interesting. Right? So again I am running. Hoping that I see different results again. Because it's all dynamic. This is all dynamic. Right? This is done. This time it took 4 seconds. And if I click here. Let me see what all executors I have got. Again I got the executors from worker node 2. Right? And we see that 4 tasks went to this, 4 to this and 1 to this. Let me run it one last time. Hoping to see little change. But I do not know what will happen. If we get multiple executors across machines. It will be fast. Now this happened in 2 seconds. And without seeing underneath, I can tell this time we got more distinct executors. So that it's not same executor will finish one task and go for another. Might be we have at the starting itself got more executors. That's the thought process I am having now. But let's confirm. And whatever I thought is true. Right? We got 5 different executors. Right? Executor ID 1 got 3 total tasks. Executor ID 5. Which is the first 3 executors are residing on worker node 2. Right? First 3 are residing on worker node 2. And the other are residing on worker node 3. Right? So we got more parallelism this time. We got more parallelism. I am curious to run it one second. To see if all 3 workers come into play. Right? But it's not guaranteed. This time it's taking even longer. Right? So let's see. Let's see. And this time it took 5 seconds. Might be again 1 or 2 executors repeating. And you can see this. Right? So 3 different executors. That's why time is more. But you got a feel of it. Like how it is. And if I go to the SQL tab. If I go to the SQL tab. That's where you need to actually check. So when you are working with your higher level APIs. You need to go to the SQL tab. And if I click on this count. Right? Whatever this is. I can see. It is reading from the disk. Scan CSV. Scan CSV. That means it is reading from the disk. And then after doing local counting. It is exchanging. So data is capturing the exchange buffer. So there is something called as exchange read and exchange write. So it is putting it to exchange write. And then the next executor will take it. And it will take it from the exchange write buffer. So I hope you understood that how it works. So basically when they are exchanging. They are exchanging. And when they are exchanging. They are exchanging. So basically when there is a shuffle. You will see this exchange. I hope this is clear. Right? Okay. It has gone to the disk. And that's what is taking time. Can we try doing a cache and see if it improves the things. Right? Let me show you. So what I can do now is. I can try caching this. So I will say. I will say. Orders bf cached equal to. Orders bf.cache. I am creating a new data frame. And orders df.cache I am saying. Ideally cache is neither a transformation nor an action. But sometimes if you see on different articles. Some people say it's a transformation. Right? But ideally it's not. It's lazy. But anything that is lazy. Does not mean it's a transformation. Transformations are lazy. But anything that is lazy does not mean it's a transformation. I hope you got this point. Ideally for me. Transformation is the one which transforms one data frame from one form to another. So I will do that. No. Cache is like a utility function for me. But again some articles might say it differently. It's again ambiguous. Right? So I will cache it. But cache is lazy. Nothing would have happened at the moment. If it would have been cached. I would have been seeing something in the storage tab. My storage tab will be blank at this moment. Empty. Now what I will do is. I will say. Orders df cached.show. Orders show. Orders show. I am calling an action. I am calling an action. I am seeing 20 records. Perfect. What are you expecting now? What would have happened? Now you might say. Sumit sir. It's a no brainer. Since you have called an action. And cache was lazy. Now it would have been executed. Now orders df cached would be cached in memory and everything is cached. You are thinking the right way. But spark is even smarter. Let me go to the spark UI and show you. It has cached only one partition. Out of 9 partitions. It has cached only one partition. You might be surprised. Why? Why it has cached one partition out of 9? We had 9. It has cached 1. It says 11% cached. Because 1 out of 9 is 11%. I hope you are enjoying it. If you are not enjoying it. Probably you have not understood it. This is so interesting. Now spark is so smart that spark says. Ok whatever is required I will cache that. And when I say show. When I say show. I am saying show me the first few records. First few records can come only when we cache the first partition. Why do we have to cache all 9 partitions just to show the 20 records. I hope you are getting it. That's why it has cached only the first partition because spark is smart. If I say orders df cached.head.head. Head is again an action. Where it says give me the first record. And I am seeing the first record. It's fine. Now what are you expecting? Anything significant would have happened? No. It will still be the same because I mean if we have to show the first record. That partition is already cached. That partition is already cached. Right? Unfortunately in this spark version 2. Tail command do not work. Ideally if you are using spark 3. You could have said orders df cache.tail and in brackets how many records you want to see. The last records. You can try the same in spark version 3. Because at least this will work. If I am saying cluster is unstable. It does not mean nothing works. It works but does not work the way as expected always. So you try.tail 10. Tail has to take the argument that is number of records. And this will work in your pi spark 3. In spark 2 it will not. New thing. This is a new action. When you run this. You will be able to see one more cached partition. Because it has to cache the last partition. The ninth one. Right? You can try that out. Right? In my case I am not showing you because this is not going to work here. So you understood spark is smart. And it will cache only whatever is required. And that is what we have seen. Now let us see little more. If I click on this. If I click on this. If I click on this. You can see that 33 MB of data is cached. Right? When this particular data or block was actually on the disk. It was around how much? What is the size of block? 128 MB. But here it comes in a slightly compressed manner. And it is 33 MB. Same data. Entire data. Right? But the way it would have kept. That it saved some memory. Right? It would have been compressed. And this is in memory. You can see this. Everything is in memory. It does not have to go to the disk because it can fit this in memory. Right? It can fit this in memory. And the storage level is memory deserialized. Right? I will talk about this. Let me take this. So memory deserialized. That means whatever can fit in memory it will keep in memory. Otherwise it will keep in disk. It means that. Right? What is deserialized? So there are two terms. Serialized and B-serialized. B-serialized. Right? What is serialized versus what is deserialized? Serialized means keeping it in binary format. It is a binary format. Keeping in serialized format means keeping in binary format. I will talk more about it in future sessions. I will have a special session to cover what is serialized versus deserialized and all of that. But at a high level. Serialized means binary format which will give you more compression. Which will be more optimized in terms of space. It will take less space. It will take less space to store. But again thing is that when we keep the results in serialized format there is a slight overhead to change it into deserialized format. So that means in terms of it takes extra CPU cycles. That means for this extra CPU computation is required to change it from serialized to deserialized format. Right? Because it's kept in binary format. To change from binary to normal format it will take slight some time. So it will be slightly computation heavy but in terms of space it is very good. It takes extra CPU cycles for the format conversion. Right? I hope this is clear at a high level. I am not intending to go in detail. Because then we will be deviated from our actual learning of cache to this. I do not want that. But since at this point you should have slight understanding I am giving you that slight understanding. What is deserialized? Keeping it in object form. Right? It's in the other form, the object kind of form just like the data is kept in memory. Right? Keeping it in object form. So this takes slightly more space. Right? But in terms of compute, computation, this is fast. Right? So one thing is that on disk, on disk the data is always kept in serialized format. Serialized form. And in memory the data is kept in deserialized form. Right? So in disk the data is kept in binary form. Right? And in memory the data is kept in deserialized form in the form of objects and all. Right? But when we talk about caching, when we talk about caching, then in case of disk it is always serialized, same way. It is always serialized, there is no other thought about it. Serialized. Serialized. Okay? And when you talk about memory, it can be serialized or it can be deserialized. Deserialized. You can take the spelling, right? Since I am writing it too many times, I am getting confused with the spelling. That's fine. Okay. So again I am saying, ideally memory stores the data in deserialized form, distorts it in serialized form. But when we talk about caching, you have provision that in disk, again serialized only, memory is always serialized or deserialized but deserialized is preferred for memory. Right? Now you can see memory deserialized, that means in object form, which might take little extra space but it will be faster. And 1x replicated means there is one copy of this. In memory we really do not want to have multiple copies. Right? So 1x replicated. And since everything is fitting in memory, it does not have to reach to the disk. That's fine. And what about things, other things you cannot understand. Do not worry that, oh what it is, how will I understand? I will explain that later, whatever is required. And not everything you should know. Whatever you should know, you will be knowing it. Do not worry. Actually you will be in fact knowing more than what is required. Right? That kind of confidence we have to bring. Okay. So storage trap has only one partition cached. Now let me basically go to my notebook and let me invoke account. Count on this. I am invoking account on my cached data frame. And when I am doing account, now it has to deal with all records. All records. Right? So when it deals with all records, that means first time everything will be cached now. Remaining 8 blocks or partitions will also be cached. I will show you. If I click on storage trap, you can see every partition is now cached. One was cached earlier only. Remaining 8 are also cached now. And 100% of this data frame is cached. Not 11%. 100%. And if I click here, you can see that okay, how it is cached, where it is cached, on which worker node it is cached, on which executor it is cached, not just the worker node, but the executor. You can have a look here. And it does not have to fall back to the disk. It is caching it in memory for now. Everything is cached now. And it has, our 1 GB of data set, slightly over 1 GB has taken a little lesser space here, 276 MB. Because it will keep it in different form and all. That's fine. Now, it would have certainly taken more time now. It would have certainly taken more time. I am clicking on jobs. And when I did a count without caching it, it took 4 seconds or 5 seconds, like that. Now it took 16 seconds. You might be wondering, hey, cache is slow. No. What you have to think? Cache was lazy. This time it would have done that extra effort to cache it. It gave you count also, but it took that extra effort to cache it. It took 16 seconds because it has to cache the results. That's where it took that time. If I click here, you see it took 16 seconds and it was distributed across a lot of executors this time. 3, 4, 5, 6, 7, 8. Totally 8 different executors we got. And these executors took good amount of time. Let me see if any executor is repeating. 16, 17, 18, 7. Here we will get to know. Executor ID. 7, 10, 7. So why it took 7, 10. This anyway is 65 milliseconds. 10, 7, 13, 15, 14, 17, 16, 18. So you can see basically multiple executors have worked and it has taken time. If one executor would have done it or would have come into play 2 times, it would have multiplied 10 into 2. It took 20 seconds like that. But since we got a lot of executors, it didn't take that much time which it could have taken otherwise. I hope you have understood. Now it has cached everything and we are at a good stage now. Everything is cached in memory. So now it will be fast. Is our job killed? Looks like the job is killed. So you can see it can happen. That's fine. So what I can do now is I can say spark.stop. Spark.stop. And now I will try running things again. Okay. I am not doing dot show now. And what I will do, I will go to my Spark UI. Okay. My Spark UI is available on, just a second. It would be available on M02 on this, right? Yeah. Okay. I will be able to see the application and I will click on application master. That's fine. Nothing will be cached. Nothing would be there. That's okay. Now what I would do is I will invoke a count. Orders df.count. So that the first job is triggered but nothing is cached because we are starting a new application. And it took 6 seconds. That's perfectly okay. Right. Now I will say cache. Orders df.cached equal to this. Then I will directly invoke a count. It will now take more time. It will now take more time because it has to do extra work of caching it. Cache was lazy. It will do a count also and it has to cache also in the meantime. So this will be time consuming. Right. The job is still running. And you can see it took 17 seconds. Earlier when we did a count without caching it was taking 6 seconds. Now it is taking 17 seconds after caching because caching has not happened. Caching is happening at this point. Right. As it was lazy. Right. If I click on this you can see we would have got different executors and all. I click here. And you can see this. You can see we got 9 different executors. I hope this is clear. How much time each executor is taking and so on. You can go through it. Right. Not local. What does it mean? Locality level, node local. That means it is working on the principle of data locality. Right. That means let's say if there is a machine, worker node. Let's say there is a worker node. Right. Worker node 1. Right. And whatever blocks are capped on HDFS. Right. HDFS is underlying layer. On this let's say block 1 is capped here. Block 1 is capped on HDFS which belongs to worker node 1. That means it is local to this. So worker node 1 will be working on this. Locally. That means it does not have to bring data from another machine and do. Right. Whatever data is capped on that machine's disk which is a part of HDFS it will process that. So that's what it means when you see node local it means it worked on the principle of data locality. Right. I hope this is clear. And that's a very important thing. If it would work on let's say if this executor 1 worked on data which would have been capped on another machine then it would have taken time for shuffling in that. It does not have to do it. Right. So it shows the hostname, executor ID and all of it. That's fine. Storage tab will show me all the 9 cached blocks. Partitions. Right. Now interesting thing is if I run a count one more time. And it will be super fast. First time it took time but it will give us eventual benefits now. If you see the first count I did was done when caching was not done. After that I did .cache and then invoke the count. This time it cached. Right. That's where it took time. But next time it directly executed in 0.2 seconds. 0.2 seconds literally. Right. Why is the difference? Because if you see first time when we see and go to the SQL tab. If I go to the SQL tab. Right. And when I did count first time where it took 6 seconds. It read it from the file. Right. It went to the file. It read all the records. It says scan CSV. Right. It took time. And actually this UI is not very intuitive when we learn Databricks. Right. There UI will give much more details. Here I am just showing you whatever you can understand because this is not that intuitive also. When we learn Databricks it will be much better. Now if I show you the count after caching has been done. Right. Here it will not go and read it from the disk. But it will read it from in-memory tablescan. Which shows that data was available in memory. Right. I hope this is clear. And that's where it is super super fast. Right. So with this you kind of understood a lot of things here. Right. And moreover one more thing I would like to add. As soon as you started. You started with one driver and two executors. And later how can you see more executors coming in? I will talk more about it but to give you a slight idea. Here it has dynamic allocation. Dynamic resource allocation enabled by default. Here in this cluster. Dynamic resource allocation enabled. And the configuration is minimum of two executors. Right. And maximum of 10. That means when our application starts. Of course our application will have one driver. And it will start with two executors to start with. Right. Executors to start with. We will not have any time less than two executors. Because two is set as minimum in the configuration. And maximum is set as 10. So whenever there is more work. We get more executors. And whenever the executors are sitting idle. The executors will be killed after some time. But it will not go below. Right. And I will show you that configuration also. In environment. In environment. You see spark.dynamic allocation enabled. It is true. And it says spark.dynamic allocation max executors 10. And min executors is 10. I hope you can understand. I will explain more about it. Right now my intention is not to explain this. But all these things should not confuse you. That's why I am saying that whatever high level I am explaining is to make you give that confidence. We started with one driver. Two executors. And we were not doing anything. As soon as we did a cache. Then other executors came into play. Right. You can see executor 3, 4, 5, 6, 7, 8, 9. And after some time. If the cluster is idle. Some of the executors will be terminated. Because we are not using that. Configuration is also set in the cluster. That what's the idle time after which the executors will be taken away. But at any point of time you will not have less than two executors based on the cluster configuration which is right now. Right. I hope this is clear. So we have learnt quite a lot. And this was such an insightful thing that we have learnt till now. What have we learnt? Let me go to the starting. We said we had 1.1 GB file. So that there will be 8 blocks in HDFS. Not 8. Sorry 9 blocks. Then I will go to my notebook and show what we have done. We created a spark session. And we defined our own schema. Because that's a good practice. And since our data set was not having header name also. We gave schema DDN style. Where we have column name, comma, data type. And we have mentioned this. And then we enforced this schema while reading the data. And our actual second column was timestamp but I gave it a date. So it has kind of casted it to a date. Right. If it could not cast any data type. For example if you try fitting a string to end. It will show you all nulls. It will not happen. Right. After that I did a count. And I showed that okay there were two stages. The first stage has 9 tasks. Because 9 partitions were there. Right. And then the results from each partition went to another executor where it gives you the final results. It took around 6 seconds. And then we cached it. Right. We say ordersdf cached equal to ordersdf.cache. And when we say .show it will cache the first partition. Because show means we want to see initial few records. Why to cache everything? Whatever has to be shown to a user that only can be cached for now. Right. So it cached first partition only. That's where we saw it showed 1 out of 9 partitions cached. 11% cached. And then even when we do head it still goes down to the first partition. If you are using Spark version 3 you can do a .tail. And you will see last partition is also cached. Two partitions you will have. When you do head first partition. When you do a .tail with some number. The last partition. Then after that when I do a count everything will be cached. Because it has to finally take care of everything. Right. And this will take some time. And more executors we were getting through our dynamic allocation. Probably if I run this might be some executors would have been killed also. Not yet. Okay. So we got more executors. And since it is dynamic resource allocation. So minimum was 2. Maximum was 10. So we got as much executors which was required. But not beyond 10. And this took more time because first time when the results have to be put in memory in some form. In a deserialized form. It takes some time. Right. And that's where it took more time than earlier. For the very first time. And then after that when we executed it again it was super fast. Because now it does not have to go back to the disk 2D. It will directly take it from the memory. One more thing I want to show. The last count when you see is. Right. When you see here and I go here. You can see it will say now process local. It will say process local. Right. Earlier what it was showing before caching. Before the caching has happened what it was showing. Let me show you that. It will show you. Node local. Right. Node local means it is working on process. I mean concept of data locality. That means it is finding it from HDFS of same machine. Right. From disk of same machine. But process local means it is in the memory of that machine. It is in the memory. Right. So I hope you are understanding what I am trying to communicate. And basically serialized means keeping it in binary form. Deserialized means keeping it in the object form. Generally serialized form is for the disk. Deserialized is for memory. But when you are talking about caching. Caching when you talk about in memory we can have serialized or deserialized form. With serialized it takes little less space. But it is little high on computation. Right. Takes more time for computation. Deserialized means it will take little extra space. But the computations will be little faster. So deserialized is preferred for memory. Serialized is for disk. Right. 1x means there is one copy of the cache. Right. All of this. And if memory would have been full it would have spilled to the disk that way. The cache would have gone to the disk. Right. So I think we have covered a lot of ground here. And you understood cache is smart. It will cache only what is required. It will not cache everything. Right. Now if you want to uncache it. Right now if I see the cache it would be there. Everything is cached in memory. Nothing is on the disk. That's fine. If I want to. Let me remove this part at the top. If I want to uncache it. Here there is nothing called as uncache. Rather we have unpersist. Right. And you run this. And of course it shows you the results. But do not worry. You will be seeing that there is no longer content in the storage. It's gone. It is gone. And again if you say orders dfcache.count. Right. And it is showing me the results. Right. Let me see how much time it took. How much time it took. It took one second. Right. It took one second. And if I click here. If I click here. You can see the results are fine. Let me try running it again. So basically why it has worked faster now. You might be wondering why it has worked faster. Because right now we have more executors. It's not just two. It is nine. So the data is spread across. I mean the work has been spread across. Since we have nine executors at this moment. Right. Because when we did cache. It was a heavy operation one time. And we got more executors that time. Because of dynamic allocation. Now still we have not. Those executors are not released. And that's why when we unpersist the data. Cache is not there. Earlier it was taking six seconds. Now it is taking very less time. Because more work is happening in parallel. We have more executors. Earlier two executors were doing the entire work. And each executor was working in serial mode then. Right. But here since we have nine executors. We got more parallel. That's the only reason. This has happened much much faster. I hope you can understand this. Now it is taking time. And I am highly doubting probably. Some of the executors might have been killed. And you can see this. See whatever we think it will happen that way. Right. Because right now I do not have much work. Executors were sitting idle. Many of the executors are killed now. Right. Now if I go to the jobs count. It took five seconds because again it would have boiled down to just few of the executors now. Right. You can see three executors are doing the work. This executor has taken four tasks. So each task it is doing one after the other. There is no parallelism when this particular executor ID 7 is performing these four tasks. No. There is no parallelism. It is just one task. Let's say it takes one second. Then it completes another task. It takes one second. Right. I hope you are getting the thought process. This session has gone very very well and should give you that confidence I feel. Right. And with this let me stop the session and we have covered a lot of ground in the next session. We will continue on our learning. Caching we will learn to a level that no one can teach you. Right. That's my confidence that to that level I will teach you caching and other things in Spark. Because Spark is super important. We will learn it this way. And I am sure you would be enjoying it more than anything else. So with this let me stop the session and let's start in the next session with the continuation for this. Thanks a lot.