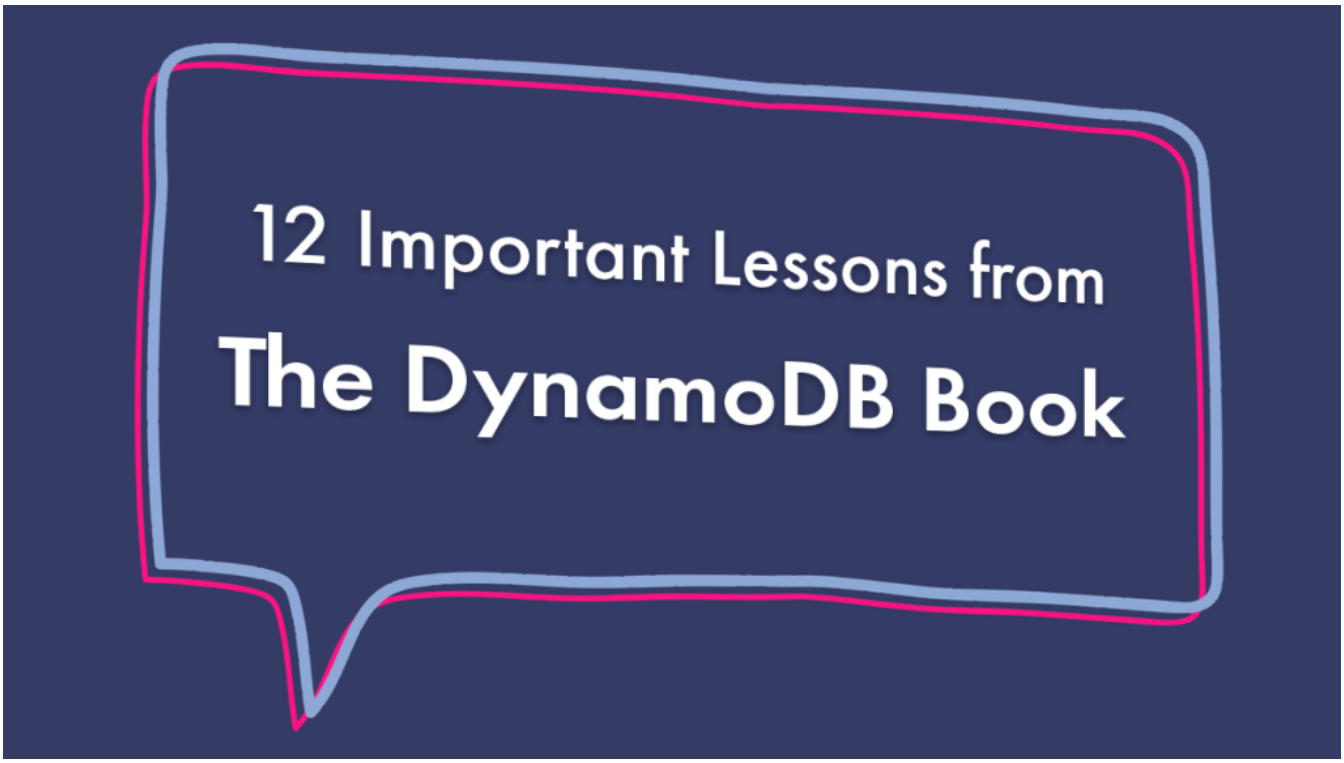
Jeremy Daly
Product Guy, Serverless Advocate & Startup Veteran

12 Important Lessons from The DynamoDB Book

Posted in Databases & Serverless on April 16, 2020



Fellow serverless advocate, and AWS Data Hero, **Alex DeBrie**, recently released The DynamoDB Book, which ventures way beyond the basics of DynamoDB, but still offers an approachable and useful resource for developers of any experience level. I had the opportunity to read the book and then speak with Alex about it on *Serverless Chats*. We discussed several really important lessons from the book that every DynamoDB practitioner needs to know. Here are twelve of my favorites, in no particular order.

1. Think in terms of item collections

It's always been helpful for me to think about partitions like folders on your computer, and items as the files within those folders. Alex thinks of partitions as collections of items and stresses that you really need to think about which collections and subset of items are required by your access patterns. The answer to questions like "How do I join data?" and "How do I represent things like one-to-many relationships?" ultimately comes down to properly organizing items into collections and understanding which groups of items you actually need to have available on the same partition. Alex does an excellent job explaining this in the book.

2. Understand partitions and consistency

While the underlying mechanics of DynamoDB may be abstracted away from you, Alex points out that understanding the basics of partitions and consistency is important to properly modeling your data. This is mainly due to how DynamoDB stores and looks up items, which is dramatically different from your traditional relational database systems. Most importantly, indexes create a **new copy of the data** (not just a reference), which means creating new lookup dimensions can add more replication overhead and additional storage costs. In terms of consistency, it's useful to know that the data is written to multiple nodes (and there's that whole CAP theorem thing). So it's quite possible to access a node that hasn't been replicated to yet. You can avoid this using more expensive strongly consistent reads, though I personally haven't found a lot of use cases for them.

3. Overload your keys and indexes

If you're doing a single table design (which you should be), you need to share your table's partition key and sort key with items that represent multiple entity types. Because of this, your primary keys will often look quite different from one another, but this allows you to satisfy multiple access patterns using just the primary index. This same strategy works for your GSI and LSI attributes as well. This does create quite the eye chart when you view the raw data, but Alex recommends creating an entity chart when working through your modeling. This way you have a document that shows the pattern needed for each entity type, and provides a blueprint for how to construct the primary keys.

4. Separate 'application' attributes from 'indexing' attributes

Alex articulates the benefit of separating out attributes used by your application from attributes used by your indexes. This is something I had previously discussed with Rick Houlihan as being sort of an evolution in DynamoDB modeling thinking. As Alex points out, it is much easier to copy data into separate indexed attributes than it is to use a single, multipurpose attribute. Even if it is the exact same value in some cases, separate attributes give you much more control over the structure of the data used by your indexes. In the past, I've built tables with composite sort keys that would parse out specific values when I'd needed them in my application as well as attempt to use the same structure as part of multiple indexes. While it was possible to make this work, it was terribly inflexible, and there were always so many things that could go wrong.

Using this new strategy, I store each distinct value in its own attribute, and then generate my index attribute values using the appropriate pattern. This allows me to easily parse them out of the table without having to worry about what the partition key pattern is or what the sort key pattern is. Plus, I can easily recreate the index attributes at any time since I have all the data readily available.

5. Add a 'TYPE' attribute to every item

This was something that Alex had actually credited to me, but I'm quite sure I'm not alone in discovering this tactic. This idea occurred to me when I was building the DynamoDB toolbox. I was thinking ahead to single table designs that stored multiple entity types, and was trying to figure out how to correctly parse those when returned from a query. You can have a lot of overlapping attributes, and sometimes you can't tell from just the partition and sort keys exactly what type of entity it is. By adding a "type" attribute to each item, you can easily distinguish between entities, making it simple to do things like unmarshall the data into a more readable format or route it properly in ETL tasks .

6. Learn relational modeling strategies

You can learn all the mechanics, but without understanding the proper strategies for modeling relational data, you can't possibly leverage the full power of DynamoDB. In the past, there hasn't been enough information out there that gives you well-documented strategies to be able to do that. The DynamoDB Best Practices guide on the AWS site offers some really good information, but doesn't go into deep detail. Alex's book does a great job filling in the missing pieces with a very thorough section on these strategies.

Five ways to handle One-to-Many relationships

I highly recommend learning these strategies that Alex outlines in the book. He covers five different One-to-Many modeling strategies including denormalization using a complex attribute, denormalization by duplicating data, using a composite primary key with the Query API action, and using a secondary index with the Query API action. The fifth strategy, using a composite sort key with hierarchical data, is a really, really powerful one. Alex shares a use case involving store locations, using a hierarchy to represent locations – country, state, city, etc. I've used this strategy to create a number of very cool data structures, including ones with a nested item model.

Four ways to handle Many-to-Many relationships

There are also a number of ways to represent Many-to-Many relationships within DynamoDB. Some of these do tend to get a bit more complicated, like adjacency lists and materialized graphs. These are typically hard to visualize when viewing the raw data, which is why using a tool like NoSQL workbench for DynamoDB can make things much easier. Both Alex and I are big fans. It allows you to create your indexes, use facets to define entity types, load up sample data, and then pivot that data so you can see how the relationships work. These patterns can be a little confusing, but Alex offers plenty of guidance along with helpful use cases.

Alex also mentions using normalization and multiple requests to model many-to-many relationships. While it's a valid pattern, it's one of those strategies you hope to avoid because you don't want to be making all those requests. Strategies like this highlight the importance of caching your 'GET' requests. If you are constantly fetching the same data, the more you can rely on a cache, the less you'll have to pay to retrieve it from the database. Having good timeouts or using a write-through cache will make sure that the data doesn't get stale.

7. Use sparse indexes

Sparse indexes are a very powerful tool at your disposal. Alex shares a number of useful patterns, like how to filter by entity types or how to list just users marked as administrators. I love using sparse indexes simply because there are so many use cases. If you're looking for things like error conditions, orders in a certain state, or tickets that have risen to some severity level, you can inexpensively copy that data into a sparse index. This gives you quick access to those lists without having to run slow and expensive scan operations.

Understanding how to use sparse indexes might also help you reconsider which access patterns the user-facing piece of your application actually needs. For example, modeling your application to get a list of all your users when it's only used for internal reporting might be an unnecessary use of DynamoDB's power. You'd likely be better off replicating that data to something like MySQL using DynamoDB streams. Ask yourself, "What happens if I have thousands of users banging up against my table, and what do they need to do quickly?" That's what I try to optimize for.

8. Have a good sorting strategy

DynamoDB gives you limited querying capabilities. You essentially request a partition and have some control over the contents of the sort key. The 'begins_with' option is the closest thing you have to true "search" capabilities. Other than that, you're basically looking for sort keys that are between certain values, or perhaps greater than or less than some value. But because DynamoDB uses lexicographical sorting, there are some really handy use cases that become possible. The most frequent use case is likely needing to sort by a timestamp. However, epoch timestamps or ISO 8601 dates can lack uniqueness, are easy to guess, and aren't always URL-friendly. Alex recommends using KSUIDs (K-Sortable Unique Identifiers) created by the team at Segment. This gives you date sortable IDs, but also the uniqueness required.

9. Don't be afraid of migrations

When designing tables, we have a tendency to ask, "What if I need to change this thing down the road?" With a few thousand records, it doesn't seem that difficult. But what if the table grows to hundreds of gigabytes? All of a sudden, the process of adding new access patterns or

attributes, and changing the underlying data model, suddenly seems like an overwhelming endeavour. Changing a data model is never easy, but Alex dedicates an entire chapter that outlines some strategies that can make these types of migrations easier. For me, I started noticing a lot of flexibility once I started separating my application attributes from my indexing attributes.

10. Parallel scans are your new best friend

One way to make the aforementioned migrations easier is with parallel scans. Couple these with Lambda functions, and you can dial up the concurrency to process as much data as you want, and in relatively short order. Alex suggests using Step Functions to fan out parallel scans to Lambda workers. As soon as you figure out how to do that, all of a sudden, all these daunting ETL tasks are much more approachable.

11. Ensure uniqueness across multiple attributes with transactions

In the book, Alex gives a strategy for ensuring uniqueness across multiple attributes. I was glad that he mentioned this since I often need to deal with this in my own designs. In cases where you might require a unique username and email address, Alex points out that you don't want to bake them both into the primary key, because that won't work. Instead, you need to create two separate items and use a condition expression and a transaction. This way, if any of those operations fail, the entire operation will be rolled back.

12. Precompute your reference counts

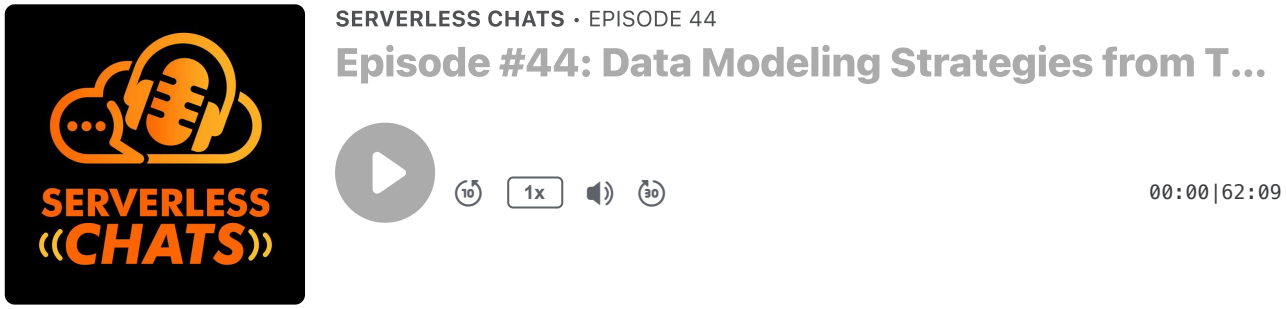
Unfortunately, getting a count of items in a DynamoDB table isn't as easy as running a `select count(*) from myTable` statement. The need to do this, however, is still critical for most applications. There are some bad ways to accomplish this, like scanning the entire table and counting the items that match a certain criteria. Or maybe if I have the right indexes set up, I could run a query on a smaller subset of data. But either way, we need to understand that these methods are probably a terrible (and expensive) idea. Alex explains that if you need to maintain reference counts, it's best to increment a counter in a parent item as new items are

added or updated. This can be done using transactions (or even simple batch writes), or by using DynamoDB streams.

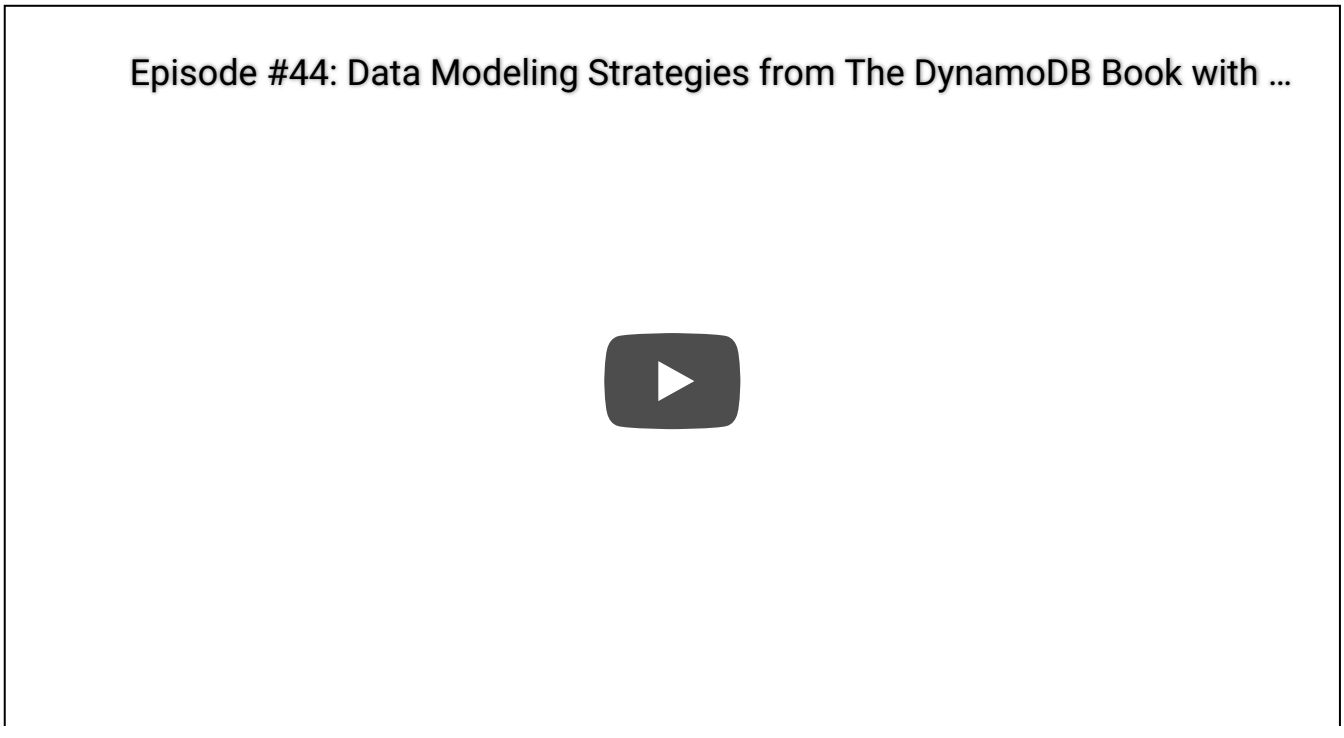
Final Thoughts

If you're using (or thinking about using) DynamoDB, then Alex's book is an absolute must-have. I've been using it as my go-to reference for pretty much everything DynamoDB, and I think it will make a great companion to anyone else exploring this awesome tool. To listen to Alex and me chat in greater detail about his book, listen to our Serverless Chats episode. If you want a copy of the book, you can buy it at dynamodbbook.com.

Listen to the episode:



Watch the episode:



Tags: aws, data modeling, dynamodb, nosql, serverless

Did you like this post? 👍 Do you want more? 🙌 Follow me on Twitter or check out some of the projects I'm working on.

🗓️ *This post was originally published on **April 16, 2020** by Jeremy Daly.*