

Welcome back, and in this lesson, I want to introduce DynamoDB indexes. Earlier in this topic, I spoke about the scan and query operations inside DynamoDB. I mentioned how scan should always be less preferred and often a last resort. You should only use scan when you absolutely need to interact with non key values in a table, and ideally, you should use query as much as possible. Now, ideally, you would know about the access patterns on your table in advance of creating it. So, given a data set and what I've done is I've created a new table called high\_scores and I have populated it with some sample scores. This is the same data as what is on the diagram on the right of your screen. This might look fairly logical. We've got a user ID and for each user ID, we've got a particular game and game is the sort key. So for a given user ID, we can have a single game score. We've got a date and time and then the high score for that user. So it might appear to be fairly logical that this is the structure that you would need this table to use in order to get access to all of the data using all of the access patterns that you need but that's not always the case. There's a lot of situations we need to design systems that can use different access patterns, and that's where indexes come in handy. Now using this data model, if I wanted to retrieve the high score for a particular user on a particular game, it would be really easy. I could use the get item operation. So I can simulate this by clicking on one of these items, and this would retrieve the high score on Beat Saber for users 0001 and for the item, I'd get the actual high score itself, together with the date and time that that high score was achieved. Now, using this data model, I could also perform a query operation, and I could look for any items where the user ID so the partition key of the table is 0001 and that would retrieve all of the high scores on any game for user ID 0001 and it might appear at first glance, that that's the only type of access patterns that you'll need on this table but there are many more. But what if I wanted to look at other perspectives? What if I wanted to retrieve all of the high scores for a certain game for all the different users? Or what if I wanted to look at scores for a particular user of the platform in the last 24 hours? I don't have the ability to do that with the query operation because, remember, query can only operate on a single partition key value and optionally limit it by the sort key in order to perform those other access patterns. If I wanted to look at all the high scores for a particular game, let's I wanted to retrieve all the high scores for Beat Saber. Well I could use a scan operation, I could filter by the game a tribute and then for value, I could just enter Beat Saber. Now that's a scan operation that would get me the data that I wanted, but it would have to read every item in that table and then filter out the nonrelevant responses, and that would consume a lot of capacity units on that table. Remember, I have to read all of an item in its entirety any time I want to access any attribute on that item and if I perform a scan operation, it consumes the capacity of all of the items that it has to read and if I'm reading an entire table looking for all occurrences of non key values then I could, in theory, consume the capacity that's equal to every item in that table and that's obviously not ideal. It takes time, and it consumes a significant amount of capacity units. So for any access patterns which don't fall into the structure of the main table, that's where I'd look at using indexes.

Now, there are two types of indexes that are available inside DynamoDB. We've got **local secondary indexes or LSIs and local secondary indexes allow us to specify an alternative sort key. So local secondary indexes exist inside a DynamoDB table. They use the same data, but they allow us to present an alternative perspective so I can see the same data as this main table that's on the diagram on the right of your screen but instead of presenting it using game as the sort key, I could swap it around and use date and time as the sort key.** Now, the problem is that local secondary indexes have to be created at the point of creating a table. So you can't create local secondary indexes or LSIs after the fact. They need to be created when you create the table. So I'm going to do that. I'm going to create a new high\_scores table. I'm going to call it high\_scores\_new and I'm going to go ahead and create a new local secondary index inside that new table. So I'm going to hit "Create" going to call it high\_scores\_new. The primary key is going to consist of a partition key, which is user ID so the same as the original table. I'm going to add a sort key and use the same sort key as before. So this is using the same structure as the original high scores table. This time, though, I'm going to untick "Use Default Settings" and I'm going to add a secondary index. So I'm going to click on "Add Index." This is going to be a local secondary index. The partition key for this index is going to stay the same so user ID and then for sort key, I'm going to add a sort key and pick date and time. So if you look at the diagram of the right of your screen, this is going to be the main table. So high\_scores\_new, and I'm going to repopulate it with the same data. But now we're creating this secondary index, I'm going to tick this box to, say create as local secondary index and this is going to create this alternative perspective. It's going to maintain the partition key, but it's going to use this new sort key for this index. I'm going to double check to make sure everything looks good. It does. So, user ID is set as the partition key and date and time is set as the sort key. Now I'm going to call this local secondary index highscoredate. So enter that into the index name and then add index. So that's configured as it needs to be. That's good. I'm going to scroll down and hit "Create Table." Okay, so now this high\_score\_new table is finished provisioning. I need to migrate across the data. Now there are lots of different ways you can migrate data in DynamoDB but because this demonstration is not about data migration and because I've only got the four items, I'm going to keep it simple. So I'm going to go to the high\_scores table, and I'm going to select each of these items in turn, change it to text, click on "DynamoDB JSON" and just copy this into my clipboard. Then I'm going to go back to this new table hit, "Create Item" change it to text click on "DynamoDB JSON" and then I'm just going to replace it with what I've just copied and hit "Save." I'm going to repeat that same process for all of the remaining three items. I'm going to click on that one go to text DynamoDB JSON select it, copy into my clipboard at the new table and follow the same process again, create, text, select the box, paste in, and save. Back to the table select this item, change the text, select JSON, copy, do the same process text, JSON, paste. Now this is obviously not a scalable process, but it is just to get the point across and it's a nice and quick data migration. So that's the last item go to high\_scores\_new and follow the same process to recreate this. So now we've got this high\_score\_new table, which has got the same data, is this original. But this time we've got this local secondary index that I created when I created the table. So what does that allow us to do? What we can do is we can change the scan dropdown to query to perform a query operation but instead of performing it on the table, we can perform it on the index. So now, instead of having a partition key of user ID and a sort key of game now, we've got partition key of user ID and a sort key of date and time so we could perform a query operation to look for a particular date range of high scores. We'd still need to use the same user ID. So because it's using query, we could only ever query a particular user at a time. So, user ID 0001 but we could look for high scores that were obtained only within the last 24 hours by limiting it based on this date and time value, which is now the sort key of this index. So that's a simple example. **LSIs allow you to specify an alternate source key. Now, LSIs will only work on tables which have a composite primary key.** So that's where they use both the partition key and a sort key logically because it adds a new sort key, it can't be used on tables, which don't have sort keys.

Now, some important things that you need to be aware of for the exam **local secondary indexes are part of the table that they relate to, so they share the same RCU and WCU capacity settings for the table. So whether you specify provisioned or on-demand, a local secondary index shares that with the table. With local secondary indexes, you can perform either strongly consistent or eventually consistent reads on that table. So you've got the options to use both, and that's because the local secondary index is using the same data as the table, and so you've got the same strongly or eventual consistency options as you do have on the main table**. Now, at the time of creating this lesson and AWS do have a habit of changing this y**ou can create five local secondary indexes per table so that's an important limit to understand.**