SUPERCHARGE YOUR DATA MODELING

The DynamoDB Book

Alex DeBrie

The DynamoDB Book

Alex DeBrie

Version 1.0.1, 2020-04-16

Table of Contents

Preface

My DynamoDB story begins with an abundance of unfounded confidence, slowly beaten out of me by the club of experience.

I first used DynamoDB in 2015 while I was building an internal application for the engineering team I was on. Whenever there was a friction point in the development process, an engineer could type a quick command in Slack. That message would be sent to my webhook and stored for posterity. Occasionally, we would pull all the data out of my table, look it over, nod approvingly, and move along.

I was proud of my application. But it's a good thing it never had to scale past single-digit requests per day.

At my next job, I helped implement a few different data models. I was all-in on the serverless ecosystem at this point, and DynamoDB was the en vogue database for serverless applications. I read what I could and thought I was pretty good at DynamoDB as I implemented a RDBMS data model on top of DynamoDB.

In December of 2017, I listened to podcasts of breakout sessions from AWS re:Invent on my commute to and from work. One morning, I stumbled upon a talk on Advanced Design Patterns with DynamoDB by some guy named Rick Houlihan. I expected a tidy review of the impressive DynamoDB knowledge I had already gathered and stored. Wrong.

That talk changed my career.

I couldn't believe the witchcraft I was hearing. Modeling with NoSQL databases is nothing like modeling with relational databases! Storage is cheap; it's compute that's sacred! Put all your

data into a single table!

When I got home that night, I watched the video recording of the same session. Sure enough, my ears weren't deceiving me. Rick Houlihan was an assassin, and DynamoDB was his weapon of choice.

Over the next month, I spent my free time trying to decipher the ins and outs of Rick's talk. I compared my notes against DynamoDB documentation and tried replicating his models in my own examples. The more I read, the more excited I became. My Christmas holiday was focused on sharing this knowledge with others, and in January 2018, I published DynamoDBGuide.com, a website aimed at sharing my newfound love for DynamoDB.

In the time since hitting publish on that site, I've learned much more about DynamoDB. DynamoDBGuide.com will get you started, but it's not going to take you to the top of the mountain.

This is the book I wish I had when I started with DynamoDB. The first few chapters will warm you up with the basics on DynamoDB features and characteristics. But to paraphrase James Carville, "It's the data model, stupid!" The hard part about DynamoDB is making the shift from an RDBMS mindset to a NoSQL mindset. We'll go deep on DynamoDB data modeling in this book, from discussion of the DynamoDB API, to various strategies to use when using DynamoDB, to five full-length walkthroughs. There are some things you can only learn by doing, but it doesn't hurt to have a guide along the way.

And that Rick Houlihan guy? He now has the most popular re:Invent session year after year, enjoys a cult following on Twitter, and somehow agreed to write the foreword to this book.

Acknowledgements

This book would not have been possible without help from so many people. I'm sure to leave out many of them.

Thanks to Rick Houlihan for introducing me to DynamoDB, for answering my questions over the years, for taking it easy on me in a live debate in early 2020, and for writing the foreword to this book.

Thanks to many other folks at AWS for building a great service, helping increase my engagement with DynamoDB, and teaching me. There are too many to name here, but Seayoung Rhee, Lizzy Nguyen, Pete Naylor, Edin Zulich, and Colin Lazier have all been great helps.

This book has been significantly improved by reviews, discussions, and encouragement from a number of people in the community. Special thanks to Paul Swail, Jeremy Daly, Jared Short, Corey Quinn, and Shawn Wang (aka Swyx) for encouragement and help with the draft, and thanks to Chris Biscardi for consistent support and encouragement. Also, thanks to so many of you that sent me messages or responded to emails and gave feedback on preview chapters.

Thanks to Ryan Hinojosa for great design help (a perennial weakness for me) including detailed feedback on fonts and Powerpoint slides. He's the reason your eyes aren't bleeding as you read this. Thanks, also, to Daniel Vassallo for sharing not only his experience self-publishing a technical book but also his AsciiDoc config without any expectation of return. I spent way too much time trying to configure AsciiDoc myself. Thanks to David Wells for sharing his marketing knowledge with me and for the breakdown of my landing page. Thanks to Andrea Passwater for assistance with copy on the DynamoDB Book website and for teaching me a ton about writing when we worked together.

Foreword

For some, data modeling is a passion. Identifying relationships, structuring the data, and designing optimal queries is like solving a complex puzzle. Like a builder when a job is done and a structure is standing where an empty lot used to be, the feelings are the same: satisfaction and achievement, validation of skill, and pride in the product of hard work.

Data has been my life for a long time. Throughout a career that has spanned three decades, data modeling has been a constant. I cannot remember working on a project where I did not have a hand in the data layer implementation, and if there is one place in the stack I am most comfortable, it is there. The data layer is not usually considered to be the most exciting component, which meant I owned it. Determining the best way to model the data, crafting the most efficient queries, and generally making things run faster and cheaper has always been a big part of my developer life.

Data layer implementation for the masses has been about abstracting developers from the internals of the database. SQL is generally considered painful by developers, and as a result many tools have been invented over the years to facilitate this abstraction. ORM frameworks such as Hibernate, JDO, Django and nHydrate exist to insulate developers from the language of data which for them was convoluted and difficult to understand. These tools have succeeded to some extent by insulating developers from the inner workings of the database. However, this has impacted the overall cost and/or performance of the system in most cases. Until recently this has been a non-issue since the majority of workloads did not scale to the point where a pain threshold was crossed.

There is a general belief that ORM tooling increases developer productivity by allowing them to focus on the code instead of the

data. Create business objects in code and let the underlying ORM tooling manage the database for you. This type of abstraction can produce reasonable results when relational databases are queried to produce "views" of data required by the application. A "view" is by definition an abstraction of the data model since it is a representation of data stored across many tables and is not persisted in the same form that the data is presented. The developer specifies a set of queries to populate business objects, and then they can leverage the ORM framework to handle the more complex process of managing data stored in multi-table relational models.

For decades this approach has proven to be an effective process for developing software services. Understanding the data was left to a small subset of software professionals, the Database Administrators or DBA's. Most of the time the job of the DBA was simply maintaining the infrastructure supporting the database. However the more valuable DBA's spend their time optimizing queries to increase efficiency and performance of the database. These DBA resources are the revered ones, considered practitioners of the black arts and masters of SQL, the language of data.

The most interesting thing about query optimization is that it is not unlike optimizing code. In the end it's all about time complexity. Identifying which columns to index to avoid table scans, how to structure tables to eliminate expensive joins, and leveraging stored procedures instead of processing data in the app tier. It is all about investigating how the system is accessing the data, and a skilled DBA will use query analyzers and break down access patterns to identify inefficiencies and introduce data model and query optimizations that inform how the entire stack is designed and operates.

What this all means is in the end the data actually is the code. There is no getting away from the fact that abstraction and optimization

are polar opposites from a design perspective. Data model abstraction makes it easy for the developer to deliver a working system, but abstraction requires a system that makes assumptions about access patterns which leads to less than optimal solutions. It is not possible to be agnostic to all access patterns and at the same time be optimized for any but the simplest. In the end DBA skills are always required to make things run smoothly at scale. There is just no avoiding it.

As modern applications have evolved, the datasets and transaction volumes they support have grown exponentially. These days it is common to hear about the need to design for scale, but what that really means depends on the application itself. There are dimensions to scale, and what it means to scale varies significantly depending on the size and the velocity of the workload. Scaling for high volumes of small transactional data requires much different architecture than scaling to support large amounts of data at rest. As modern applications have evolved, so have the demands they place on the database and the relational database technology that we have used for the last few decades to meet those demands is simply failing to deliver at scale.

To understand why this is the case, it is important to consider the problem the relational database was designed to solve. Technology is always invented to make a process faster, cheaper, or more reliable. In the case of relational database technology, the driving force was cost. Storing data was expensive, really expensive. A normalized relational data model combined with an ad hoc query engine reduced storage cost significantly by deduplicating data at rest and enabling applications to reconstruct formatted views of data on demand. The normalized relational model increased time complexity for queries as executing complex joins is not free; however, the cost of CPU was not significant compared to the cost of storage in 1980.

Today it becomes immediately apparent after the most cursory analysis that the cost dynamics of managing applications at scale have completely reversed. Storage is literally pennies per gigabyte and CPU time is where the money is being spent. In today's IT environment, the relational database has become a liability, not an asset, and the CPU cost of executing ad hoc queries at high transaction rates against large datasets has become a barrier to scale. The era of Big Data has pushed the traditional relational database platforms to the breaking point and beyond, and has produced requirements that simply cannot be met by legacy relational database technology.

Enter the NoSQL database. Built on the concept of a denormalized data model, NoSQL databases are designed to eliminate the need for complex joins between tables. To accomplish this, all data objects that had previously been distributed across multiple tables in a normalized data model are consolidated into a common collection or "schemaless" table in the NoSQL database. These objects are then decorated with common attributes and indexed to produce groupings that meet the conditions of queries made by the application. In the chapters that follow, Alex describes this process in great detail, but for this discussion suffice to say that in NoSQL we are collapsing the "rows" of all the "tables", describing an entity into a common collection or monolithic object in a manner that is strongly tied to a specific access pattern.

The vast majority of software is written to automate a commonly executed process. Shopping carts, dashboards, ETL, ITSM, trading, online banking, and almost every other application that people interact with can be described as a collection of repeatable processes. These types of applications are highly transactional and are commonly described as Online Transaction Processing (OLTP) apps. OLTP apps do the same things over time without deviation, and many of these applications do not need the kind of analytics and complex computations that require an ad hoc query engine,

which makes NoSQL a perfect choice.

NoSQL is a journey. Over the years my understanding of the technology has evolved and changed dramatically. When I was first introduced to NoSQL like many developers I could not understand how or why anyone would want to use this technology to build an application. It was difficult to see value in anything but the most basic use cases where the workload is simply retrieving blobs of data with a Key/Value access pattern. Most applications, however, are built on highly relational data with much more complex access patterns. How that kind of data could be managed in a NoSQL database is certainly not immediately obvious.

Over the last decade, I have personally executed well over a thousand NoSQL design consultations. In addition to hundreds of Amazon services, the projects I have modeled run across almost every industry sector, and I have seen deployments scaled to unimaginable levels of throughput and storage capacity. I have also seen how the power of Cloud Native NoSQL services like DynamoDB can be harnessed by individual developers, startups, and small businesses to do incredible things for a fraction of the cost of deploying legacy technology built for yesterday's on prem IT models. There is no doubt in my mind that today the future of database technology is NoSQL, and the future of NoSQL is the cloud.

After all is said and done, I have come to realize that there is one underlying principle when it comes to NoSQL. At the core of all NoSQL databases there is a collection of disparate objects, tied together by indexed common attributes, and queried with conditional select statements to produce result sets. NoSQL does not join data across tables, it effectively achieves the same result by storing everything in one table and using indexes to group items. I will never get tired of watching a team I am working with turn the corner and have what I call the "light bulb" moment when they

finally understand data modeling for NoSQL, and how truly powerful and flexible it really is.

To the reader, I would emphasize this. Alex has invested himself in the subject matter he is presenting here for you in great detail, and he is no doubt a more than capable guide. Open your mind and let this book lead you to that lightbulb moment. When you have it, the mantra of the NoSQL developer will become clear.

Data is code.

Rick Houlihan

Chapter 1. What is DynamoDB?

> The future is here, it's just not evenly-distributed.

— William Gibson, 1990

Mr. Gibson wasn't talking about databases when he uttered these words, but tweak them slightly and it fits remarkably well. And no, I don't mean the way I react in disappointment when reviewing my old data models ("The *data* is here, it's just not evenly-distributed..."). Rather, I mean something like the following:

> *The future is here, but some people are still using legacy databases.*

I've been using DynamoDB for the last five years, and I can't go back to any other database. The billing model, the permissions system, the scalability, the connection model—it's an improvement from other databases in so many ways.

It took time and some failed experiences to learn how to model properly in DynamoDB, but it was worth the cost. This book is designed to make that journey easier for you. In the coming chapters, you'll get almost everything I know about DynamoDB. From modeling relationships to modeling reference counts, from handling uniqueness to handling migrations, you get it all.

But first, let's cut loose the inaccurate information you may have picked up around DynamoDB and NoSQL over the years.

Behold: **the Five Misconceptions about DynamoDB.**

1. DynamoDB is just a key-value store.

You may have heard that DynamoDB can only handle simple access patterns. Insert an individual item and read it back; anything

more complex, you'll need to use a "real" database.

Nothing could be further from the truth. DynamoDB can handle relationships between multiple records (See Chapters 11 and 12 on one-to-many relationships and many-to-many relationships) and complex requirements around filtering (see Chapter 13). The way you do it is different than in a relational database, but it's still possible.

In the walkthrough examples in Chapters 18-22, we go through some complex patterns, including modeling the entire GitHub metadata backend. The more complex examples involve more than ten entities and over twenty access patterns. I only stopped at ten entities because it starts to get repetitive after a while.

If you can model it in an RDBMS, you can probably model it in DynamoDB.

2. DynamoDB doesn't scale.

The oddest misconception I hear about DynamoDB is that it doesn't scale. "Yea, it may work for your simple app to hold a bit of data, but don't expect to handle multiple users with it."

This is poppycock. Amazon (both Amazon.com retail and Amazon Web Services) requires the use of DynamoDB for all Tier 1 services. A Tier 1 service is any service that would lose money if it went down.

Think of the high-traffic Amazon services: the shopping cart, inventory, AWS IAM, AWS EC2. DynamoDB can handle scale.

And DynamoDB is used by a number of high-volume customers outside Amazon as well. Lyft uses DynamoDB to handle locations for all rides in its application. Think about the number of Lyft cars on the road at any given time. Likewise, many mobile game

companies rely on DynamoDB to handle high-volume transactions against core data at scale.

I think this misconception stems from folks that have used DynamoDB incorrectly. And it's true that if you rely on Scans in your access patterns, or if you place all your data into a single partition, DynamoDB won't scale as far as you want it to. But that is an issue of misuse, rather than a faulty tool.

3. DynamoDB is only for enormous scale.

A third misconception is the reverse: DynamoDB should *only* be used at scale.

This is false as well. DynamoDB has taken off in the serverless community for all kinds of applications. Its combination of easy provisioning, flexible billing model, HTTP connection model, and pay-per-use pricing all fit well within the serverless ecosystem. I default to DynamoDB for all new applications, and countless other developers are doing the same.

4. You can't use DynamoDB if your data model will change.

As we learn the data modeling concepts of DynamoDB in subsequent chapters, I'll often emphasize the point that you must know your access patterns before you model. With a relational database, you often design your table based on your objects without thinking about how they'll be queried. With DynamoDB, you can't design your table until you know how you'll use your data.

Because you have to know your access patterns upfront, many people take this to mean your access patterns can't change over time. That's not true! You can evolve your DynamoDB data model just like you can evolve with other databases. The same general DynamoDB principles apply—you must know your new access patterns before modeling them out—but many changes are additive

to your existing model. In the event you need to modify existing records, there's a straight-forward pattern for doing so.

We'll discuss migration strategies in Chapter 15 of this book, and we'll show these strategies in action in Chapter 22.

5. You don't need a schema when using DynamoDB.

DynamoDB (and other NoSQL databases) are often called *schemaless*, leading developers to think they'll gain speed by not having to design their entities and relationships upfront.

But truly schemaless data is madness. Good luck reading out the garbage you've written into your table.

While it's true that DynamoDB won't *enforce* a schema to the extent that a relational database will, you will still need a schema somewhere in your application. Rather than validating your data at the database level, it will now be an application-level concern.

You still need to plan your data model. You still need to think about object properties. Don't use NoSQL as an excuse to skimp on your job.

With these misconceptions partially dispelled, let's dive into what DynamoDB is.

1.1. Key Properties of DynamoDB

DynamoDB is a fully-managed, NoSQL database provided by Amazon Web Services.

The sentence above is more than just a bunch of buzzwords—it describes the core of how DynamoDB is different than other

databases. Let's see some of the features that distinguish DynamoDB from other databases.

- **Key-value or wide-column data model**

DynamoDB is a NoSQL database. But NoSQL is a terrible name, as it only describes what a database *isn't* (it's not a database that uses the popular SQL query language used by traditional relational databases), rather than what it *is*. And NoSQL databases come in a variety of flavors. There are document databases like MongoDB, column stores like Cassandra, and graph databases like Neo4J or Amazon Neptune.

DynamoDB has support for two similar data models. First, you can use DynamoDB as a key-value store. Think of a key-value store like a giant, distributed hash table (in your programming language of choice, it may be called a dictionary or a map). A hash table contains a large number of elements, each of which are uniquely identifiable by a key. You can get, set, update, and delete these elements by referring to its primary key. Hash tables are a commonly-used data structure because of their fast, consistent performance no matter the size of the data set.

The problem with a key-value store is that you can only retrieve one record at a time. But what if you want to retrieve multiple records? For example, I might want to read the ten most recent readings for my IoT sensor, or I may want to fetch a Customer and all of the Customer's Orders over the last 6 months.

To handle these more complex access patterns, you can also use DynamoDB as a wide-column store. A wide-column store is like a super-charged version of a hash table where the value for each record in your hash table is a B-tree. A B-tree is another commonly-used data structure that allows you to quickly find a particular item in the data structure while also allowing for range queries. Think of a B-tree like a phone book. It is relatively easy to

open a phone book and find the entry for DeBrie, Alex. It's also straightforward to find all entries with the last name "Cook" or to find all entries between "Benioff, Marc" and "Bezos, Jeff".

> To my readers born after 1995, a "phone book" was a real-life physical book containing the names, addresses, and phone numbers of everyone in a particular location. The phone book was in alphabetical order by last name. Imagine that! This was before fancy things like Google and Facebook rendered phone books obsolete for anything other than kindling.

To complete the analogy, you could think of a wide-column store as a bookshelf full of phone books, each one for a different city. The bookshelf is the hash table, and each phone book is a B-tree. When someone asks you to find some entries--"Give me all entries between Buffett, Warren and DeBrie, Alex in Omaha, NE"--you can quickly find the proper phone book and the relevant range in the phone book.

Proper data modeling with DynamoDB is all about making sure you have the right books on your bookshelf with the entries in the right order. We'll cover much more about DynamoDB's data model in subsequent chapters.

- **Infinite scaling with no performance degradation**

DynamoDB was built to be fast and scalable, and it delivers on both counts. Most operations in DynamoDB have response times in single-digit milliseconds. If you need better than that, AWS offers DynamoDB Accelerator (DAX), which is a fully-managed in-memory cache for your DynamoDB table.

DynamoDB also scales to as large a table as you need. You'll almost certainly hit limits on your wallet before you hit actual limits on DynamoDB table size, as there's no theoretical limit to how big a DynamoDB table can be.

The real magic comes in the combination of these two factors—you

will continue to get lightning-fast response times as your database grows. You can scale your database to 10 TB, and you won't see any performance degradation from when your database was only 10GB. There are DynamoDB users with tables over 100TB, and they still see response times in the single-digit milliseconds.

This performance pattern is not found with SQL-based, relational databases whose performance tails off as your application scales.

- **HTTP connection model**

All requests to DynamoDB are made to the DynamoDB API via HTTP requests. This is in contrast to most database systems that initialize persistent TCP connections that are reused many times over the lifetime of an application.

The HTTP-based model can be a bit slower than the persistent TCP connection model for some requests, since there isn't a readily-available connection to use for requests. However, persistent connections have downsides as well. You need some initialization time to create the initial connection. Further, holding a persistent connection requires resources on the database server, so most databases limit the number of open connections to a database. For example, PostgreSQL, the popular open-source relational database engine, sets the default number of maximum connections to 100.

With DynamoDB, you have none of these limitations. Your application can start querying DynamoDB right away without creating a connection pool. Further, you can connect to DynamoDB with a virtually unlimited number of concurrent requests, provided you have paid for the throughput. These factors are a key reason that DynamoDB is so popular for the hyper-ephemeral compute (or "serverless") use case discussed later in this chapter.

- **IAM authentication**

DynamoDB uses AWS IAM for authentication and authorization of database requests rather than a username and password model that is common with other database systems.

The AWS IAM method is convenient for a number of reasons. First, most users of DynamoDB are likely using AWS services for compute, whether EC2 instances, ECS containers, or AWS Lambda functions. Each of these compute options can be associated with IAM Roles that specify the exact IAM permissions for the given compute instance, and your application doesn't need to worry about credential management and rotation.

Second, AWS IAM provides a granular permission system that is well-known by AWS administrators, rather than a database-specific permission system whose intricacies are less well-known. With basic IAM syntax, you can write an IAM policy that allows only the specific actions (e.g., "GetItem") on a specific DynamoDB table, without allowing other actions (e.g. "UpdateItem", "DeleteItem") on that same table.

If you want to get more granular, you can even limit IAM permissions such that the authenticated user may only operate on DynamoDB items with certain primary keys or may only view certain attributes on allowed keys.

- **Infrastructure-as-code friendly**

Infrastructure-as-code is a pattern for managing application infrastructure in which all infrastructure needed for an application is described and maintained in code. All work to provision, update, or remove infrastructure resources should be done in automated fashion using an infrastructure-as-code tool rather than a manual, ad-hoc fashion by an engineer. Many of the infrastructure-as-code tools use a declarative syntax to describe the infrastructure you want. The two most popular declarative tools are AWS CloudFormation and Hashicorp's Terraform.

For most databases, managing the database via infrastructure-as-code is an awkward endeavor. You can do part of the work to prepare your database for use in your application—such as creating the database and configuring network access—using a tool like Terraform or CloudFormation. However, there's another set of tasks, such as creating database users, initializing tables, or performing table migrations, that don't fit well in an infrastructure-as-code world. You will often need to perform additional administrative tasks on your database outside your infrastructure-as-code workflow.

In contrast, DynamoDB works perfectly within an infrastructure-as-code workflow. Creating a DynamoDB table and specifying the primary key and secondary indexes can be done declaratively via Terraform and CloudFormation. You can handle database users and access control by creating AWS IAM roles and policies as well. There are no out-of-band administrative tasks that don't fit in an infrastructure-as-code world.

- **Flexible pricing model**

DynamoDB offers a flexible pricing model that is more attractive than other options. With most databases, you specify the size of the server. You look at various combinations of CPU, RAM, and disk to find the server that works for you. This is suboptimal as you're paying for resource capacity ("How much RAM do I need?") rather than workload capacity ("How many read queries per second will I have?").

In contrast, DynamoDB is priced directly based on the amount of workload capacity you need. You specify the throughput you want in terms of Read Capacity Units and Write Capacity Units. A Read Capacity Unit gives you a single strongly-consistent read per second or two eventually-consistent reads per second, up to 4KB in size. A Write Capacity Unit allows you to write a single item per second, up to 1KB in size.

There are two amazing things about DynamoDB's pricing model. The first is that you can tweak your read and write throughput separately. Usually your database is fighting over the same resources for handling read and write workflows. Not so with DynamoDB. If you have a write-heavy workload, you can crank up your write throughput while leaving your read throughput low.

The second amazing thing about DynamoDB's pricing model is that you can dynamically scale your throughput up and down as needed. If, like most applications, your traffic is significantly lower at night and on weekends, you can scale the throughput down to save money. This is simply not feasible with most databases.

If you don't know your access patterns or don't want to take the time to capacity plan your workload, you can use On-Demand Pricing from DynamoDB. With this pricing model, you pay per request rather than provisioning a fixed amount of capacity. The per-request price is higher than the provisioned mode, but it can still save you money if you have a spiky workload that doesn't take full advantage of your provisioned capacity.

The best part is that you can switch between pricing models over time. I recommend that you start with On-Demand Pricing as you develop a baseline traffic level for your application. Once you feel like you have a strong understanding of your traffic needs, you can switch to defining your Provisioned Throughput to lower costs.

- **Change data capture with DynamoDB Streams**

One of the coolest features of DynamoDB is DynamoDB Streams. With DynamoDB Streams, you get a transactional log of each write transaction in your DynamoDB table. You can programmatically process this log, which opens up a huge number of use cases.

Event-based architectures have exploded in popularity in recent years, partly due to architectural needs with the rise of

microservices and partly due to the popularization of technologies like Apache Kafka and AWS Lambda. There are a number of efforts to retrofit change data capture onto existing databases in order to enable other services to react to changes in a database. With DynamoDB, this feature is built directly into the core and doesn't require difficult workarounds and additional infrastructure maintenance.

- **Fully-managed**

DynamoDB is a fully-managed database. You won't need to spin up server instances, install software, manage failovers, handle backups, upgrade software, or handle any other basic database maintenance tasks. This offloads a ton of low-level work from your plate.

With a fully-managed database, you do give up some element of control. You can't SSH into your instances and look at log files. You can't change a bunch of configuration values. But you should question whether managing a database is something you really *want* to control. Data safety and reliability is one of the more important factors in any application's success. You should question who is best able to provide the safety and reliability you need.

Just as more and more companies are moving from on-prem data centers to cloud-based compute, we're seeing people prefer fully-managed database solutions due to the complexity required in keeping a database running. AWS has an enormous team of world-leading experts in building and maintaining critical infrastructure. It's likely they are able to handle this better than you and in a more cost-effective manner.

1.2. When to use DynamoDB

In the previous section, we learned some key characteristics of

DynamoDB. The next question to ask is when you should choose DynamoDB in your application. There's been a proliferation of database options in recent years, and you should carefully consider your application needs when deciding on your database.

In the last few years, two areas have been driving the adoption of DynamoDB:

1. hyper-scale applications; and
2. hyper-ephemeral compute.

The first occasion is a result of business realities—are we going to have so much traffic that some traditional technologies won't be able to keep up? The second occasion is a result of technical choices—are we making architectural choices that make it difficult to use certain technologies?

Even outside these situations, DynamoDB is a great choice for most workloads.

Let's review these core use cases for DynamoDB.

1.2.1. Hyperscale

The first core use case for DynamoDB is for hyper-scale applications. This is what DynamoDB was made for. To understand this, let's have a history lesson.

From the 1970s through the year 2000, the relational database management system (RDBMS) reigned supreme. Oracle, Microsoft SQL Server, and IBM's DB2 were popular proprietary choices, while MySQL and PostgreSQL took off as open-source offerings.

The relational database was built for a world where storage was the limiting factor. Memory and disk were expensive in the early

computing age, so relational databases optimized for storage. Data would be written once, in a normalized fashion, to avoid duplicating information. Developers learned SQL (which stands for "Structured Query Language") as a way to reassemble related bits of data spread across multiple locations. SQL and database normalization were focal points of developer education.

In the early 2000s, we started to see the first signs of cracks in the RDBMS monopoly. A few factors contributed to this breakup. First, the price of storage fell through the floor. Hard drives went from costing $200,000 per GB in 1980 to $0.03 per GB in 2014. Conserving on storage was no longer economically necessary.

Further, the advent of the internet changed the needs of applications. All of a sudden, your potential market was the entire population of the globe, and users demanded faster and faster performance from web applications. This led to a rethinking of database design.

Amazon.com has been known for its "Cyber Monday" deals. This is an online version of "Black Friday", the major shopping day right after Thanksgiving where people are gearing up for the holiday season. As Cyber Monday became more and more popular, Amazon had trouble keeping up with the load on their infrastructure. In 2004, this came to a head with a number of scaling challenges that led Amazon to rethink some of their core infrastructure.

The public got a peek behind the curtain of this new infrastructure through the Dynamo Paper. Published by a group of Amazon.com engineers in 2007, the Dynamo Paper described a new kind of database. It rethought a number of key assumptions underlying relational databases based on the needs of modern applications.

Based on the Dynamo Paper and a follow-up blog post from Amazon CTO Werner Vogels, some of the main reasons for

creating Dynamo were:

- **Inability to use advanced relational features at scale.** As Amazon scaled up their operations, they couldn't use costly operations like joins because they were too slow and resource-intensive. Werner shared that 70% of their database operations operated on a single record, and another 20% would return multiple rows but would use only a single table. Thus, over 90% of all queries did not use joins!
- **Ability to relax relational constraints.** Relational databases had strict consistency functionality, which roughly means that clients will see the same data if querying at the same time. Consistency is crucial for certain use cases—think bank account balances—but less critical for others. Like joins, consistency is expensive, particularly as you scale.

The two aspects above—join functionality and strong consistency—often meant that all data for an application needed to be stored on a single server instance. A single instance requirement limits your scalability options, as a server with 64 CPUs and 256GB of RAM is significantly more expensive than 16 servers with 4 CPUs and 16GB of RAM each.

The engineers behind the Dynamo Paper realized the power of relaxing the two constraints above. This allowed them to shard application data across machines without a loss of application performance. Not only that, it allowed them to scale essentially infinitely without any performance degradation. This was a huge difference from the dominant relational databases of the time.

The Dynamo Paper was enormously influential in the database space and helped kickstart the NoSQL revolution. Open-source NoSQL options like MongoDB, Apache Cassandra, and others took inspiration from concepts in the Dynamo Paper, and Amazon released a fully-managed version of its in-house database in the

form of DynamoDB in 2012.

1.2.2. Hyper-ephemeral compute (aka 'Serverless')

The second core use case for DynamoDB is more recent. Like the first, it came about due to a paradigm shift in how applications were built. This shift meant that certain technologies no longer fit the new model, so developers looked for other options to fill the gap.

The first paradigm shift was due to the plummeting cost of storage and the increasing performance needs due to the internet. The second paradigm shift was in the rise of what I call 'hyper-ephemeral compute'. Again, let's look at the full history to understand this change.

For the last forty years, the basic compute model for applications has been relatively consistent. Generally, you have some amount of CPU, memory, and disk available somewhere as a server. You execute a command to start a long-running process to run your application. Your application might reach out to other resources, such as a database server, to help perform its actions. At some point, your application will be restarted or killed.

This basic pattern has held across several different compute paradigms. First they ran on bare metal instances before moving to virtualized service instances. These virtualized machines first ran on on-prem servers before moving to the cloud. More recently, applications are running in containers rather than full-fledged virtual machines.

Each of the shifts above was important, but you still had the same general shape: a long-running compute instance that handled multiple requests over its lifetime.

With AWS Lambda, everything changed. AWS Lambda

popularized the notion of event-driven compute. You upload your code to AWS, and AWS will execute it in response to an event.

When your compute is pull-based ("I have an event that needs handled, give me some compute") rather than push-based ("I expect X traffic over the next 3 hours, so provision Y instances to handle that traffic"), lots of things change. The most significant changes for our purposes are the speed of provisioning and the dynamism of your compute locations.

Because our compute may not be created until there's a live event, ready and waiting to be processed, we need the compute provisioning speed to be as fast as possible. AWS Lambda has optimized a lot of this by making it lightning fast to pull down your code and start its execution, but you need to do your part as well. And doing your part means avoiding long initialization steps. You don't have the time to set up persistent database connection pools.

Further, the dynamism of your compute locations adds new challenges as well. In the old world, you knew where your application instances were, whether that was a particular rack of servers in a colo site or certain instance IDs in AWS. Because you knew this, you could use services that should be partitioned from the public internet, like relational databases or caches that use TCP connections. With AWS Lambda, you don't know where your compute will be, and it makes network partitioning harder and slower. It's better to rely on services that use stateless connection models like HTTP and rely on client-server authentication to filter out bad traffic.

DynamoDB is a perfect fit for these hyper-ephemeral applications. All access is over HTTP and uses AWS IAM for authentication. AWS is able to handle surges in traffic by using a shared Request Router across instances, which authenticates and validates your request before sending it to the proper shard for processing. This means you don't need network partitioning to protect your database. And

since you don't need network partitioning, your ephemeral compute is able to access your database without first setting up the proper network configuration.

1.2.3. Other situations

While hyper-scale and hyper-ephermeral compute use cases are the main ones, DynamoDB fits in other situations as well. A few of these situations are:

- **Most OLTP applications**: The hyperscale and hyper-ephemeral compute uses above are two examples of on-line, transactional processing (OLTP) applications. OLTP applications are those where end users are reading and writing small bits of data at high speeds. This describes most applications that you interact with as a user—Amazon.com, Twitter, and Candy Crush. The contrast of OLTP is on-line analytical processing (OLAP), which is used for more internal use cases where you're performing giant analyses of data sets, usually for reporting purposes. DynamoDB is a great fit for nearly all OLTP applications due to its fast, consistent performance.

- **Caching**: DynamoDB can be used as a cache to store the results of complex, frequently-accessed queries from other databases or other operations. You won't get the same level of performance as you would with a fully in-memory cache like Redis, but DynamoDB can be a fast, low-maintenance, cost-effective solution for specific use cases.

- **Simple data models**: If you have a simple data model that doesn't require complex querying, DynamoDB is a great fit, especially when you are mostly doing key-value lookups. A typical example here is a session store where you're saving session tokens that are used for authentication. It's not quite a caching use case, as this is the primary data store for that information, but it uses similar access patterns to a cache.

1.3. Comparisons to other databases

Choosing a database is one of the most important architectural decisions you will make. This is both due to the importance of your data and the difficulty in performing a migration. There are several database options to choose from. This section looks at some popular options that are often considered against DynamoDB.

The three most frequent competitors to DynamoDB are:

- Relational databases
- MongoDB
- Apache Cassandra

Let's look at each of these in turn.

1.3.1. DynamoDB vs. Relational databases

The first competitor to DynamoDB is a traditional relational database. This includes proprietary options like Oracle, SQL Server, or Amazon Aurora, as well as open-source options like MySQL, MariaDB, and PostgreSQL. Relational databases are likely the most common alternative that developers consider when evaluating DynamoDB. The nice part about comparing DynamoDB with a relational database is that the strengths and weaknesses are pretty apparent and thus the choice is usually straightforward.

Relational databases include a lot of benefits. They're well-known by most developers, so there's less ramp-up time. Because of their familiarity, developers are (hopefully) less likely to make a critical data modeling error that will force a costly migration. The tooling for relational databases is strong in all popular programming languages, and relational database support comes native in many

web application frameworks. Finally, a relational database gives you a ton of flexibility in your data model, and you can iterate on your application more easily.

The main benefits of DynamoDB over a relational database are two mentioned in the previous section. DynamoDB will allow you to scale significantly further than a relational database, and DynamoDB works better with the hyper-ephemeral compute model.

There are two situations where it's challenging to choose between DynamoDB and a relational database.

The first situation is when you need to make a temporal tradeoff about the amount of scale you need. Imagine you're building a new startup that, if successful, will hit hyper scale. Think Uber in its early days, where you could be serving billions of rides in a year. That scale will be difficult in a relational database (Uber has written posts on its usage of relational databases, though the databases are sharded and used in a key-value manner similar to DynamoDB).

However, using a relational database might enable faster development, both due to developer familiarity and due to the query flexibility that a relational database provides. In this sense, you're making a decision about whether to move faster now with the likelihood that you'll need to migrate if successful vs. moving more slowly now but knowing you won't need to migrate if you hit it big. That's a tough choice with no clear answer.

The second situation is when you're making a technological tradeoff about the rest of your application architecture. I'm a huge fan of serverless technologies, and I love using AWS Lambda for my applications. That said, if you're a small startup, you might find the core of your application fits better with the flexibility of a relational data model.

For me, I still choose DynamoDB every time because I think the benefits of serverless applications are so strong. That said, you need to really learn DynamoDB patterns, think about your access patterns upfront, and understand when you should deviate from DynamoDB practices built for scale.

1.3.2. DynamoDB vs. MongoDB

The second most common competitor to DynamoDB is MongoDB, the open-source NoSQL database that was started in 2007 by the company 10gen. MongoDB provides a similar sharding strategy to DynamoDB, which enables the same consistent performance as your application scales.

There are two main differences between DynamoDB and MongoDB. The first, and most important, is in the data model itself. MongoDB uses a document-oriented data model as compared to the wide column key-value storage of DynamoDB. Many people gloss over this difference when comparing the two databases, but the difference is important.

MongoDB's document-oriented model provides significantly more flexibility in querying your data and altering your access patterns after the fact. Several index types in MongoDB aren't present in DynamoDB, such as text indexing for searching, geospatial indexing for location-based queries, or multi-key indexes for searching within arrays.

While these indexes give you additional power, they come at a cost. Using these special indexes are likely to hurt you as your data scales. When you're talking about an immense scale, you need to use a targeted, specialized tool rather than a more generic one. It's like the difference between a power saw and a Swiss army knife—the Swiss army knife is more adaptable to more situations, but the power saw can handle some jobs that a Swiss army knife never

could.

Because this flexibility is available, you need to be sure to use MongoDB in the right way for your application. If you know your application won't need enormous scale, you are fine to use some of the more flexible features. However, if you are planning to scale to terabytes and beyond, your developer team needs the discipline to plan for that upfront.

One benefit of DynamoDB is that its rigidity limits you in a good way. As long as you aren't using full-table scans, it's tough to write an inefficient query in DynamoDB. The data model and API restrict you into best practices upfront and ensure your database will scale with your application.

The other thing to consider is the tradeoff between lock-in and the hosting options. If you're concerned about cloud provider lock-in, MongoDB is a better bet as you can run it on AWS, Azure, GCP, or the RaspberryPi in your closet. Again, my bias is toward not worrying about lock-in. AWS has, thus far, acted in the best interests of its users, and there are tremendous benefits of going all-in on a single cloud.

You should also consider how you're going to host your MongoDB database. You can choose to self-host on your server instances, but I would strongly recommend against it for reasons further discussed in the Apache Cassandra section below. Your data is the most valuable part of your application, and you should rely on someone with some expertise to protect it for you.

There are multiple MongoDB hosting options available, including MongoDB Atlas, which is provided by MongoDB, Inc., as well as DocumentDB, an AWS option which is MongoDB-compatible.

1.3.3. DynamoDB vs. Apache Cassandra

Apache Cassandra is an open-source NoSQL database that was created at Facebook and donated to the Apache Foundation. It's most similar to DynamoDB in terms of the data model. Like DynamoDB, it uses a wide-column data model. Most of the data modeling recommendations for DynamoDB also apply to Cassandra, with some limited exceptions for the difference in feature set between the two databases.

I think the unfamiliarity of the data model is one of the biggest hurdles to deciding to use DynamoDB. However, if you're deciding between DynamoDB and Cassandra, you've already decided to learn and use a wide-column data store. At this point, you're deciding between lock-in and total cost of ownership. Because of that, it's the option I least recommend when considering alternatives to DynamoDB.

We've touched on the lock-in problem in the MongoDB section, so I won't reiterate it here. The short answer is that if you're anxious about cloud portability, then Cassandra does provide you more optionality than DynamoDB. That's never been a huge concern for me, so I don't factor it into my decisions.

If the data model is the same and lock-in isn't a concern to you, the only difference remaining is how you host the database. If you choose DynamoDB, you get a fully-managed database with no server provisioning, failover management, automated backups, and granular billing controls. If you choose Cassandra, you have to hire a team of engineers whose entire job it is to make sure a giant cluster of machines with your company's most crucial asset doesn't disappear.

I'm being facetious, but only slightly. Operations pain is a real thing, and I have a hard time understanding why you would opt into hosting your own database when a reliable, managed solution

is available.

One of the biggest reasons to use a NoSQL database is its scalability. This scalability is handled by sharding data across multiple machines. If you have more data or activity than you expected, then you'll need more machines to handle the traffic. With Cassandra, you'll have to handle provisioning your instance, adding it to the cluster, re-sharding your data, and more. You'll be responsible for instance failures when they happen (and they will happen).

With DynamoDB, this is all handled. If you want more read capacity, you tweak a single parameter and update your CloudFormation stack. If you don't want to worry about capacity at all, channel your inner Elsa—use On-Demand pricing and "let it go".

There are a few hosting options around Cassandra. DataStax provides some hosting options for Cassandra. And at re:Invent 2019, AWS announced Managed Cassandra Service (MCS), a fully-managed hosting solution for Cassandra. These hosted options make the decision a little closer, and I understand that different companies and industries have different needs and regulatory environments, so there may be other factors involved. That said, I'd recommend DynamoDB over Cassandra wherever possible.

Chapter 2. Core Concepts in DynamoDB

> ## Chapter Summary
>
> This chapter covers the core concepts of DynamoDB. For those new to DynamoDB, understanding these concepts and terminology will help with the subsequent sections of this book.
>
> ## Sections
>
> 1. Basic vocabulary
> 2. A deeper look at primary keys & secondary indexes
> 3. The importance of item collections

Now that we know what DynamoDB is and when we should use it, let's learn some of the key concepts within DynamoDB. This chapter will introduce the vocabulary of DynamoDB—tables, items, attributes, etc.--with comparisons to relational databases where relevant. Then we'll take a deeper look at primary keys, secondary indexes, and item collections, which are three of the foundational concepts in DynamoDB.

2.1. Basic Vocabulary

There are five basic concepts in DynamoDB—tables, items, attributes, primary keys, and secondary indexes. Let's review each

of them.

2.1.1. Table

The first basic concept in DynamoDB is a table. A DynamoDB table is similar in some ways to a table in a relational database or a collection in MongoDB. It is a grouping of records that conceptually belong together.

A DynamoDB table differs from a relational database table in a few ways. First, a relational database table includes only a single type of entity. If you have multiple entity types in your application, such as Customers, Orders, and Inventory Items, each of them would be split into a separate table in a relational database. You can retrieve items from different tables in a single request by using a join operation to combine them.

In contrast, you often include multiple entity types in the same DynamoDB table. This is to avoid the join operation, which is expensive as a database scales.

Second, a relational database table has a specified schema that describes and enforces the shape of each record in the table. With DynamoDB, you do not declare all of your columns and column types on your table. At the database level, DynamoDB is *schemaless*, meaning the table itself won't ensure your records conform to a given schema.

The fact that DynamoDB (and other NoSQL databases) are schemaless does not mean that your data should not have a schema—that way leads to madness. Rather, your record schema is enforced elsewhere, in your application code, rather than in your database.

2.1.2. Item

An item is a single record in a DynamoDB table. It is comparable to a row in a relational database or a document in MongoDB.

2.1.3. Attributes

A DynamoDB item is made up of attributes, which are typed data values holding information about the element. For example, if you had an item representing a User, you might have an attribute named "Username" with a value of "alexdebrie".

Attributes are similar to column values on relational records, with the caveat that attributes are not required on every item like they are in a relational database.

When you write an item to DynamoDB, each attribute is given a specific type. There are ten different data types in DynamoDB. It's helpful to split them into three categories:

- **Scalars:** Scalars represent a single, simple value, such as a username (string) or an age (integer). There are five scalar types: string, number, binary, boolean, and null.
- **Complex:** Complex types are the most flexible kind of attribute, as they represent groupings with arbitrary nested attributes. There are two complex types: lists and maps. You can use complex attribute types to hold related elements. In the Big Time Deals example in Chapter 20, we use lists to hold an array of Featured Deals for the front page of our application. In the GitHub example in Chapter 21, we use a map on our Organization items to hold all information about the Payment Plan for the Organization.
- **Sets:** Sets are a powerful compound type that represents multiple, unique values. They are similar to sets in your favorite

programming language. Each element in a set must be the same type, and there are three set types: string sets, number sets, and binary sets. Sets are useful for tracking uniqueness in a particular domain. In the GitHub example in Chapter 21, we use a set to track the different reactions (e.g. heart, thumbs up, smiley face) that a User has attached to a particular issue or pull request.

The type of attribute affects which operations you can perform on that attribute in subsequent operations. For example, if you have a number attribute, you can use an update operation to add or subtract from the attribute. If you use a set attribute, you can check for the existence of a particular value before updating the item.

You will likely use scalars for most of your attributes, but the set and document types are very powerful. You can use sets to keep track of unique items, making it easy to track the number of distinct elements without needing to make multiple round trips to the database. Likewise, the document types are useful for several things, particularly when denormalizing the data in your table.

2.1.4. Primary keys

While DynamoDB is schemaless, it is not completely without structure. When creating a DynamoDB table, you must declare a primary key for your table. The primary key can be simple, consisting of a single value, or composite, consisting of two values. These two primary key types are discussed further in the next section.

Each item in your table must include the primary key. If you attempt to write an item without the primary key, it will be rejected. Further, each item in your table is uniquely identifiable by its primary key. If you attempt to write an item using a primary key that already exists, it will overwrite the existing item (unless you explicitly state that it shouldn't overwrite, in which case the write

will be rejected).

Primary key selection and design is the most important part of data modeling with DynamoDB. Almost all of your data access will be driven off primary keys, so you need to choose them wisely. We will discuss primary keys further in this chapter and in subsequent chapters.

2.1.5. Secondary indexes

Primary keys drive your key access patterns in DynamoDB, but sometimes you need additional flexibility. The way you configure your primary keys may allow for one read or write access pattern but may prevent you from handling a second access pattern.

To help with this problem, DynamoDB has the concept of secondary indexes. Secondary indexes allow you to reshape your data into another format for querying, so you can add additional access patterns to your data.

When you create a secondary index on your table, you specify the primary keys for your secondary index, just like when you're creating a table. AWS will copy all items from your main table into the secondary index in the reshaped form. You can then make queries against the secondary index.

Secondary indexes are a core tool in the tool belt of DynamoDB data modeling and something we'll cover extensively in this book.

The first four concepts can be seen in the following table containing some example users:

Primary key		Attributes	Attributes		Table
Partition key: Username					
alexdebrie	Primary key	FirstName	LastName	Birthdate	
		Alex	DeBrie	1988-05-26	
dayoneguy	Item	FirstName	LastName	Birthdate	
		Jeff	Bezos	1964-01-12	
oracleofomaha		FirstName	LastName	Birthdate	
		Warren	Buffett	1930-07-30	

We have three records in our example. All the records together are our *table* (outlined in red). An individual record is called an *item*. You can see the item for Jeff Bezos outlined in blue. Each item has a *primary key* of `Username`, which is outlined in green. Finally, there are other *attributes*, like `FirstName`, `LastName`, and `Birthdate`, which are outlined in black.

2.2. A Deeper Look: Primary keys and secondary indexes

Primary keys and secondary indexes are a detailed area, so we'll spend a little more time here. Both primary keys and secondary indexes are discussed in many of the subsequent chapters due to their centrality to proper DynamoDB data modeling.

In this section, we'll cover the two kinds of primary keys, the two kinds of secondary indexes, and the concept of projection in secondary indexes.

2.2.1. Types of primary keys

In DynamoDB, there are two kinds of primary keys:

- **Simple primary keys**, which consist of a single element called a partition key.
- **Composite primary keys**, which consist of two elements, called a partition key and a sort key.

You may occasionally see a partition key called a "hash key" and a sort key called a "range key". I'll stick with the "partition key" and "sort key" terminology in this book.

The type of primary key you choose will depend on your access patterns. A simple primary key allows you to fetch only a single item at a time. It works well for one-to-one operations where you are only operating on individual items.

Composite primary keys, on the other hand, enable a "fetch many" access pattern. With a composite primary key, you can use the Query API to grab all items with the same partition key. You can even specify conditions on the sort key to narrow down your query space. Composite primary keys are great for handling relations between items in your data and for retrieving multiple items at once.

2.2.2. Kinds of secondary indexes

When creating a secondary index, you will need to specify the key schema of your index. The key schema is similar to the primary key of your table—you will state the partition and sort key (if desired) for your secondary index that will drive your access patterns.

There are two kinds of secondary indexes in DynamoDB:

- Local secondary indexes
- Global secondary indexes

A local secondary index uses the same partition key as your table's primary key but a different sort key. This can be a nice fit when you are often filtering your data by the same top-level property but have access patterns to filter your dataset further. The partition key can act as the top-level property, and the different sort key arrangements will act as your more granular filters.

In contrast, with a global secondary index, you can choose any attributes you want for your partition key and your sort key. Global secondary indexes are used much more frequently with DynamoDB due to their flexibility.

There are a few other differences to note between local and global secondary indexes. For global secondary indexes, you need to provision additional throughput for the secondary index. The read and write throughput for the index is separate from the core table's throughput. This is not the case for local secondary indexes, which use the throughput from the core table.

Another difference between global and secondary indexes are in their consistency models. In distributed systems, a consistency model describes how data is presented as it is replicated across multiple nodes. In a nutshell, "strong consistency" means you will get the same answer from different nodes when querying them. In contrast, "eventual consistency" means you could get slightly different answers from different nodes as data is replicated.

> The notes on consistency above are a gross simplification. There are additional notes on DynamoDB's consistency model in the next chapter. If you want more detail on this topic, I recommend reading Designing Data-Intensive Systems by Martin Kleppmann.

With global secondary indexes, your only choice is eventual

consistency. Data is replicated from the core table to global secondary indexes in an asynchronous manner. This means it's possible that the data returned in your global secondary index does not reflect the latest writes in your main table. The delay in replication from the main table to the global secondary indexes isn't large, but it may be something you need to account for in your application.

On the other hand, local secondary indexes allow you to opt for strongly-consistent reads if you want it. Strongly-consistent reads on local secondary indexes consume more read throughput than eventually-consistent reads, but they can be beneficial if you have strict requirements around consistency.

In general, I opt for global secondary indexes. They're more flexible, you don't need to add them at table-creation time, and you can delete them if you need to. In the remainder of this book, you can assume all secondary indexes are global secondary indexes.

The differences between local and global secondary indexes are summarized in the table below.

	Key schema	Creation time	Consistency
Local secondary index	Must use same partition key as the base table	Must be created when table is created	Eventual consistency by default. Can choose to receive strongly-consistent reads at a cost of higher throughput usage
Global secondary index	May use any attribute from table as partition and sort keys	Can be created after the table exists	Eventual consistency only

Table 1. Secondary index types

2.3. The importance of item collections

One of the most important yet underdiscussed concepts in DynamoDB is the notion of *item collections*. An item collection refers to a group of items that share the same partition key in either the base table or a secondary index.

One example I'll use a few times in this book is a table that includes actors and actresses and the movies in which they've played roles. We could model this with a composite primary key where the partition key is `Actor` and the sort key is `Movie`.

The table with some example data looks as follows:

Primary key		Attributes		
Partition key: Actor	**Sort key: Movie**			
Item collection Tom Hanks	Cast Away	Role	Year	Genre
		Chuck Noland	2000	Drama
	Toy Story	Role	Year	Genre
		Woody	1995	Children's
Tim Allen	Toy Story	Role	Year	Genre
		Buzz Lightyear	1995	Children's
Item collection Natalie Portman	Black Swan	Role	Year	Genre
		Nina Sayers	2010	Drama

There are four movie roles in this table. Notice that two of those movie roles have the same partition key: `Tom Hanks`. Those two movie role items are said to be in the same item collection. Likewise, the single movie role for Natalie Portman is in an item collection, even though it only has one item in it.

Item collections are important for two reasons. First, they are useful

for partitioning. DynamoDB partitions your data across a number of nodes in a way that allows for consistent performance as you scale. However, all items with the same partition key will be kept on the same storage node. This is important for performance reasons. DynamoDB partitioning is discussed further in the next chapter.

Second, item collections are useful for data modeling. In Chapter 4, you will learn about the Query API action. The Query action can retrieve multiple items within a single item collection. It is an efficient yet flexible operation. A lot of data modeling tips will be focused on creating the proper item collections to handle your exact needs.

In the subsequent chapters, think about how you're working to build purpose-built item collections to satisfy your access patterns.

2.4. Conclusion

In this chapter, we discussed the core concepts in DynamoDB. We started off with the core vocabulary of tables, items, primary keys, attributes, and secondary indexes. Then, we took a deeper look at primary keys and secondary indexes as well as the notion of item collections.

You shouldn't be an expert in these topics yet, but they are the foundational building blocks of DynamoDB. Almost all of your data modeling will be focused on designing the right primary key and secondary indexes so that you're building the item collections to handle your needs.

Chapter 3. Advanced Concepts

> ## Chapter Summary
>
> This chapter covers advanced concepts in DynamoDB. While these concepts aren't strictly necessary, they are useful for getting the most out of DynamoDB.
>
> ## Sections
>
> - DynamoDB Streams
> - Time-to-live (TTL)
> - Partitions
> - Consistency
> - DynamoDB Limits
> - Overloading keys and indexes

To this point, we've covered some of the foundational elements of DynamoDB. We learned the core vocabulary, such as tables, items, primary keys, attributes, and secondary indexes, as well as the foundational concept of item collections. With this background, you should have a solid grounding in the basics of DynamoDB.

In this chapter, we're going to cover some advanced concepts in DynamoDB.

A few of these concepts, like DynamoDB streams and time-to-live (TTL) will allow you to handle more advanced use cases with DynamoDB. Other concepts, like partitions, consistency, and DynamoDB limits, will give you a better understanding of proper

data modeling with DynamoDB. Finally, the concept of overloaded keys and indexes is a data modeling concept that will be used frequently in your data modeling.

Let's get started.

3.1. DynamoDB Streams

DynamoDB streams are one of my favorite features of DynamoDB, and the inclusion of streams in DynamoDB reflects the microservice-friendly, event-driven environment in which DynamoDB was created.

The concept of streams and event streaming has exploded in popularity over the past decade with the rise of tools like Apache Kafta and Amazon Kinesis. Streams are an immutable sequence of records that can be processed by multiple, independent consumers. The combination of immutability plus multiple consumers has propelled the use of streams as a way to asynchronously share data across multiple systems.
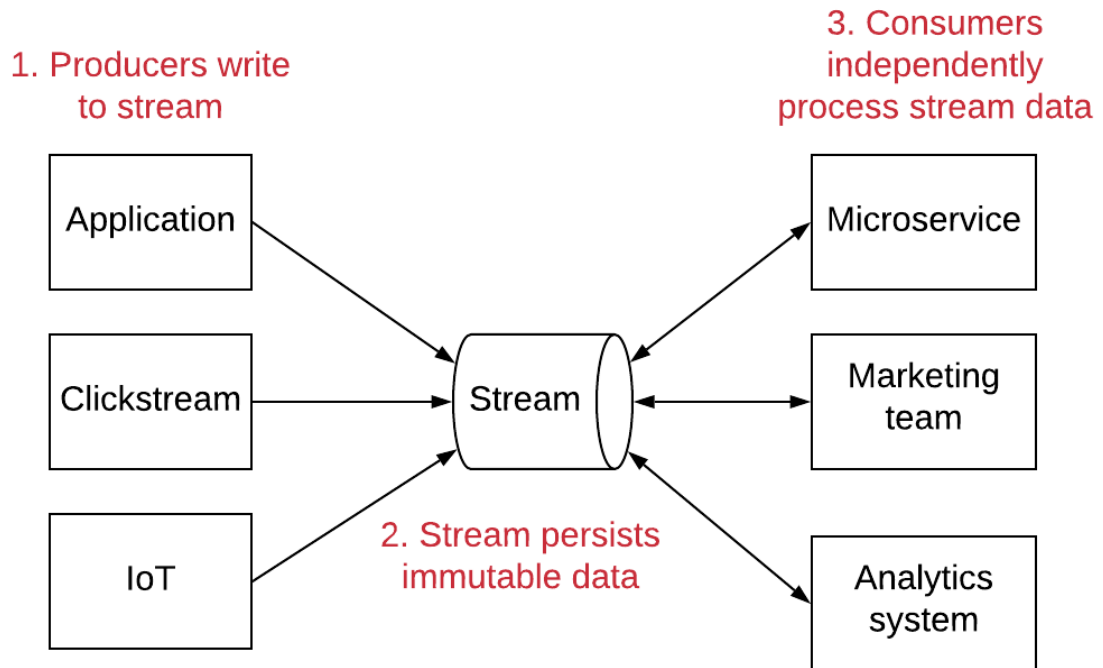
With DynamoDB streams, you can create a stream of data that includes a record of each change to an item in your table. Whenever an item is written, updated, or deleted, a record containing the details of that record will be written to your DynamoDB stream. You can then process this stream with AWS Lambda or other compute infrastructure.

1. DynamoDB sends write operations to stream

3. Consumers independently process stream data

2. DynamoDB Stream persists immutable data
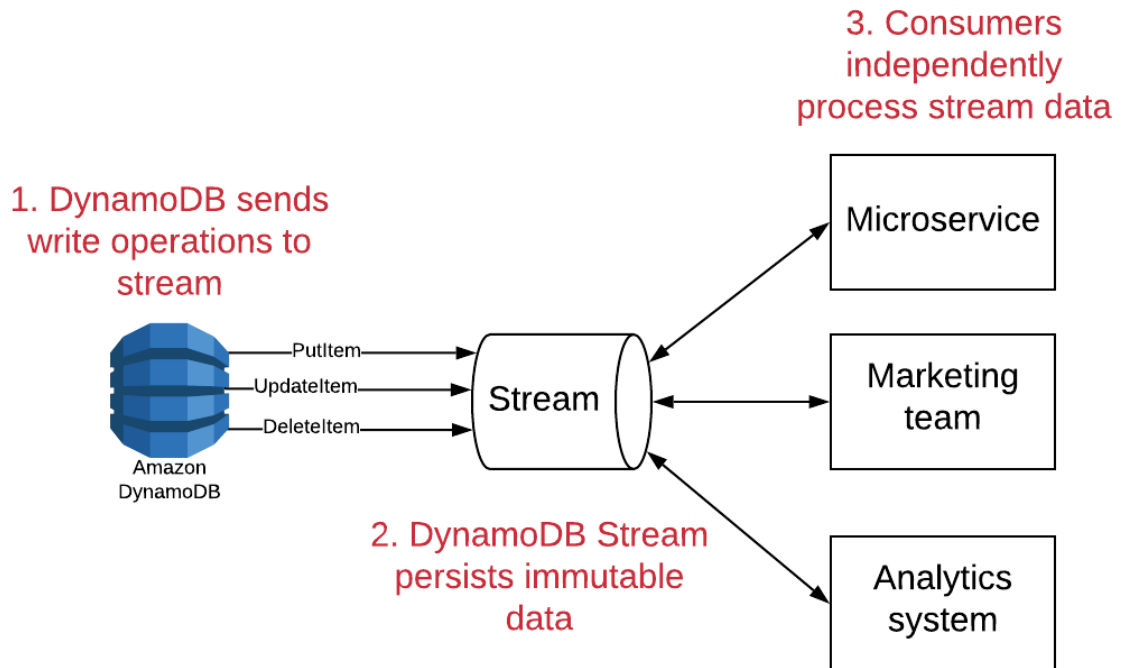
DynamoDB streams enable a variety of use cases, from using DynamoDB as a work queue to broadcasting event updates across microservices. The combination of DynamoDB Streams with serverless compute with AWS Lambda gives you a fully-managed system to react to database changes.

3.2. Time-to-live (TTL)

TTLs are a more recent feature addition to DynamoDB. Announced in 2017, TTLs allow you to have DynamoDB automatically delete items on a per-item basis. This is a great option for storing short-term data in DynamoDB as you can use TTL to clean up your database rather than handling it manually via a scheduled job.

To use TTL, you specify an attribute on your DynamoDB table that will serve as the marker for item deletion. For each item that you want to expire, you should store a Unix timestamp as a number in

your specified attribute. This timestamp should state the time after which the item should be deleted. DynamoDB will periodically review your table and delete items that have your TTL attribute set to a time before the current time.

One nice feature about TTLs is that you don't need to use it for all items in your table. For items that you don't want to automatically expire, you can simply not set the TTL attribute on the item. This can be useful when you are storing items with different mechanics in the table. Imagine an access keys table where user-generated tokens are active until intentionally deactivated, while machine-generated tokens are expired after ten minutes.

A final note on TTLs: your application should be safe around how it handles items with TTLs. Items are generally deleted in a timely manner, but AWS only states that items will *usually* be deleted within 48 hours after the time indicated by the attribute. This delay could be unacceptable for the access patterns in your application. Rather than relying on the TTL for data accuracy in your application, you should confirm an item is not expired when you retrieve it from DynamoDB. You can see an example of this in action in Chapter 18, where we implement a session store.

3.3. Partitions

One of the more hidden topics in DynamoDB is the topic of partitions. You don't really need to know about partitions to start playing with data in DynamoDB, but having an understanding of how partitions work will give you a better mental model of how to think about data in DynamoDB.

Partitions are the core storage units underlying your DynamoDB table. We noted before that DynamoDB is built for infinite scale,

and it does that by sharding your data across multiple server instances.

When a request comes into DynamoDB, the request router looks at the partition key in the request and applies a hash function to it. The result of that hash function indicates the server where that data will be stored, and the request is forwarded to that server to read or write the data as requested. The beauty of this design is in how it scales—DynamoDB can add additional storage nodes infinitely as your data scales up.

In earlier versions of DynamoDB, you needed to be more aware of partitions. Previously, the total throughput on a table was shared evenly across all partitions. You could run into issues when unbalanced access meant you were getting throttled without using your full throughput. You could also run into an issue called throughput dilution if you temporarily scaled the throughput very high on your table, such as if you were running a bulk import job, then scaled the throughput back down.

All of this is less of a concern now as the DynamoDB team has added a concept called *adaptive capacity*. With adaptive capacity, throughput is automatically spread around your table to the items that need it. There's no more uneven throughput distribution and no more throughput dilution.

While you don't need to think much about partitions in terms of capacity, partitions are the reason you need to understand item collections, which we discussed in the last chapter. Each item collection will be in a particular partition, and this will enable fast queries on multiple items.

There are two limits around partitions to keep in mind. One is around the maximum throughput for any given partition, and the other is around the item collection size limit when using local secondary indexes. Those limits are discussed in section 3.5 of this

chapter.

3.4. Consistency

When talking about databases and distributed systems, one of the properties you need to consider is its *consistency* mechanics.

At a general level, consistency refers to whether a particular read operation receives all write operations that have occurred prior to the read. To understand why that might not happen, let's do a quick primer on how DynamoDB handles data.

As we just read in the previous section, DynamoDB splits up, or "shards", its data by splitting it across multiple partitions. This allows DynamoDB to horizontally scale by adding more storage nodes. This horizontal scaling is how it can provide fast, consistent performance no matter your data size.

To handle this, there are vast numbers of storage partitions spread out across a giant fleet of virtual machines. When you write data to DynamoDB, there is a request router that is the frontend for all requests. It will authenticate your request to ensure you have access to write to the table. If so, it will hash the partition key of your item and send that key to the proper primary node for that item.
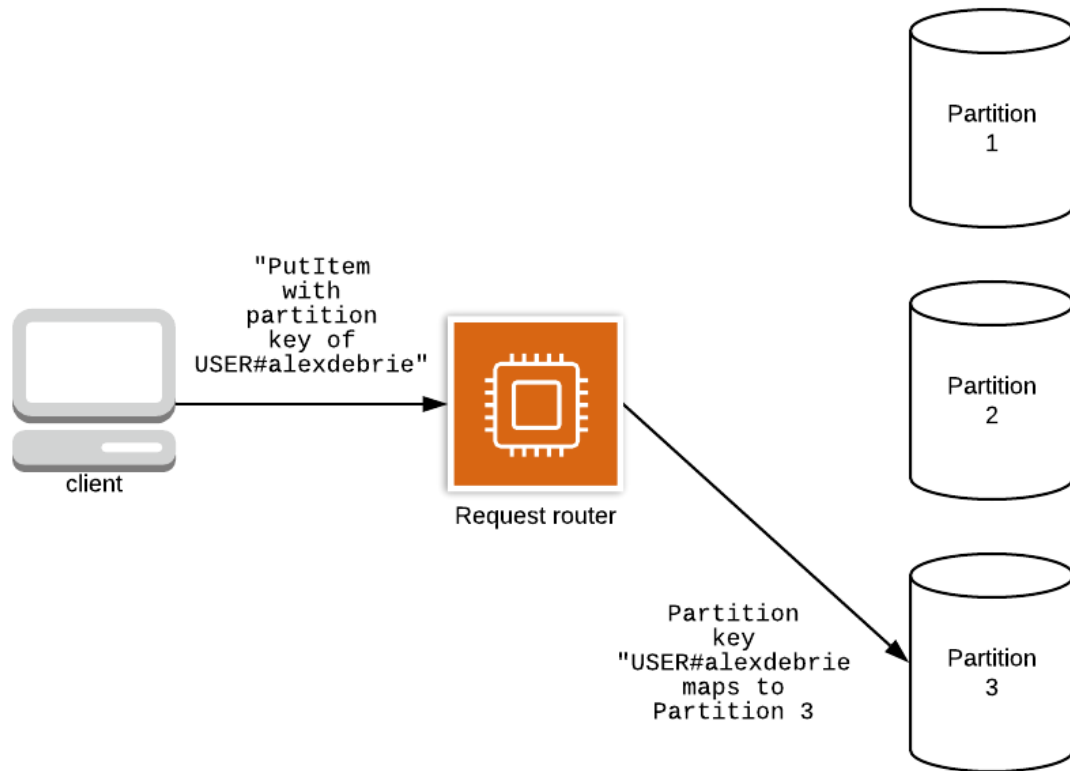
The primary node for a partition holds the canonical, correct data for the items in that node. When a write request comes in, the primary node will commit the write and commit the write to one of two secondary nodes for the partition. This ensures the write is saved in the event of a loss of a single node.
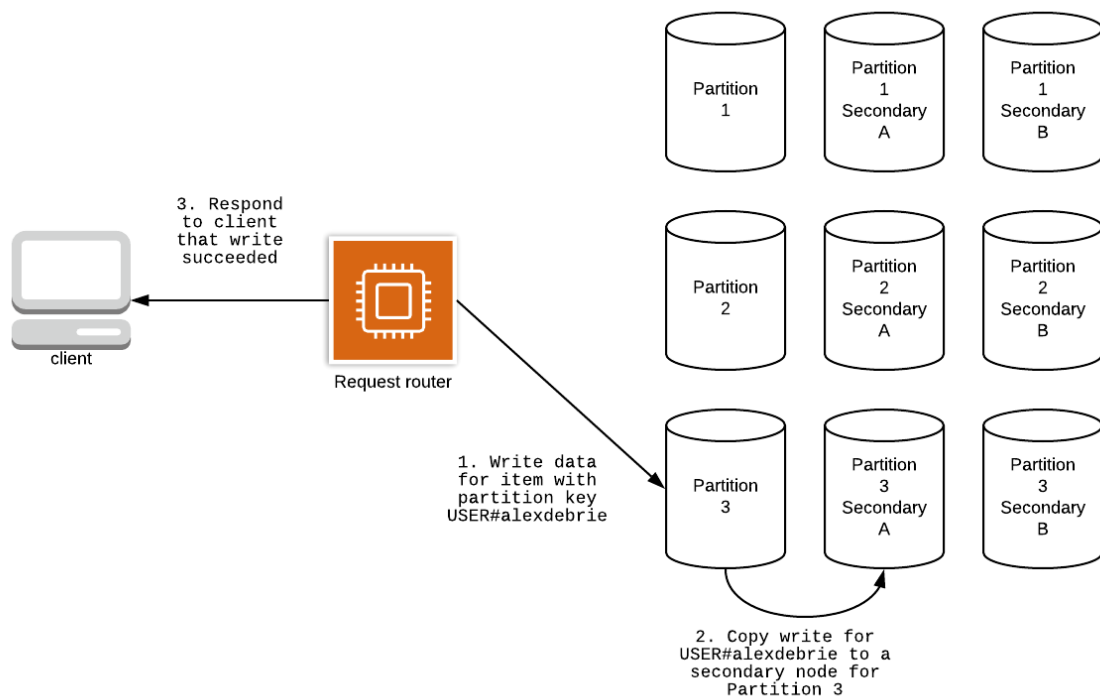
After the primary node responds to the client to indicate that the write was successful, it then asynchronously replicates the write to a third storage node.
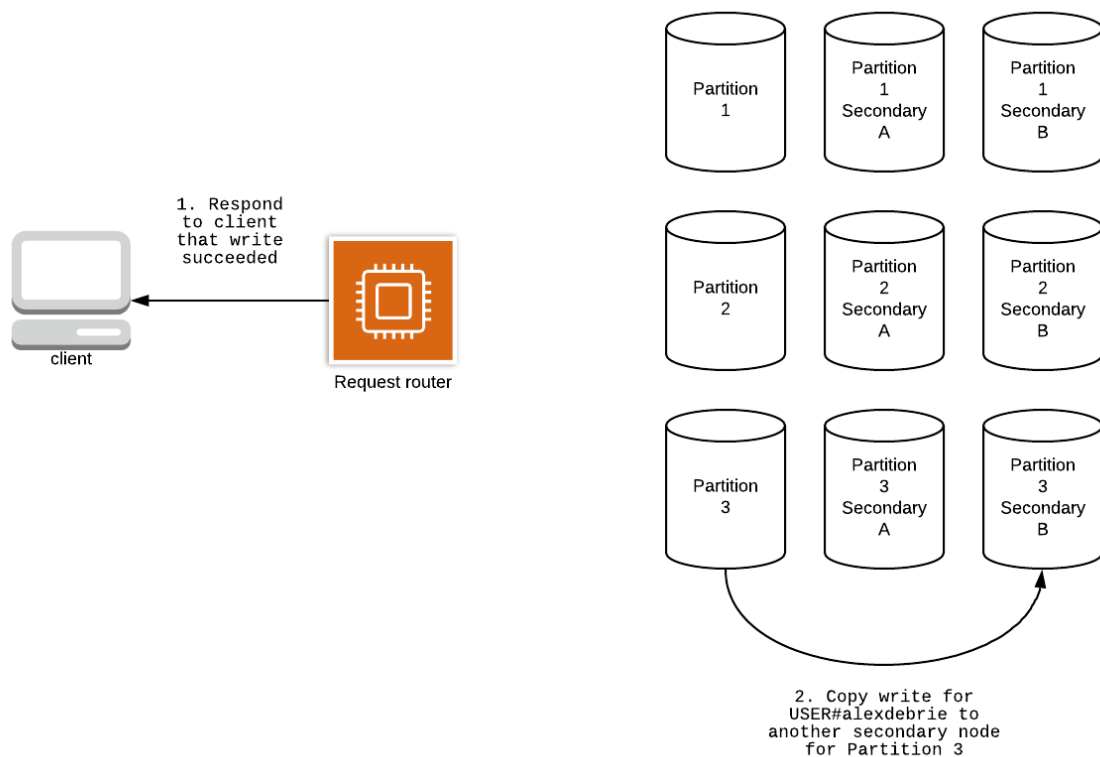
Thus, there are three nodes—one primary and two secondary—for

each partition. These secondary nodes serve a few purposes. First, they provide fault-tolerance in case the primary node goes down. Because that data is stored on two other nodes, DynamoDB can handle a failure of one node without data loss.

Secondly, these secondary nodes *can serve read requests* to alleviate pressure on the primary node. Rather than requiring that all reads and writes go through the primary, we can have all writes go through the primary and then have the reads shared across the three nodes.
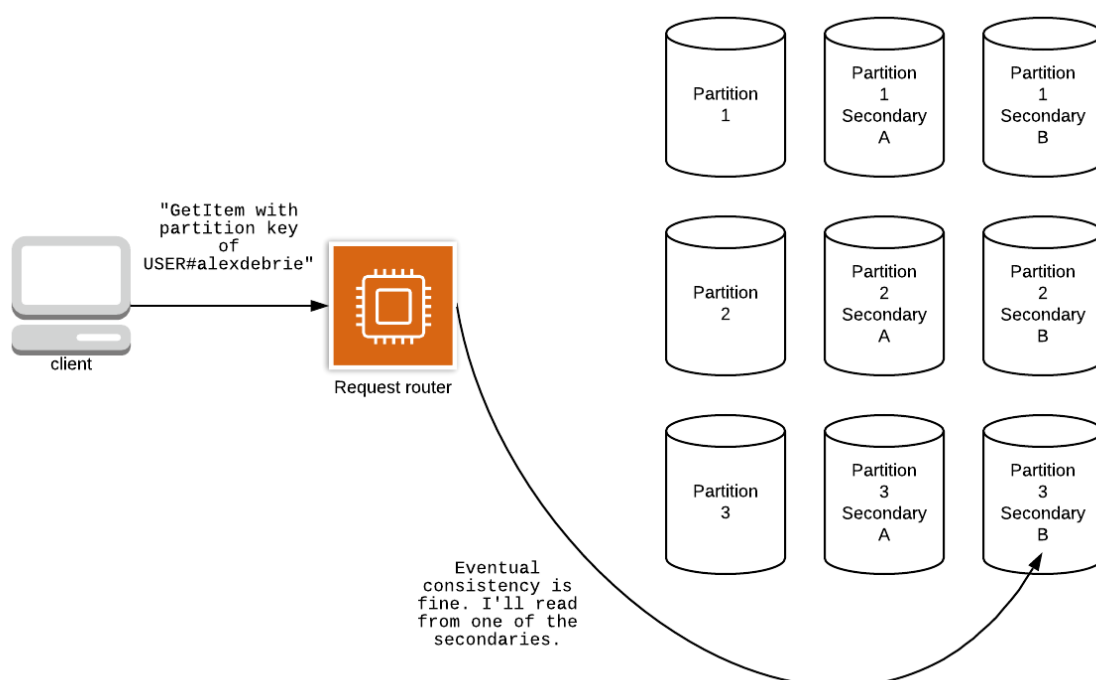
However, notice that there is a potential issue here. Because writes are asynchronously replicated from the primary to secondary nodes, the secondary might be a little behind the primary node. And because you can read from the secondary nodes, it's possible you could read a value from a secondary node that does not reflect the latest value written to the primary.

With that in mind, let's look at the two consistency options available with DynamoDB:

- **Strong consistency**

- **Eventual consistency**

With strong consistency, any item you read from DynamoDB will reflect all writes that occurred prior to the read being executed. In contrast, with eventual consistency, it's possible the item(s) you read will not reflect all prior writes.

Finally, there are two times you need to think about consistency with DynamoDB.

First, whenever you are reading data from your base table, you can choose your consistency level. By default, DynamoDB will make an eventually-consistent read, meaning that your read may go to a secondary node and may show slightly stale data. However, you can opt into a strongly-consistent read by passing `ConsistentRead=True` in your API call. An eventually-consistent read consumes half the write capacity of a strongly-consistent read and is a good choice for many applications.

Second, you should think about consistency when choosing your secondary index type. A local secondary index will allow you to make strongly-consistent reads against it, just like the underlying table. However, a global secondary index will only allow you to make eventually-consistent reads. If you do choose a local secondary index, the mechanics are the same as with your base table—you can opt in to strongly consistent reads by setting `ConsistentRead=True`.

3.5. DynamoDB Limits

In this last section, I want to discuss a few DynamoDB limits. There are a number of limits in DynamoDB, from the characters you can

use in a table's name to the maximum length of a partition key. Most of these are minutia that we won't cover here. I want to cover a few high-salience limits that may affect how you model your data in DynamoDB.

3.5.1. Item size limits

A single DynamoDB item is limited to 400KB of data. While that may seem like a lot, it's a surprise to folks coming from NoSQL systems like MongoDB or Cassandra where the record sizes are much larger (16MB for MongoDB, and a whopping 2GB in Cassandra!)

Like everything in DynamoDB, these size limits are intentional. They push you away from anti-patterns and toward proper data modeling. Large item sizes mean larger reads from disk, resulting in slower response times and fewer concurrent requests. You should break down larger items into smaller items and do more targeted operations.

This limit will affect you most commonly as you denormalize your data. When you have a one-to-many relationship, you may be tempted to store all the related items on the parent item rather than splitting this out. This works for many situations but can blow up if you have an unbounded number of related items.

3.5.2. Query and Scan Request Size Limits

The second limit is about the maximum result set for the `Query` and `Scan` operations, the two "fetch many" API actions in DynamoDB. `Query` and `Scan` will read a maximum of 1MB of data from your table. Further, this 1MB limit is applied *before* any filter expressions are considered.

This 1MB limit is crucial to keeping DynamoDB's promise of consistent single-digit response times. If you have a request that will address more than 1MB of data, you will need to paginate through the results by making follow-up requests to DynamoDB.

In the next chapter, we'll see why the 1MB limit is crucial for DynamoDB to ensure you don't write a query that won't scale.

3.5.3. Partition throughput limits

The third important limit you should understand is the throughput limit around a single partition. A single partition can have a maximum of 3000 Read Capacity Units or 1000 Write Capacity Units. Remember, capacity units are on a per-second basis, and these limits apply to a single partition, not the table as a whole. Thus, you will need to be doing 3000 reads per second *for a given partition key* to hit these limits.

This is pretty high traffic volume, and not many users will hit it, though it's definitely possible. If this is something your application could hit, you'll need to look into read or write sharding your data.

3.5.4. Item collection limits

The last limit you should know about involves local secondary indexes and item collections. An item collection refers to all items with a given partition key, both in your main table and any local secondary indexes. If you have a local secondary index, a single item collection cannot be larger than 10GB. If you have a data model that has many items with the same partition key, this could bite you at a bad time because your writes will suddenly get rejected once you run out of partition space.

The partition size limit is not a problem for global secondary

indexes. If the items in a global secondary index for a partition key exceed 10 GB in total storage, they will be split across different partitions under the hood. This will happen transparently to you—one of the significant benefits of a fully-managed database.

3.6. Overloading keys and indexes

The previous five concepts have been features or characteristics of the DynamoDB service itself. This last concept is a data modeling concept that you'll use over and over in your application.

We saw in the last chapter about the importance of your primary key, both on the base table and on each of the secondary indexes. The primary key will be used to efficiently handle your access patterns.

In the examples we've shown thus far, like the Users table or the Movie Roles table in the previous chapter, we've had pretty simple examples. Our tables had just a single type of entity, and the primary key patterns were straightforward.

One unique quirk of modeling with DynamoDB is that you will often include different types of entities in a single table. The reasoning for this will be explained later in the book (see Chapters 7 & 8 for most of the background). For now, it's enough to know that including multiple entity types in one table will make for more efficient queries.

For an example of what this looks like, imagine you had a SaaS application. Organizations signed up for your application, and each Organization had multiple Users that belonged to the Organization. Let's start with a table that just has our Organization items in it:

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
ORG#BERKSHIRE	ORG#BERKSHIRE	OrgName	SubscriptionLevel	GSI1PK	GSI1SK
		Berkshire Hathaway	Enterprise	ORGANIZATIONS	Berkshire Hathaway
ORG#FACEBOOK	ORG#FACEBOOK	OrgName	SubscriptionLevel	GSI1PK	GSI1SK
		Facebook	Pro	ORGANIZATIONS	Facebook

In the image above, we have two Organization items—one for Berkshire Hathaway and one for Facebook. There are two things worth noting here.

First, notice how generic the names of the partition key and sort key are. Rather than having the partition key named 'OrgName`, the partition key is titled PK, and the sort key is SK. That's because we will also be putting User items into this table, and Users don't have an OrgName. They have a UserName.

Second, notice that the PK and SK values have prefixes. The pattern for both is ORG#<OrgName>. We do this for a few reasons. First, it helps to identify the type of item that we're looking at. Second, it helps avoid overlap between different item types in a table. Remember that a primary key must be unique across all items in a table. If we didn't have this prefix, we could run into accidental overwrites. Imagine if the real estate company Keller Williams signed up for our application, and the musician Keller Williams was a user of our application. The two could overwrite each other!

Let's edit our table to add Users now. A table with both Organization and User entities might look as follows:

Primary key		Attributes	
Partition key: PK	Sort key: SK		
ORG#BERKSHIRE	ORG#BERKSHIRE	OrgName	SubscriptionLevel
		Berkshire Hathaway	Enterprise
	USER#CHARLIEMUNGER	UserName	Role
		Charlie Munger	Member
	USER#WARRENBUFFETT	UserName	Role
		Warren Buffett	Admin
ORG#FACEBOOK	ORG#FACEBOOK	OrgName	SubscriptionLevel
		Facebook	Pro
	USER#SHERYLSANDBERG	UserName	Role
		Sheryl Sandberg	Admin

Here we've added three Users to our existing Organization items. Our User items use a `PK` value of `ORG#<OrgName>` and an `SK` value of `USER#<UserName>`.

This concept of using generic names for your primary keys and using different values depending on the type of item is known as *overloading* your keys. You will do this with both your primary keys and your secondary indexes to enable the access patterns you need.

If you feel confused, that's OK. In the coming chapters, we'll slowly build up to how and why you want to use key overloading to handle your access patterns. I wanted to introduce the idea here so that you're not confused if you see `PK` and `SK` examples in subsequent chapters.

3.7. Conclusion

In this chapter, we reviewed some more advanced concepts in DynamoDB.

First, we reviewed two additional features of DynamoDB. DynamoDB Streams give you native change data capture support in DynamoDB and enable powerful fan-out data patterns. And the TTL feature makes it easy to expire items out of your table as they are no longer needed.

Second, we reviewed some underlying concepts about DynamoDB. We learned about the concept of partitions, which is useful for building the proper mental model of how DynamoDB works. Then we learned about consistency and the two consistency modes that DynamoDB offers. Next, we learned about a few DynamoDB limits you should know.

Finally, we learned the data modeling concept of overloading your keys and indexes. Primary key overloading is a crucial concept for successfully using DynamoDB, but it's not intuitive at first. We introduced the concept in this chapter, but it's something that will come up at multiple points in subsequent chapters.

Chapter 4. The Three API Action Types

> ## Chapter Summary
>
> This chapter covers the types of API actions in DynamoDB.
> You will learn about the three types of API actions and when
> they are useful.
>
> ## Sections
>
> 1. Background on the DynamoDB API
> 2. Item-based actions
> 3. Queries
> 4. Scans
> 5. How DynamoDB enforces efficient data access

Reading and writing from DynamoDB will feel different than
interacting with a relational database. With a relational database,
you're often writing a text string using SQL (Structured Query
Language), which is parsed by the relational database server to
perform your requested action.

```sql
SELECT *
FROM users
WHERE users.username = 'alexdebrie';
```

In contrast, you usually interact with DynamoDB by using the AWS SDK or a third-party library in your programming language of choice. These SDKs expose a few API methods to write to and read from your DynamoDB table.

```python
import boto3

client = boto3.client('dynamodb')

user = client.get_item(
    TableName='AppTable',
    Key={
        'PK': { 'S': 'USER#alexdebrie' },
        'SK': { 'S': 'USER#alexdebrie' }
    }
)
```

In this chapter, we'll learn about the core API actions with DynamoDB. The API for DynamoDB is small but powerful. This

makes it easy to learn the core actions while still providing a flexible way to model and interact with your data.

I like to split the API actions into three categories:

1. Item-based actions
2. Queries
3. Scans

The API actions are divided based on what you're operating on.

Operating on specific items? Use the item-based actions.

Operating on an item collection? Use a Query.

Operating on the whole table? Use a Scan.

In the sections below, we'll walk through the details of these three categories.

4.1. Item-based actions

Item-based actions are used whenever you are operating on a specific item in your DynamoDB table. There are four core API actions for item-based actions:

1. **GetItem**--used for reading a single item from a table.
2. **PutItem**--used for writing an item to a table. This can completely overwrite an existing item with the same key, if any.
3. **UpdateItem**--used for updating an item in a table. This can create a new item if it doesn't previously exist, or it can add, remove, or alter properties on an existing item.
4. **DeleteItem**--used for deleting an item from a table.

There are three rules around item-based actions. First, the full primary key must be specified in your request. Second all actions to alter data—writes, updates, or deletes—must use an item-based action. Finally, all item-based actions must be performed on your main table, not a secondary index.

> ⛔ Single-item actions must include the entire primary key of the item(s) being referenced.

The combination of the first two rules can be surprising—you can't make a write operation to DynamoDB that says, "Update the attribute X for all items with a partition key of Y" (assuming a composite primary key). You would need to specify the full key of *each* of the items you'd like to update.

In addition to the core single-item actions above, there are two sub-categories of single-item API actions—batch actions and transaction actions. These categories are used for reading and writing multiple DynamoDB items in a single request. While these operate on multiple items at once, I still classify them as item-based actions because you must specify the exact items on which you want to operate. The separate requests are split up and processed once they hit the DynamoDB router, and the batch requests simply save you from making multiple trips.

There is a subtle difference between the batch API actions and the transactional API actions. In a batch API request, your reads or writes can succeed or fail independently. The failure of one write won't affect the other writes in the batch.

With the transactional API actions, on the other hand, all of your reads or writes will succeed or fail together. The failure of a single write in your transaction will cause the other writes to be rolled back.

4.2. Query

The second category of API actions is the Query API action. The Query API action lets you retrieve multiple items with the same partition key. This is a powerful operation, particularly when modeling and retrieving data that includes relations. You can use the Query API to easily fetch all related objects in a one-to-many relationship or a many-to-many relationship.

We'll show how the Query operation can be useful by way of an example. Imagine you modeled a table that tracked actors and actresses and the movies in which they acted. Your table might have some items like the following:

Primary key		Attributes		
Partition key: Actor	Sort key: Movie			
Tom Hanks	Cast Away	Role	Year	Genre
		Chuck Noland	2000	Drama
	Toy Story	Role	Year	Genre
		Woody	1995	Children's
Tim Allen	Toy Story	Role	Year	Genre
		Buzz Lightyear	1995	Children's
Natalie Portman	Black Swan	Role	Year	Genre
		Nina Sayers	2010	Drama

Here we have four items in our table. The primary key is a composite primary key using the actor's name as the Partition key and the Movie name as the Sort key. We also store additional attributes on our items, such as the performer's name in the movie, the year the movie was released, and the genre of the movie.

You can use the Query operation on either your base table or a

secondary index. When making a Query, you must include a partition key in your request. In our example, this can be useful to find all the roles an actor has played.

For example, we could use the Query operation and specify a partition key of "Tom Hanks". The code to do so would look as follows:

```
items = client.query(
    TableName='MoviesAndActors',
    KeyConditionExpression='#actor = :actor',
    ExpressionAttributeNames={
        '#actor': 'Actor'
    },
    ExpressionAttributeValues={
        ':actor': { 'S': 'Tom Hanks' }
    }
)
```

Put aside the funky syntax in the key condition expression for now—we cover that in more detail in the following chapters.

This Query would return two items—Tom Hanks in Cast Away and Tom Hanks in Toy Story.

Primary key		Attributes		
Partition key: Actor	Sort key: Movie			
Tom Hanks	Cast Away	Role	Year	Genre
		Chuck Noland	2000	Drama
	Toy Story	Role	Year	Genre
		Woody	1995	Children's
Tim Allen	Toy Story	Role	Year	Genre
		Buzz Lightyear	1995	Children's
Natalie Portman	Black Swan	Role	Year	Genre
		Nina Sayers	2010	Drama

Remember that all items with the same partition key are in the same item collection. Thus, the Query operation is how you efficiently read items in an item collection. This is why you carefully structure your item collections to handle your access patterns. This will be a major theme of the subsequent chapters.

While the partition key is required, you may also choose to specify conditions on the sort key in a Query operation. In our example, imagine we want to get all of Tom Hanks' roles in movies where the title is between A and M in the alphabet. We could use the following Query action:

```
items = client.query(
    TableName='MoviesAndActors',
    KeyConditionExpression='#actor = :actor AND #movie BETWEEN :a AND :m',
    ExpressionAttributeNames={
        '#actor': 'Actor',
        '#movie': 'Movie'
    },
    ExpressionAttributeValues={
        ':actor': { 'S': 'Tom Hanks' },
        ':a': { 'S': 'A' },
        ':m': { 'S': 'M' }
    }
)
```

Which would fetch the following result on our DynamoDB table:

Primary key		Attributes		
Partition key: Actor	**Sort key: Movie**			
Tom Hanks	Cast Away	Role	Year	Genre
		Chuck Noland	2000	Drama
	Toy Story	Role	Year	Genre
		Woody	1995	Children's
Tim Allen	Toy Story	Role	Year	Genre
		Buzz Lightyear	1995	Children's
Natalie Portman	Black Swan	Role	Year	Genre
		Nina Sayers	2010	Drama

This would return a single item—Tom Hanks in Cast Away—as it is the only item that satisfies both the partition key requirement and the sort key requirement.

As mentioned, you can use the Query API on either the main table or a secondary index. While we're here, let's see how to use the Query operation on a secondary index.

With our example, we can query movie roles by the actor's name. But what if we want to query by a movie? Our current pattern doesn't allow this, as the partition key must be included in every request.

To handle this pattern, we can create a global secondary index that flips our partition key and sort key:

Primary key		Attributes		
Partition key: Actor	**Sort key: Movie**			
Tom Hanks	Cast Away	Role	Year	Genre
		Chuck Noland	2000	Drama
	Toy Story	Role	Year	Genre
		Woody	1995	Children's
Tim Allen	Toy Story	Role	Year	Genre
		Buzz Lightyear	1995	Children's
Natalie Portman	Black Swan	Role	Year	Genre
		Nina Sayers	2010	Drama

Our secondary index will look like this:

Primary key		Attributes		
Partition key: Movie	**Sort key: Actor**			
Cast Away	Tom Hanks	Role	Year	Genre
		Chuck Noland	2000	Drama
Toy Story	Tim Allen	Role	Year	Genre
		Buzz Lightyear	1995	Children's
	Tom Hanks	Role	Year	Genre
		Woody	1995	Children's
Black Swan	Natalie Portman	Role	Year	Genre
		Nina Sayers	2010	Drama

Notice that we have the same four items as in our previous table. The primary key has changed, but the data is the same.

Now, we can get all performers in Toy Story by using a Query operation on the secondary index and specifying "Toy Story" as the partition key:

```
items = client.query(
    TableName='MoviesAndActors',
    IndexName='MoviesIndex'
    KeyConditionExpression='#movie = :movie',
    ExpressionAttributeNames={
        '#movie': 'Movie'
    },
    ExpressionAttributeValues={
        ':movie': { 'S': 'Toy Story' }
    }
)
```

Which will hit the following partition in our table:

Primary key		Attributes		
Partition key: Movie	Sort key: Actor			
Cast Away	Tom Hanks	Role	Year	Genre
		Chuck Noland	2000	Drama
Toy Story	Tim Allen	Role	Year	Genre
		Buzz Lightyear	1995	Children's
	Tom Hanks	Role	Year	Genre
		Woody	1995	Children's
Black Swan	Natalie Portman	Role	Year	Genre
		Nina Sayers	2010	Drama

We received two items back—Tom Hanks and Tim Allen—representing the Toy Story roles in our database.

You'll use the Query operation quite heavily with DynamoDB. It's an efficient way to return a large number of items in a single request. Further, the conditions on the sort key can provide powerful filter capabilities on your table. We'll learn more about conditions later in this chapter.

4.3. Scan

The final kind of API action is the Scan. The Scan API is the bluntest tool in the DynamoDB toolbox. By way of analogy, item-based actions are like a pair of tweezers, deftly operating on the exact item you want. The Query call is like a shovel—grabbing a larger amount of items but still small enough to avoid grabbing everything.

The Scan operation is like a payloader, grabbing everything in its path.

A Scan will grab everything in a table. If you have a large table, this will be infeasible in a single request, so it will paginate. Your first request in a Scan call will read a bunch of data and send it back to you, along with a pagination key. You'll need to make another call, using the pagination key to indicate to DynamoDB where you left off.

There are a few special occasions when it's appropriate to use the Scan operation, but you should seldom use it during a latency-sensitive job, such as an HTTP request in your web application.

The times you may consider using the Scan operation are:

- When you have a very small table;
- When you're exporting all data from your table to a different system;
- In exceptional situations, where you have specifically modeled a sparse secondary index in a way that expects a scan.

The first two should be pretty straightforward. If you don't understand the last, there is a description of this pattern in Chapter 19.

In sum, don't use Scans.

4.4. How DynamoDB enforces efficiency

The DynamoDB API may seem limited, but it's very intentional. The key point to understand about DynamoDB is that it won't let you write a bad query. And by 'bad query', I mean a query that will degrade in performance as it scales.

In the last chapter, we discussed the concept of *partitions* in DynamoDB. DynamoDB uses partitions, or small storage nodes of about 10GB, to shard your data across multiple machines. The sharding is done on the basis of the *partition key*. Thus, if the DynamoDB request router is given the partition key for an item, it can do an `O(1)` lookup in a hash table to find the exact node or set of nodes where that item resides.
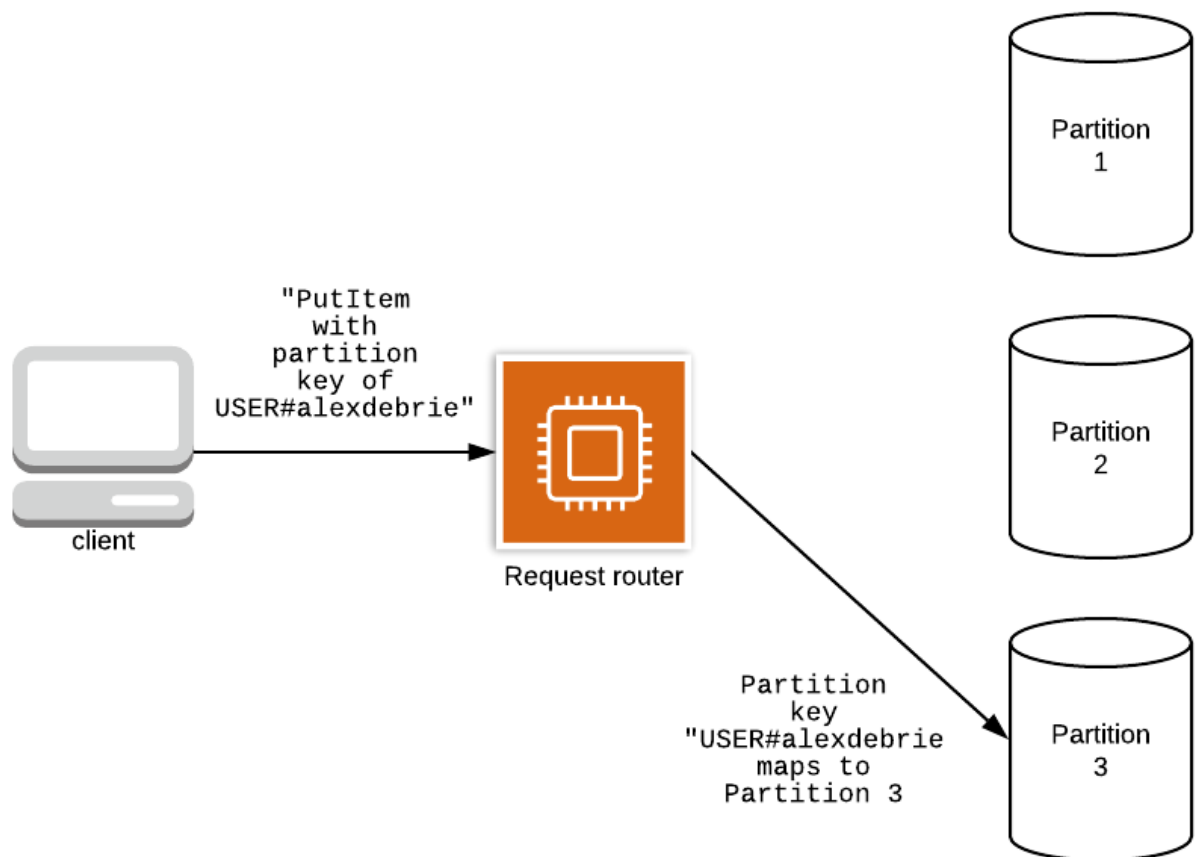
This is why all the single-item actions and the Query action require a partition key. No matter how large your table becomes, including the partition key makes it a constant time operation to find the item or item collection that you want.

All the single-item actions also require the sort key (if using a composite primary key) so that the single-item actions are constant time for the entire operation. But the Query action is different. The Query action fetches multiple items. So how does the Query action stay efficient?

Note that the Query action only allows you to fetch a contiguous block of items within a particular item collection. You can do operations like >=, <=, `begins_with()`, or `between`, but you can't do `contains()` or `ends_with()`. This is because an item collection is ordered and stored as a B-tree. Remember that a B-tree is like a phone book or a dictionary. If you go to a dictionary, it's trivial to

find all words between "hippopotamus" and "igloo". It's much harder to find all words that end in "-ing".

The time complexity of a B-tree search is `O(log n)`. This isn't as fast as our constant-time `O(1)` lookup for finding the item collection, but it's still pretty quick to grab a batch of items from a collection. Further, the size of `n` is limited. We're not doing an `O(log n)` search over our entire 10TB dataset. We're doing it on a single item collection, which is likely a few GB at most.

Finally, just to really put a cap on how slow an operation can be, DynamoDB limits all Query and Scan operations to 1MB of data in total. Thus, even if you have an item collection with thousands of items and you're trying to fetch the entire thing, you'll still be bounded in how slow an individual request can be. If you want to fetch all those items, you'll need to make multiple, serial requests to DynamoDB to page through the data. Because this is explicit—you'll need to write code that uses the `LastEvaluatedKey` parameter—it is much more apparent to you when you're writing an access pattern that won't scale.

In shorter form, let's review those steps and the time complexity of each one again:

Operation	Data structure	Notes
1. Find node for partition key	Hash table	Time complexity of `O(1)`
2. Find starting value for sort key	B-tree	Time complexity of `O(log n)`, where `n` is the size of the item collection, not the entire table
3. Read values until end of sort key match	N/A	Sequential read. Limited to 1MB of data

Table 2. DynamoDB operations time complexity

4.5. Conclusion

In this chapter, we learned about the types of API actions in DynamoDB and how DynamoDB enforces efficiency in its data access.

I mentioned at the beginning that we should split the DynamoDB API actions into three buckets. In the first bucket are single-item actions that operate on specific items in DynamoDB. These are all of the write-based actions, as well as some batch-based and transactional APIs.

In the second category is the Query API action. The Query action operates on a single *item collection* in a DynamoDB table. It allows you to fetch multiple items in a single request and will be crucial for handling relationships and advanced access patterns.

Finally, the third category contains the Scan API action. In general, you should avoid the Scan API as it operates on your entire table. However, there are some useful patterns that use Scans, and we'll see those in upcoming chapters.

Understanding these groupings of actions will be useful as you think about efficient DynamoDB data access. Further, we can understand how the creators of DynamoDB have been very intentional about evolving their API. The top priority is focusing on the long-term scalability of any access patterns. They won't implement features that seem nice at low volume but fall apart at scale.

Now that we know these broad ideas about the DynamoDB API, we'll look at some more practical advice on the DynamoDB API in the next two chapters.

Chapter 5. Using the DynamoDB API

Chapter Summary

This chapter looks at the mechanics of using the DynamoDB API. You learn some high-level tips for interacting with DynamoDB in your application code.

Sections

1. Learn how expression names and values work.

2. Don't use an ORM.

3. Understand the optional properties on individual requests

In the previous chapter, we learned the basics of the DynamoDB API. In this chapter, I'll give a few of my tips for working with the DynamoDB API.

I don't spend a ton of time teaching the DynamoDB API, because it's relatively straight-forward. The biggest hurdle in learning DynamoDB is the mindset shift required in learning the data modeling principles. Once you get the principles down and have written out your access patterns for a particular model, translating those patterns into the proper API calls is usually much easier.

That said, it is helpful to know a few of the tricky bits and hidden areas of the DynamoDB API. We'll focus on those here.

5.1. Learn how expression names and values work

The first tip around using the DynamoDB API is that you should spend the time necessary to understand how expression attribute names and values work. Expressions are necessary to efficiently query data in DynamoDB. The mechanics around expressions are covered in the next chapter, but we'll do some introductory work on expression names and values here.

Let's explore this by way of an example. Think back to when we were looking at the Query API action in the previous chapter. We were querying against our Movie Roles table. Here's the example Query for returning Tom Hanks' roles in movies whose title was between A and M:

```python
items = client.query(
    TableName='MoviesAndActors',
    KeyConditionExpression='#actor = :actor AND #movie BETWEEN :a AND :m',
    ExpressionAttributeNames={
        '#actor': 'Actor',
        '#movie': 'Movie'
    },
    ExpressionAttributeValues={
        ':actor': { 'S': 'Tom Hanks' },
        ':a': { 'S': 'A' },
        ':m': { 'S': 'M' }
    }
)
```

Look at the `KeyConditionExpression` property. It looks like hieroglyphics. Let's break down what's happening:

```
KeyConditionExpression: '#actor = :actor AND #movie BETWEEN :a AND :m'
```

The first thing to note is that there are two types of placeholders. Some placeholders start with a `#`, like `#actor` and `#movie`, and other placeholders start with a ":", like `:actor`, `:a`, and `:m`.

The ones that start with colons are your *expression attribute values*. They are used to represent the value of the attribute you are evaluating in your request. Look at the `ExpressionAttributeValues` property in our request. You'll see that there are matching keys in the object for the three colon-prefixed placeholders in our request.

```
ExpressionAttributeValues={
  ':actor': { 'S': 'Tom Hanks' },
  ':a': { 'S': 'A' },
  ':m': { 'S': 'M' }
}
```

The values in the `ExpressionAttributeValues` property are substituted into our `KeyConditionExpression` by the DynamoDB server when it receives our request.

But why do we need this substitution? Why can't we just write our attribute values directly into the expression?

Remember that each attribute value in DynamoDB has a type. DynamoDB does not infer this type—you must explicitly provide it when you make a request. Attributes with the same underlying value but a different type are not equal. Thus a timestamp value of 1565954664 stored as a string is distinctly different than the same value stored as a number.

Because attribute types are so important, we need to specify them in our expressions. However, the manner in which types are indicated is difficult to express and parse in an expression. Imagine a world where the type was written into the expression, such as the following:

```
KeyConditionExpression: "#actor = { 'S': 'Tom Hanks' }"
```

We've replaced the `:actor` from our original expression with `{ 'S': 'Tom Hanks' }`. When this request makes it to the

79

DynamoDB server, the parsing logic is much more complicated. It will need to parse out all the attribute values by matching opening and closing brackets. This can be a complicated operation, particularly if you are writing a nested object with multiple levels. By splitting this out in a separate property, DynamoDB doesn't have to parse this string.

Further, it can help you client-side by validating the shape of the `ExpressionAttributeValues` parameter before making a request to the server. If you have formatted it incorrectly, you can know immediately in your application code.

Now let's look at the placeholders that start with a #. These are your *expression attribute names*. These are used to specify the name of the attribute you are evaluating in your request. Like `ExpressionAttributeValues`, you'll see that we have corresponding properties in our `ExpressionAttributeNames` parameter in the call.

If the `ExpressionAttributeValues` placeholder is about the type of the value, why do we need placeholders for attribute names? The same logic doesn't apply here, as attribute names are not typed.

Unlike `ExpressionAttributeValues`, you are not required to use `ExpressionAttributeNames` in your expressions. You can just write the name of the attribute directly in your expression, as follows:

```
KeyConditionExpression: 'Actor = :actor AND Movie BETWEEN :a AND :m'
```

That said, there are a few times when the use of `ExpressionAttributeNames` is required. The most common reason is if your attribute name is a reserved word in DynamoDB. There are 573 reserved words in DynamoDB, and many of them will conflict with normal attribute names. For example, the following words are all reserved words that are commonly used as attribute names:

- Bucket
- By
- Count
- Month
- Name
- Timestamp
- Timezone

If any of these words conflict, you'll need to use `ExpressionAttributeNames`. Because the list of reserved names is so long, I prefer not to check every time I'm writing an expression. In most cases, I'll use `ExpressionAttributeNames` just to be safe. That said, there are times in this book where I won't use them just to keep the examples terse.

Another common reason to use expression attribute names is when working with nested documents, such as an attribute of the map type, or if you have an item whose attribute name includes a period. DynamoDB interprets a period as accessing a nested attribute of an object, and this can result in some unexpected behavior if you're not using expression attribute names.

5.2. Don't use an ORM

Object-relational mappers (ORMs) are popular tools when using relational databases in your application code. Rather than writing raw SQL statements, developers can use ORMs to operate directly on objects in their programming languages.

There are fierce opinions between the pro-ORM and anti-ORM factions, and I'm not about to wade into that minefield. Generally, ORMs can be helpful for application developers that don't know

SQL as well, but they won't be as performant as writing the SQL yourself. Your choice in that situation may vary.

Regardless, I would not recommend using an ODM in DynamoDB (the NoSQL equivalent to ORMs is sometimes called an ODM, for Object-Document Mapper. There's not a great term here). There are a few reasons for this.

First, ODMs push you to model data incorrectly. ORMs make some sense in a relational world because there's a single way to model data. Each object type will get its own table, and relations are handled via foreign keys. Fetching related data involves following the foreign key relationships.

This isn't the case with DynamoDB. All of your object types are crammed into a single table, and sometimes you have multiple object types in a single DynamoDB item. Further, fetching an object and its related objects isn't straightforward like in SQL—it will depend heavily on the design of your primary key.

The second reason to avoid ODMs is that it doesn't really save you much time or code compared to the basic AWS SDK. Part of the benefit of ORMs is that it removes giant SQL strings from your application code. That's not happening with DynamoDB. DynamoDB is API-driven, so you'll have a native method for each API action you want to perform. Your ORM will mostly be replicating the same parameters as the AWS SDK, with no real gain in ease or readability.

There are two exceptions I'd make to my "No ODM / Use the bare SDK" stance. The first exception is for tools like the Document Client provided by AWS in their Node.js SDK. The Document Client is a thin wrapper on top of the core AWS SDK. The usage is quite similar to the core SDK—you'll still be doing the direct API actions, and you'll be specifying most of the same parameters. The difference is in working with attribute values.

With the core AWS SDK, you need to specify an attribute type whenever you're reading or writing an item. This results in a lot of annoying boilerplate, like the following:

```javascript
const AWS = require('aws-sdk')
const dynamoDB = new AWS.DynamoDB()


const resp = await dynamodb.putItem({
  TableName: 'MoviesAndActors',
  Item: {
      Actor: { 'S': 'Tom Hanks' },
      Movie: { 'S': 'Forrest Gump' },
      Role: { 'S': 'Forrest Gump' },
      Year: { 'S': '1994' },
      Genre: { 'S': 'Drama' }
  }
}).promise()
```

With the Document Client, attribute types are inferred for you. This means you can use regular JavaScript types, and they'll get converted into the proper attribute type by the Document Client before sending it to the DynamoDB server.

The example written above would be rewritten as:

```javascript
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

const resp = await docClient.put({
  TableName: 'MoviesAndActors',
  Item: {
      Actor: 'Tom Hanks',
      Movie: 'Forrest Gump',
      Role: 'Forrest Gump',
      Year: '1994',
      Genre: 'Drama'
  }
}).promise()
```

It's a small change but a nice quality-of-life improvement. The `Table` resource in Boto3, the AWS SDK for Python, has similar helper functionality that takes the edge off.

The second exception to my policy involves libraries like Jeremy Daly's DynamoDB Toolbox for Node.js. DynamoDB Toolbox is explicitly *not* an ORM. However, it does help you define entity types in your application and map those to your DynamoDB table. It's not going to do all the work to query the table for you, but it does simplify a lot of the boilerplate around interacting with DynamoDB.

I prefer this approach over an ORM as it's a helper, not a magic wand. You'll still need to model your database properly. You'll still need to understand how to translate application objects into database objects. And you'll still need to interact with the DynamoDB API yourself. This library and libraries like it help with the rough edges.

5.3. Understand the optional properties on individual requests

The final advice for working with the DynamoDB API is to understand the optional properties you can add on to individual requests. The properties are:

- `ConsistentRead`
- `ScanIndexForward`
- `ReturnValues`
- `ReturnConsumedCapacity`
- `ReturnItemCollectionMetrics`

The first three properties could affect the items you receive back in your DynamoDB request. The last two can return additional metric information about your table usage.

Let's review each of them.

5.3.1. ConsistentRead

In the Advanced Concepts chapter, we discussed the different consistency modes DynamoDB provides. By default, reads from DynamoDB are eventually consistent, meaning that the reads will likely, but not definitely, reflect all write operations that have happened before the given read operation.

For some use cases, eventual consistency may not be good enough. You may want to ensure that you're reading an item that contains all writes that have occurred to this point. To do that, you would need to indicate you want a strongly-consistent read.

To get a strongly-consistent read, you need to set `ConsistentRead=True` in your API call. The `ConsistentRead` property is available on four operations:

- GetItem
- BatchGetItem
- Query
- Scan

There are two more aspects to consistent reads that are worth knowing. First, opting into a strongly-consistent read consumes more read request units than using an eventually-consistent read. Each read request unit allows you to read an item up to 4KB in size. If you use the default of an eventually-consistent read, your read request units will be cut in half. Thus, reading a single item of up to 4KB in size would only cost you half of a read request unit. Opting into a strongly-consistent read will consume the full read request unit.

The second thing to note about consistency is related to secondary indexes. Note that the `ConsistentRead` property is available on the `Query` and `Scan` operations, which are the two API actions you can use with secondary indexes. If you are using a *local secondary index*, you may opt into strong consistency by passing `ConsistentRead=True`. However, you may not request strong consistency when using a global secondary index. All reads from a global secondary index are eventually consistent.

5.3.2. ScanIndexForward

The second API property to know is the `ScanIndexForward` property. This property is available only on the `Query` operation, and it controls which way you are reading results from the sort key.

This is best explained by way of an example. Imagine you have an IoT application where you are storing sensor data. Each item in the table reflects a reading at a particular point in time for a given sensor. You have a composite primary key where the partition key is `SensorId`, and the sort key is `Timestamp`.

The table might look as follows:

Primary key		Attributes
Partition key: SensorId	Sort key: Timestamp	
14891	2020-02-15 00:00:00	Temperature 32.4
	2020-02-15 00:01:00	Temperature 32.4
	2020-02-15 00:02:00	Temperature 32.3
	2020-02-15 01:...	
	2020-02-15 02:...	
	2020-02-15 15:47:00	Temperature 67.1
	2020-02-15 15:48:00	Temperature 67.4
	2020-02-15 15:49:00	Temperature 67.5

A common access pattern for your application may be to find the most recent 20 readings for a particular sensor. To do this, you would use a `Query` where the partition key is the ID for the sensor you want.

However, DynamoDB orders its sort key in *ascending* order. Imagine your sensor reported data every minute. After a single day of operation, your sensor would have 1440 readings. However, to find the most recent 20 readings, you would need to page through all 1440 readings until you got to the end. That would consume a lot of read capacity!

Primary key		Attributes	
Partition key: SensorId	Sort key: Timestamp		
14891	2020-02-15 00:00:00	Temperature	32.4
	2020-02-15 00:01:00	Temperature	32.4
	2020-02-15 00:02:00	Temperature	32.3
	2020-02-15 01:...		
	2020-02-15 02:...		
	2020-02-15 15:47:00	Temperature	67.1
	2020-02-15 15:48:00	Temperature	67.4
	2020-02-15 15:49:00	Temperature	67.5

Ascending order

The `ScanIndexForward` property allows you to flip the direction in which DynamoDB will read your sort key. If you set `ScanIndexForward=False`, then DynamoDB will read your sort key in *descending* order.

That makes our query operation much more efficient. We start at the most recent operation and limit our results to 20. This consumes significantly less read capacity.

Primary key		Attributes
Partition key: SensorId	Sort key: Timestamp	
14891	2020-02-15 00:00:00	Temperature 32.4
	2020-02-15 00:01:00	Temperature 32.4
	2020-02-15 00:02:00	Temperature 32.3
	2020-02-15 01:...	
	2020-02-15 02:...	
	2020-02-15 15:47:00	Temperature 67.1
	2020-02-15 15:48:00	Temperature 67.4
	2020-02-15 15:49:00	Temperature 67.5

Using the `ScanIndexForward` can help find the most recent timestamps or for reading items in reverse alphabetical order. It can also help with "pre-joining" your data in one-to-many relationships. For more on this, check out the one-to-many relationship strategies in Chapter 11.

5.3.3. ReturnValues

When working with items in DynamoDB, you may execute an operation against an item without knowing the full current state of the item. A few examples of this are:

- When writing an item, you may include a `ConditionExpression` to assert certain conditions about an existing item with the same primary key without knowing the exact attributes of the existing item.

- When updating an item, you could increment a number attribute on the item without knowing the current value of the attribute.

- When deleting an item, you may not know any of the current attributes on the item.

In each of these cases, it can be helpful to receive additional

information about the existing or updated item from DynamoDB after the call finishes. For example, if you were incrementing a counter during an update, you may want to know the value of the counter after the update is complete. Or if you deleted an item, you may want to view the item as it looked before deletion.

To help with this, certain API actions have a `ReturnValues` attribute that affects the payload that is returned to your client. This property is available on the following write-based API actions:

- `PutItem`
- `UpdateItem`
- `DeleteItem`
- `TransactWriteItem` (here, the property is referred to as `ReturnValuesOnConditionCheckFailure`)

By default, DynamoDB will not return any additional information about the item. You can use the `ReturnValues` attribute to change these defaults.

There are a few different available options for the `ReturnValues` property:

- **NONE:** Return no attributes from the item. This is the default setting.
- **ALL_OLD:** Return all the attributes from the item as it looked *before* the operation was applied.
- **UPDATED_OLD:** For any attributes updated in the operation, return the attributes *before* the operation was applied.
- **ALL_NEW:** Return all the attributes from the item as it looks *after* the operation is applied.
- **UPDATED_NEW:** For any attributes updated in the operation, return the attributes *after* the operation is applied.

For an example of the `ReturnValues` attribute in action, check out the auto-incrementing integers strategy in Chapter 16 and the implementation in the GitHub example in Chapter 21.

5.3.4. ReturnConsumedCapacity

The fourth property we'll review is different from the first three, as it won't change the item(s) that will be returned from your request. Rather, it is an option to include additional metrics with your request.

The `ReturnConsumedCapacity` property is an optional property that will return data about the capacity units that were used by the request. Recall that DynamoDB is priced based on read and write capacity units, and the amount of capacity used depends on a few factors:

- the size of the item(s) you're reading or writing;
- for read operations, whether you're doing an eventually- or strongly-consistent read;
- whether you're making a transactional request.

There are a few levels of detail you can get with the consumed capacity. You can specify `ReturnConsumedCapacity=INDEXES`, which will include detailed information on not only the consumed capacity used for your base table but also for any secondary indexes that were involved in the operation. If you don't need that level of precision, you can specify `ReturnConsumedCapacity=TOTAL` to simply receive an overall summary of the capacity consumed in the operation.

The returned data for your consumed capacity will look similar to the following:

```
'ConsumedCapacity': {
    'TableName': 'GitHub',
    'CapacityUnits': 123.0,
    'ReadCapacityUnits': 123.0,
    'WriteCapacityUnits': 123.0,
    'GitHub': {
        'ReadCapacityUnits': 123.0,
        'WriteCapacityUnits': 123.0,
        'CapacityUnits': 123.0
    },
    'GlobalSecondaryIndexes': {
        'GSI1': {
            'ReadCapacityUnits': 123.0,
            'WriteCapacityUnits': 123.0,
            'CapacityUnits': 123.0
        }
    }
}
```

The consumed capacity numbers won't be helpful in every scenario, but there are a few times you may want to use them. There are two situations I find them most helpful.

First, you may use these metrics if you are in the early stages of designing your table. Perhaps you feel pretty good about your table design. To confirm it will work as intended, you can build out a prototype and throw some sample traffic to see how it will perform. During your testing, you can use the consumed capacity metrics to get a sense of which access patterns use a lot of capacity or to understand how much capacity you'll need if your traffic goes to 10 or 100 times your testing.

The second example where consumed capacity is useful is if you're passing on the underlying cost of DynamoDB table access to your customers. For example, maybe you have a pricing scheme where users are charged according to the size of read or write items. You could use the consumed capacity metrics to track the size of the items written and allocate them to the proper customer.

5.3.5. ReturnItemCollectionMetrics

The final API property we'll cover is `ReturnItemCollectionMetrics`. Like the `ReturnConsumedCapacity` property, this will not change the item(s) returned from your request. It will only include additional metadata about your table.

In Chapter 2, we introduced the concept of *item collections*. An item collection refers to all items in a table or index that have the same partition key. For example, in our Actors and Movies example below, we have three different item collections: Tom Hanks, Tim Allen, and Natalie Portman.

Primary key		Attributes		
Partition key: Actor	**Sort key: Movie**			
Tom Hanks	Cast Away	Role	Year	Genre
		Chuck Noland	2000	Drama
	Toy Story	Role	Year	Genre
		Woody	1995	Children's
Tim Allen	Toy Story	Role	Year	Genre
		Buzz Lightyear	1995	Children's
Natalie Portman	Black Swan	Role	Year	Genre
		Nina Sayers	2010	Drama

If you have a local secondary index on a table, all the items in the local secondary index are included as part of the same item collection as the base table. Thus, in the example below, I have a local secondary index on my Actors and Movies table where the key schema has a partition key of `Actor` and a sort key of `Year`.

Primary key		Attributes		
Partition key: Actor	Sort key: Year			
Tom Hanks	1995	Movie	Role	Genre
		Toy Story	Woody	Children's
	2000	Movie	Role	Genre
		Cast Away	Chuck Noland	Drama
Tim Allen	1995	Movie	Role	Genre
		Toy Story	Buzz Lightyear	Children's
Natalie Portman	2010	Movie	Role	Genre
		Black Swan	Nina Sayers	Drama

Recall that local secondary indexes must use the same partition key as the base table. When calculating the item collection size for Tom Hanks, I would include *both* the size of the items in the base table and the size of the items in the local secondary index.

Here's the critical part: **If your table has a local secondary index, then a single item collection cannot be larger than 10GB in size.** If you try to write an item that would exceed this limit, the write will be rejected.

This could be a painful surprise to discover that writes to your database are failing due to an item collection size limit. To give yourself advanced warning of this problem, you can use the `ReturnItemCollectionMetrics` property on all write-based API calls. After you receive a response, you can check the size of the altered item collection. If the item collection exceeds a given threshold—perhaps 7 or 8 GB—you could trigger an internal alert that you are approaching an item collection limit for a particular partition key. This would give you a warning of a potential issue before writes started failing.

Apart from the item collection size limit imposed by a local secondary index, this property isn't all that helpful. If you don't have a local secondary index, there are no limits on the size of an item collection.

If you choose to return item collection metrics, your response will have data similar to the following:

```
'ItemCollectionMetrics': {
    'ItemCollectionKey': {
        'S': 'USER#alexdebrie'
    },
    'SizeEstimateRangeGB': [
        123.0,
    ]
}
```

5.4. Summary

In this chapter, we reviewed some tips for working with the
DynamoDB API. In general, you should work with the DynamoDB
API as little as possible. Most of your work will be done before you
write any code. Model out your data in a spreadsheet or in a
modeling tool like the NoSQL Workbench, which is provided by
AWS. Once you've done the work to model your entities and map
out your access patterns, then do the work to translate that to code
in your application.

Even though your work with the DynamoDB API will be limited, it
is helpful to know your way around. In this chapter, we looked at
the basics behind expression attribute names and values, we saw
that you should avoid ODMs, and we learned about some optional
properties in your API requests.

In the next chapter, we'll take a deeper look at the various
expressions in DynamoDB.

Chapter 6. Expressions

Chapter Summary

This chapter covers DynamoDB expressions that are used in the DynamoDB API. You will learn about the five kinds of expressions and when they are helpful.

Sections

1. Key Condition Expressions
2. Filter Expressions
3. Projection Expressions
4. Condition Expressions
5. Update Expressions

As you're working with the DynamoDB API, a lot of your time will be spent writing various kinds of expressions. Expressions are statements that operate on your items. They're sort of like mini-SQL statements. In this chapter, we'll do a deep dive into expressions in DynamoDB.

There are five types of expressions in DynamoDB:

- **Key Condition Expressions:** Used in the Query API call to describe which items you want to retrieve in your query

- **Filter Expressions:** Used in Query and Scan operations to describe which items should be returned to the client after finding items that match your key condition expression

- **Projection Expressions:** Used in all read operations to describe which attributes you want to return on items that were read

- **Condition Expressions:** Used in write operations to assert the existing condition (or non-condition) of an item before writing to it

- **Update Expressions:** Used in the UpdateItem call to describe the desired updates to an existing item

The first three expressions are for read-based operations. The fourth, Condition Expressions, is for all write-based operations, and the last, Update Expressions, is for update operations only.

All expressions may use the expression attribute names that we discussed in the previous section, and all expressions other than the projection expression must use expression attribute values.

Let's review each of these expressions and their usage.

6.1. Key Condition Expressions

The key condition expression is the expression you'll use the most. It is used on every Query operation to describe which items you want to fetch.

Remember that the Query API call must include a partition key in the request. It may also include conditions on the sort key in the request. The key condition expression is how you express these parameters in your request. A key condition can be used *only* on elements of the primary key, not on other attributes on the items.

Let's start with a simple Query request using just the partition key. Returning to our popular Movies and Actors example in a previous chapter, the Query request below shows the

KeyConditionExpression, the ExpressionAttributeNames, and the ExpressionAttributeValues for fetching all movie roles for Natalie Portman:

```
result = dynamodb.query(
    TableName='MovieRoles',
    KeyConditionExpression="#a = :a",
    ExpressionAttributeNames={
        "#a": "Actor"
    },
    ExpressionAttributeValues={
        ":a": { "S": "Natalie Portman" }
    }
)
```

The partition key of our MovieRoles table is `Actor`. Once we've declared the attribute name in ExpressionAttributeNames and the value in ExpressionAttributeValues, our KeyConditionExpression expresses our desire to retrieve all items with Natalie Portman as the actor.

You can also use conditions on the sort key in your key condition expression. This is useful for finding a specific subset of your data. All elements with a given partition key are sorted according to the sort key (hence the name). If your sort key is a number, it will be ordered numerically. If your sort key is a string, it will be sorted according to its UTF-8 bytes, which is essentially alphabetically, with provision for characters not in the alphabet and placing all uppercase letters before lowercase. For more on sorting in DynamoDB, see Chapter 13.

You can use simple comparisons in your sort key conditions, such as greater than (>), less than (<), or equal to (=). In our example, we could use it to filter our query results by the name of the movie title:

```
result = dynamodb.query(
    TableName='MovieRoles',
    KeyConditionExpression="#a = :a AND #m < :title",
    ExpressionAttributeNames={
        "#a": "Actor",
        "#m": "Movie"
    },
    ExpressionAttributeValues={
        ":a": { "S": "Natalie Portman" },
        ":title": { "S": "N" }
    }
)
```

In this example, we are specifying that the `Movie` value must come before the letter "N".

A typical example of using the key condition on the sort key is when your sort key is a timestamp. You can use the sort key condition to select all items that occurred before, after, or during a specific time range. Imagine we had a table that represented orders from a customer. The table had a composite primary key with a partition key of `CustomerId` and a sort key of `OrderTime`, which is an ISO-8601 timestamp. The image below shows some example records.

Primary key		Attributes	
Partition key: CustomerId	Sort key: OrderTime		
aef7159cd662	2020-01-06 14:22:48	Amount	Status
		42.19	SHIPPED
36ab55a589e4	2020-01-08 02:27:04	Amount	Status
		179.98	CANCELLED
	2020-01-11 04:24:58	Amount	Status
		66.21	SHIPPED
	2020-01-16 02:01:36	Amount	Status
		87.77	PLACED
f7f2cb482b74	2020-01-15 14:28:29	Amount	Status
		12.44	SHIPPED

We have five orders here. The middle three are from the same `CustomerId`: 36ab55a589e4. Imagine that customer wants to view all orders between January 10 and January 20. We could handle that with the following Query:

```
result = dynamodb.query(
    TableName='CustomerOrders',
    KeyConditionExpression="#c = :c AND #ot BETWEEN :start and :end",
    ExpressionAttributeNames={
        "#c": "CustomerId",
        "#ot": "OrderTime"
    },
    ExpressionAttributeValues={
        ":c": { "S": "36ab55a589e4" },
        ":start": { "S": "2020-01-10T00:00:00.000000" },
        ":end": { "S": "2020-01-20T00:00:00.000000" }
    }
)
```

In addition to our partition key specifying the `CustomerId` we want, we also use the `BETWEEN` operator with the sort key to find all items between our given time range. It would return the two items in our table outlined in red:

Primary key		Attributes	
Partition key: CustomerId	Sort key: OrderTime		
aef7159cd662	2020-01-06 14:22:48	Amount	Status
		42.19	SHIPPED
36ab55a589e4	2020-01-08 02:27:04	Amount	Status
		179.98	CANCELLED
	2020-01-11 04:24:58	Amount	Status
		66.21	SHIPPED
	2020-01-16 02:01:36	Amount	Status
		87.77	PLACED
f7f2cb482b74	2020-01-15 14:28:29	Amount	Status
		12.44	SHIPPED

While you can use greater than, less than, equal to, or `begins_with`, one little secret is that *every* condition on the sort key can be

expressed with the `BETWEEN` operator. Because of that, I almost always use it in my expressions.

Key expressions are critical when fetching multiple, heterogeneous items in a single request. For more on this, check out Chapter 11, which includes strategies for modeling one-to-many relationships in DynamoDB. With the right combination of item collection design and proper key expressions, you can essentially 'join' your data just like a relational database but without the performance impact of joins.

6.2. Filter Expressions

The second type of expression is also for read-based operations. A filter expression is available for both Query and Scan operations, where you will be receiving multiple items back in a request. As its name implies, the filter expression is used for filtering. It will filter items that matched your key condition expression but not the filter condition.

The key difference with a filter expression vs. a key condition expression is that a filter expression can be applied on *any* attribute in the table, not just those in the primary key. This may feel more flexible than the key condition expression, but we'll soon see its shortfalls.

Let's see an example with our Movies & Actors table. Recall that our base table uses `Actor` as the partition key and `Movie` as the sort key.

Primary key		Attributes			
Partition key: Actor	Sort key: Movie				
Tom Hanks	Cast Away	Role	Year	Genre	
		Chuck Noland	2000	Drama	
	Toy Story	Role	Year	Genre	
		Woody	1995	Children's	
Tim Allen	Toy Story	Role	Year	Genre	
		Buzz Lightyear	1995	Children's	
Natalie Portman	Black Swan	Role	Year	Genre	
		Nina Sayers	2010	Drama	

However, that means if I want to filter on something like `Year` or `Genre`, I can't do it with my key condition expression. This is where the filter expression helps me out. If I wanted to find all of Tom Hank's movies where the `Genre` was "Drama", I could write the following Query:

```
result = dynamodb.query(
    TableName='MovieRoles',
    KeyConditionExpression="#actor = :actor",
    FilterExpression="#genre = :genre",
    ExpressionAttributeNames={
        "#actor": "Actor",
        "#genre": "Genre"
    },
    ExpressionAttributeValues={
        ":actor": { "S": "Tom Hanks" },
        ":genre": { "S": "Drama" }
    }
)
```

Notice the `FilterExpression` parameter, which uses expression attribute names and values to filter for items where the `Genre` attribute is "Drama". This would return a single result—Cast Away—from our table.

The filter expression can also be used in the Scan operation. In our

table example, we could use the same `Genre=Drama` filter expression to find the two Dramas in our table—Tom Hanks in Cast Away and Natalie Portman in Black Swan.

With a tiny table that includes five small items, it seems like the filter expression is a way to enable any variety of access patterns on your table. After all, we can filter on any property in the table, just like SQL!

This is not the case in reality. To understand this, you need to know how DynamoDB's read limits interact with the expression evaluation ordering.

When you issue a Query or Scan request to DynamoDB, DynamoDB performs the following actions in order:
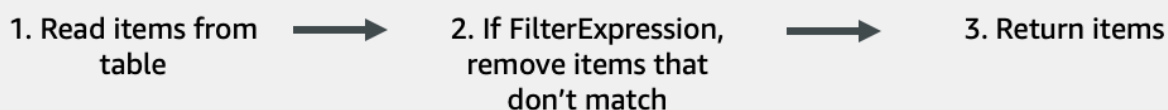
First, it reads items matching your Query or Scan from the database. Second, if a filter expression is present, it filters out items from the results that don't match the filter expression. Third, it returns any remaining items to the client.

However, the key point to understand is that the Query and Scan operations will return a maximum of 1MB of data, and this limit is applied in step 1, **before the filter expression is applied**.

1MB limit

1. Read items from table → 2. If FilterExpression, remove items that don't match → 3. Return items
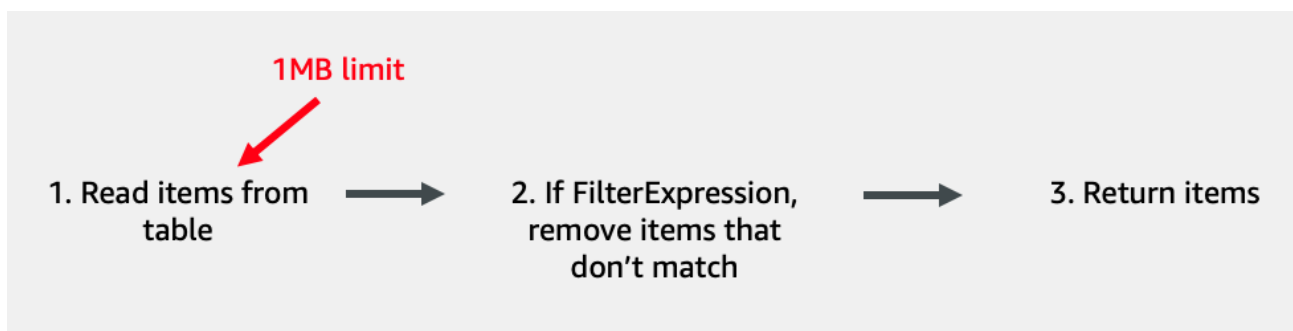
Imagine our Movies & Actors table was 1 GB in size but that the combined size of the Drama movies was only 100KB. You might expect the Scan operation to return all the Drama movies in one request since it is under the 1MB limit. However, since the filter expression is not applied until after the items are read, your client will need to page through 1000 requests to properly scan your table. Many of these requests will return empty results as all non-matching items have been filtered out.

A 1GB table is a pretty small table for DynamoDB — chances are that yours will be much bigger. This makes the Scan + filter expression combo even less viable, particularly for OLTP-like use cases.

A filter expression isn't a silver bullet that will save you from modeling your data properly. If you want to filter your data, you need to make sure your access patterns are built directly into your primary keys. Filter expressions can save you a bit of data sent over the wire, but it won't help you find data more quickly.

Filter expressions are useful for a few limited contexts:

1. *Reducing response payload size*. DynamoDB can return up to 1MB in a single response. That's a lot of data to send over the wire, particularly if you're just going to filter out a bunch of it once it's returned. Server-side filtering can reduce the data transferred and speed up response times.

2. *Easier application filtering*. If you'll retrieve some results from DynamoDB and immediately run a `filter()` method to throw

some away, it can be easier to handle than in your API request to DynamoDB. This is more of personal preference.

3. *Better validation around time-to-live (TTL) expiry.* When using DynamoDB TTL, AWS states that items are generally deleted within 48 hours of their TTL expiry. This is a wide range! If you're counting on expiration as a part of your business logic, you could get incorrect results. To help guard against this, you could write a filter expression that removes all items that should have been expired by now, even if they're not quite expired yet. To see an example of this in action, check out Chapter 17.

In general, filter expressions are helpful but limited. Don't rely on them too heavily. Instead, model your data correctly and use the filtering strategies shown in Chapter 12.

6.3. Projection expressions

The last kind of read-based expression is a projection expression. A projection expression is similar to a filter expression in that its main utility is in reducing the amount of data sent over the wire in your response. While a filter expression works on an item-by-item basis, the projection expression works on an attribute-by-attribute basis within an item.

Let's alter our Movies & Actors example to imagine that each item included a `CoverImage` attribute that represented the cover artwork for the movie as a giant blob of binary data. Now our table looks as follows:

Primary key		Attributes			
Partition key: Actor	Sort key: Movie				
Tom Hanks	Cast Away	Role	Year	Genre	CoverImage
		Chuck Noland	2000	Drama	..... 350KB of image data ....
	Toy Story	Role	Year	Genre	CoverImage
		Woody	1995	Children's	..... 350KB of image data ....
Tim Allen	Toy Story	Role	Year	Genre	CoverImage
		Buzz Lightyear	1995	Children's	..... 350KB of image data ....
Natalie Portman	Black Swan	Role	Year	Genre	CoverImage
		Nina Sayers	2010	Drama	..... 350KB of image data ....

For many of our requests, the large `CoverImage` attribute may not be necessary. Sending nearly 350KB of image data per item is a waste of bandwidth and memory in my application.

I can use projection expressions to solve this problem. A projection expression is used to specify exactly the attributes you want to receive from the DynamoDB server. Our example query to get all of Tom Hank's movies could look as follows:

```
result = dynamodb.query(
    TableName='MovieRoles',
    KeyConditionExpression="#actor = :actor",
    ProjectionExpression: "#actor, #movie, #role, #year, #genre"
    ExpressionAttributeNames={
        "#actor": "Actor",
        "#movie": "Movie",
        "#role": "Role",
        "#year": "Year",
        "#genre": "Genre"
    },
    ExpressionAttributeValues={
        ":actor": { "S": "Tom Hanks" }
    }
)
```

In our `ProjectionExpression`, we specified each of the attributes we wanted (all of the attributes except `CoverImage`). Note that we used expression attribute names for all of these. We didn't need to, but we would have had to for `Year`, as that's a reserved word in DynamoDB. Rather than checking the reserved list for every

attribute, I prefer to just use expression attribute names for all attributes.

The projection expression can also be used to access nested properties, such as in a list or map attribute. If you have a large object and you know the exact property you want, you can really cut down on the bandwidth by specifying only the elements that you need.

Projection expressions are subject to the same caveats as filter expressions—they are evaluated *after* the items are read from the table and the 1MB limit is reached. Thus, if you are looking to fetch a large number of items but each of them has a large attribute, you may find yourself paginating more than you'd like. If this is the case, you may need to create a secondary index with a custom projection that only copies certain attributes into the index. This will allow you to quickly query multiple items without having to paginate due to large, unneeded attributes.

6.4. Condition Expressions

The first three expressions we've discussed are for read operations. The next two expressions are for write operations. We'll review condition expressions first.

Condition expressions are available on every operation where you will alter an item—PutItem, UpdateItem, DeletetItem, and their batch and transactional equivalents. They allow you to assert specific statements about the status of the item before performing the write operation. If the condition expression evaluates to false, the operation will be canceled.

There are a number of reasons you may want to add condition expressions to your write operations, such as:

- To avoid overwriting an existing item when using PutItem;

- To prevent an UpdateItem operation from putting an item in a bad state, such as reducing an account balance below 0;

- To assert that a given user is the owner of an item when calling DeleteItem.

Without the existence of condition expressions, you would need to add costly additional requests to fetch an item before manipulating it, and you would need to consider how to handle race conditions if another request tried to manipulate your item at the same time.

Condition expressions use a similar syntax to the previous expressions we've seen. In addition to the comparison operators of greater than, less than, equal to, and between, there are also several functions that are available, such as:

- **attribute_exists():** Used to assert that a given attribute exists

- **attribute_not_exists():** Just the opposite—assert that an attribute does not exist on the item. This one is commonly used to prevent overwrites by using it on the partition key of the item you're writing.

- **attribute_type():** Used to assert that an attribute is of a particular type

- **begins_with():** Assert that an attribute value begins with a particular substring

- **contains():** Assert that a string contains a particular substring, or that a set contains a particular value.

- **size():** Allows you to assert various properties about the size of an attribute value. For things like strings or binary values, it's the length of the string or number of bytes in the binary value. For things like lists, maps, or sets, it returns the number of elements in a set.

Condition expressions can operate on any attribute on your item, not just those in the primary key. This is because condition expressions are used with item-based actions where the item in question has already been identified by passing the key in a different parameter.

Let's walk through a few quick examples where these can be useful.

6.4.1. Preventing overwrites or checking for uniqueness

The PutItem API action will insert an item into your table and completely overwrite any existing item with the same primary key. This is often undesirable, as you don't want to blow away existing data.

You can prevent this overwrite behavior by using the `attribute_not_exists()` operator in your condition expression.

Let's see an example of this in action. Imagine we had a Users table with a simple primary key of `Username`. When creating a new user, we want to ensure that there isn't an existing user with the same username. The code below shows how we would do this.

```
result = dynamodb.put_item(
    TableName='Users',
    Item={
        "Username": { "S": "bountyhunter1" },
        "Name": { "S": "Boba Fett" },
        "CreatedAt": { "S": datetime.datetime.now().isoformat() }
    },
    ConditionExpression: "attribute_not_exists(#username)",
    ExpressionAttributeNames={
        "#username": "Username"
    }
)
```

We specify the attributes we want to set on our item, but we also

include a `ConditionExpression` parameter that asserts that there is no item with the same username.

This is one of the more common condition expressions I use, as I use it to manage uniqueness requirements on almost all entities in DynamoDB. Over 90% of my PutItem requests include condition expressions to assert there is not an existing item with the same primary key.

6.4.2. Limiting in-progress items

Another common use case for conditions is to limit the number of in-progress items for a workflow or to limit the size of a group. For example, maybe you have a set of background workers that are processing a queue. To protect your downstream services, you don't want more than 10 jobs running in a particular state at a given time.

You could use an UpdateItem API action as follows:

```
result = dynamodb.update_item(
    TableName='WorkQueue',
    Key={
        "PK": { "S": "Tracker" }
    }
    ConditionExpression: "size(#inprogress) <= 10",
    UpdateExpression="Add #inprogress :id",
    ExpressionAttributeNames={
        "#inprogress": "InProgress"
    },
    ExpressionAttributeValues={
        ":id": { "SS": [ <jobId> ] }
    }
)
```

In this example, we have a tracking item whose sole job is to manage the number of jobs in a particular state. It has an `InProgress` attribute of type string set to manage all jobs in the given state.

When we have a job that we want to start the guarded state, we run an `UpdateItem` call to attempt to add the job ID into the `InProgress` set. In our `UpdateItem` call, we specify a condition expression that the current size of the `InProgress` set is less than or equal to 10. If it's more than 10, we have the maximum number of jobs in that stage and need to wait until a spot is open.

We cover update expressions in the next section but take a quick look at the `UpdateExpression` while we're here. The full expression is `Add #inprogress :id`. It's directing DynamoDB to add a particular value to the set attribute called `InProgress`. It uses expression attribute names and values, like the other expressions. The terse update syntax is pretty powerful for expressing your update needs.

6.4.3. Asserting user permissions on an item

You may have items in your table that can be read by many users but can only be updated by a select group of administrators. For example, you could have a table that contains billing details for your SaaS subscription that looks as follows:

Primary key	Attributes	
Partition key: PK		
Amazon	SubscriptionType	Admins
	Enterprise	["JeffBezos","AndyJassy"]
Google	SubscriptionType	Admins
	Pro	["Satya Nadella"]
Oracle	SubscriptionType	Admins
	Free	["Larry Ellison"]
Facebook	SubscriptionType	Admins
	Enterprise	["Mark Zuckerberg"]

Each organization has an item in the table that describes the current subscription plan they're on (`SubscriptionType`), as well as a set of usernames that have the authority to change the subscription plan or the payment details (`Admins`). Before changing the billing details, you need to confirm that the user making the request is an admin.

You can do this by using the `contains()` function in your condition expression, as shown below:

```python
result = dynamodb.update_item(
    TableName='BillingDetails',
    Key={
        "PK": { "S": 'Amazon' }
    }
    ConditionExpression="contains(#a, :user)",
    UpdateExpression="Set #st :type",
    ExpressionAttributeNames={
        "#a": "Admins",
        "#st": "SubscriptionType"
    },
    ExpressionAttributeValues={
        ":user": { "S": 'Jeff Bezos' },
        ":type": { "S": 'Pro' }
    }
)
```

In our condition expression, we are using the `contains()` function to assert that Jeff Bezos, the user making the request, is an admin for the Amazon organization. If that condition is satisfied, then the update request can continue.

6.4.4. Checks across multiple items

A final pattern where you may use condition expressions involves checking across multiple entities. This could be an extension of the previous example where you need to check if a user is an admin. However, there may be some items that are only editable by administrators. Rather than storing the permission information in each item, you may store the list of administrators in a separate item that may be checked in needed requests.

DynamoDB transactions can help us here. The TransactWriteItem API allows you to use up to 10 items in a single request. They can be a combination of different write operations—PutItem, UpdateItem, or DeleteItem—or they can be ConditionChecks, which simply assert a condition about a particular item.

Let's see how a ConditionCheck can help us with our problem. Let's assume a similar setup as the previous example. In this example, a user wants to delete their subscription altogether. Further, the list of admins is kept on a different item because the admins are used to verify a number of different requests.

Our TransactWriteItems request is as follows:

```
result = dynamodb.transact_write_items(
    TransactItems=[
        {
            "ConditionCheck": {
                "Key": {
                    "PK": { "S": "Admins#<orgId>" }
                },
                "TableName": "SaasApp",
                ConditionExpression: "contains(#a, :user)",
                ExpressionAttributeNames={
                    "#a": "Admins"
                },
                ExpressionAttributeValues={
                    ":user": { "S": <username> }
                }
            }
        },
        {
            "Delete": {
                "Key": {
                    "PK": { "S": "Billing#<orgId>" }
                },
                "TableName": "SaasApp"
            }
        }
    ]
}
```

We have two operations in our TransactWriteItems request. First, there is a ConditionCheck on our Admins item for this organization to assert that the requesting user is an admin in this account.

Second, there is a Delete operation to remove the billing record for this organization. These operations will succeed or fail together. If the condition check fails because the user is not an administrator, the billing record will not be deleted.

6.5. Update Expressions

The final type of expression is the update expression. Like a condition expression, it is for a write-based action. However, it is distinct from all other expressions in that it is actually manipulating an item rather than reading or asserting the truth about existing properties.

Note that when using the UpdateItem API, you will only alter the properties you specify. If the item already exists in the table, the attributes that you don't specify will remain the same as before the update operation. If you don't want this behavior, you should use the PutItem API, which will completely overwrite the item with only the properties you give it.

In an update expression, you need to state the changes you want to make. There are four verbs for stating these changes:

- **SET:** Used for adding or overwriting an attribute on an item. Can also be used to add or subtract from a number attribute

- **REMOVE:** Used for deleting an attribute from an item or deleting nested properties from a list or map

- **ADD:** Used for adding to a number attribute or inserting an element into a set attribute

- **DELETE:** Used for removing an element from a set attribute

You may use any combination of these four verbs in a single update statement, and you may use multiple operations for a single verb.

If you have multiple operations for a single verb, you only state the verb once and use commas to separate the clauses, as shown below:

```
UpdateExpression="SET Name = :name, UpdatedAt = :updatedAt"
```

In the operation above, we're setting both the `Name` and `UpdatedAt` attributes to new values.

If you want to include multiple verbs in the same update expression, you don't need anything to separate the verb clauses. The presence of reserved verb words will be enough. For example:

```
UpdateExpression="SET Name = :name, UpdatedAt = :updatedAt REMOVE InProgress"
```

In the example above, we're using our same SET clause to update the Name and UpdatedAt properties, and we're using a REMOVE clause to delete the InProgress attribute from our item.

The best way to learn these is through some common examples, so let's look through a few.

6.5.1. Updating or setting an attribute on an item

The most common example of update expression is to set the value of an attribute. This may mean overwriting an existing attribute or setting a brand new attribute. An example UpdateItem request to set an attribute is below:

```
result = dynamodb.update_item(
    TableName='Users',
    Key={
        "Username": { "S": "python_fan" }
    }
    UpdateExpression="SET #picture :url",
    ExpressionAttributeNames={
        "#picture": "ProfilePictureUrl"
    },
    ExpressionAttributeValues={
        ":url": { "S": <https://....> }
    }
)
```

In this example, a user is updating the profile picture associated with their account. It doesn't matter if there was a `ProfilePictureUrl` attribute previously for this user. After the update operation, this attribute will now be set.

6.5.2. Deleting an attribute from an item

The opposite of setting an item is useful too—sometimes you want to delete an attribute for an item. This can be useful for a few reasons. You might just need to remove an attribute from an item because the attribute has an important value in your application that is no longer true about the item.

An example of removing an attribute from an item is below:

```
result = dynamodb.update_item(
    TableName='Users',
    Key={
        "Username": { "S": "python_fan" }
    }
    UpdateExpression="REMOVE #picture",
    ExpressionAttributeNames={
        "#picture": "ProfilePictureUrl"
    }
)
```

In this example, we're undoing the action we took in the last example. If the user deletes the picture from their user profile, we

want to delete it from their item.

Another reason to remove attributes is more DynamoDB-specific. Imagine you're using a global secondary index with a sparse pattern where only items with a given attribute are in the sparse index. If you no longer want your item to appear in that index, you could delete the attribute from the item. An example of this pattern can be found in Chapter 20, where we remove secondary index attributes from Messages once they are marked as read.

6.5.3. Incrementing a numeric value

Another common use case is incrementing or decrementing a number on an item. Maybe you want to track the page views on a particular page, or you want to reduce the inventory count for an item after you make a sale. You can use update expressions to increment or decrement the existing value by a given number, as shown below:

```
result = dynamodb.update_item(
    TableName='PageViews',
    Key={
        "Page": { "S": "ContactUsPage" }
    }
    UpdateExpression="SET #views = #views + :inc",
    ExpressionAttributeNames={
        "#views": "PageViews"
    },
    ExpressionAttributeValues={
        ":inc": { "N": "1" }
    }
)
```

In this example, we are recording a page view on the ContactUsPage item. Note that we're setting the new value equal to the existing value plus 1. This is powerful—it saves us from having to make one request to read the value and another request to set the new value. Additionally, we don't have to worry about race conditions if our application is trying to increment that value

multiple times in the same period.

6.5.4. Adding a nested property

The document attribute types—lists and maps—are powerful for expressing complex objects in your items. DynamoDB also allows you to act directly on nested properties within these document types.

Let's see how that works in practice using the example below.

```python
result = dynamodb.update_item(
    TableName='Users',
    Key={
        "Username": { "S": "python_fan" }
    }
    UpdateExpression="SET #phone.#mobile :cell",
    ExpressionAttributeNames={
        "#phone": "PhoneNumbers",
        "#mobile": "MobileNumber"
    },
    ExpressionAttributeValues={
        ":cell": { "S": "+1-555-555-5555" }
    }
)
```

Imagine we had a map property of `PhoneNumbers` on our user profile items. The `PhoneNumbers` property stores a flexible number of named phone numbers—Home, Business, Mobile, etc. A user may have all or none of these numbers set on their profile.

Using the SET verb, you can operate directly on a nested property in your map attribute. In our example, we're setting the `PhoneNumber.MobileNumber` property in our update. Like the increment example, this saves us from making multiple requests first to read the existing attribute and then update the full attribute.

6.5.5. Adding and removing from a set

The final example I want to cover is manipulating items in a set type attribute. We saw an example in the Condition Expression section, where we were checking that a user was an administrator before allowing a particular update expression. Now, let's see how we can add and remove administrators from that set attribute.

First, you can add a new user to the administration set using the ADD verb:

```
result = dynamodb.update_item(
    TableName="SaasApp",
    Key={ "PK": { "S": "Admins#<orgId>" },
    UpdateExpression="ADD #a :user",
    ExpressionAttributeNames={
        "#a": "Admins"
    },
    ExpressionAttributeValues={
        ":user": { "SS": ["an_admin_user"] }
    }
)
```

Our update expression of `ADD #a :user` tells DynamoDB to add our given username string into the `Admins` set.

The nice thing about using sets is that this is an idempotent operation—you could run it five times in a row, and it will end with the same result. If the given string already exists in the set, DynamoDB won't throw an error.

Similarly, you could remove elements from the set with the REMOVE verb:

```
result = dynamodb.update_item(
    TableName="SaasApp",
    Key={ "PK": { "S": "Admins#<orgId>" },
    UpdateExpression="REMOVE #a :user",
    ExpressionAttributeNames={
        "#a": "Admins"
    },
    ExpressionAttributeValues={
        ":user": { "SS": ["an_admin_user"] }
    }
)
```

This is the exact same as our ADD operation, but with the verb switched to REMOVE.

Note that you can add and remove multiple elements to a set in a single request. Simply update your expression attribute value to contain multiple items, as shown below:

```
result = dynamodb.update_item(
    TableName="SaasApp",
    Key={ "PK": { "S": "Admins#<orgId>" },
    UpdateExpression="ADD #a :user",
    ExpressionAttributeNames={
        "#a": "Admins"
    },
    ExpressionAttributeValues={
        ":user": { "SS": ["an_admin_user", "another_user"] }
    }
)
```

This operation would add two elements to the set--`an_admin_user` and `another_user`.

6.6. Summary

In this chapter, we read about the five different types of expressions in DynamoDB. Three types of expressions are for read-based operations: key condition expressions, filter expressions, and projection expressions. The other two expression types, condition

expressions and update expressions, are used for write-based operations. Understanding the mechanics of these operations, particularly the key condition expression and the write-based expressions, is crucial for effective data modeling with DynamoDB.

Chapter 7. How to approach data modeling in DynamoDB

Chapter Summary

This chapter details the process of building a data model with DynamoDB. First, it includes some key differences between data modeling with a relational database and data modeling with DynamoDB. Then, it describes the steps you should follow to create an efficient data model in DynamoDB.

Sections

1. Differences with relational databases
2. Steps for modeling with DynamoDB

To this point, we've covered a lot of background information on DynamoDB. We learned the basic and advanced concepts in DynamoDB, and we walked through the key elements of the DynamoDB API.

In this chapter and the next few chapters, we will drill in on data modeling with DynamoDB. This chapter will look at how data modeling with a NoSQL database is different than a relational database, as well as the concrete steps you should follow when modeling with DynamoDB.

7.1. Differences with relational databases

First, let's look at a few ways that data modeling in DynamoDB is different than data modeling with a relational database. Relational databases have been the de facto standard for a few decades, and most developers have familiarity with relational data modeling.

It's a bad idea to model your data in DynamoDB the same way you model your data in a relational database. The entire point of using a NoSQL datastore is to get some benefit you couldn't get with a relational database. If you model the data in the same way, you not only won't get that benefit but you will also end up with a solution that's worse than using the relational database!

Below are a few key areas where DynamoDB differs from relational databases.

7.1.1. Joins

In a relational database, you use the `JOIN` operator to connect data from different tables in your query. It's a powerful way to reassemble your data and provide flexibility in your access patterns.

But joins come at a cost. Joins need large amounts of CPU to combine your disparate units of data. Further, joins work best when all relevant data is co-located on a single machine so that you don't need to wait on a network call to fetch data from different instances. However, this co-location requirement will limit your ability to scale. You can only take advantage of vertical scaling (using a bigger instance) rather than horizontal scaling (spreading your data across multiple, smaller instances). Vertical scaling is generally more expensive, and you'll eventually run into limits as you hit the maximum size of a single instance.

You won't find information about joins in the DynamoDB documentation because there are no joins in DynamoDB. Joins are inefficient at scale, and DynamoDB is built for scale. Rather than reassembling your data at read time with a join, you should preassemble your data in the exact shape that is needed for a read operation.

One last note—people often try to do joins in their application code. They'll make an initial request to DynamoDB to fetch a record, then make a follow-up request to query all related records for that record. You should avoid this pattern. DynamoDB doesn't have joins for a reason!

In the next chapter, we'll take a look at how you can get join-like functionality with DynamoDB.

7.1.2. Normalization

Data normalization. First, second, and third normal form. You probably learned these concepts as you got started with relational databases. For those unfamiliar with the technical jargon, 'normalization' is basically the database version of the popular code mantra of "Don't Repeat Yourself" (or, "DRY"). If you have data that is duplicated across records in a table, you should split the record out into a separate table and refer to that record from your original table.

Let's walk through some basic principles of normalization.

Basics of Normalization

Normalization is on a spectrum, and you go to increasingly higher 'forms' of normalization. This goes from first normal form (1NF)

and second normal form (2NF) all the way to sixth normal form (6NF), with a few unnumbered forms like elementary key normal form (EKNF) in between.

The first three normal forms—1NF, 2NF, and 3NF—are the most commonly used ones in applications, so that's all we'll cover here.

Imagine you have an online clothing store. You might have a single, denormalized table that stores all of your items. In its denormalized form, it looks as follows:

Items					
Item	**Size**	**Price**	**Categories**	**ManufacturerId**	**ManufacturerName**
Just Do It T-Shirt	Women's Medium	29.99	Womens, T-shirt, Sportswear	1	Nike
Just Do It T-Shirt	Women's Large	34.99	Womens, T-shirt, Sportswear	1	Nike
Nebraska hat	One size fits all	9.99	Nebraska, Hats	2	Adidas

In our single, denormalized table, we have a few different items. Let's work to move this to third normal form.

First, we want to get to first normal form. To do that, all column values must be *atomic*, meaning they have a single value. Notice that the `Categories` column of our table includes multiple values, violating first normal form.

We can fix that using the following structure:

Items				
Item	**Size**	**Price**	**ManufacturerId**	**ManufacturerName**
Just Do It T-Shirt	Women's Medium	29.99	1	Nike
Just Do It T-Shirt	Women's Large	34.99	1	Nike
Nebraska hat	One size fits all	9.99	2	Adidas

ItemsCategories		
Item	**Size**	**CategoryId**
Just Do It T-Shirt	Women's Medium	1
Just Do It T-Shirt	Women's Medium	2
Just Do It T-Shirt	Women's Medium	3
Just Do It T-Shirt	Women's Large	1
Just Do It T-Shirt	Womens' Large	2
Just Do It T-Shirt	Women's Large	3
Nebraska hat	One size fits all	1
Nebraska hat	One size fits all	2

Categories	
CategoryId	**Category**
1	Womens
2	T-shirt
3	Sportswear
4	Nebraska
5	Hats

We no longer have a `Categories` column in our Items table. Rather, we've added two tables. First, we split Categories out into a separate table with two columns: `CategoryId` and `Category`. Then, we have a linking table for ItemsCategories, which maps each Item to its respective categories.

With the data split across three tables, we would need to use joins to pull it back together. The join to retrieve an item with its categories might look as follows:

```
SELECT *
FROM items
JOIN items_categories ON items.item = items_categories.item AND items.size =
items_categories.size
JOIN categories ON items_categories.category_id = categories.category_id
WHERE items.item = "Nebraska hat"
```

This would retrieve the item and all item categories for the Nebraska hat.

Our data is now in first normal form. Hurrah! Let's move on to second normal form.

In a database, each record must be uniquely identifiable by a key. This can be a single value or it can be a combination of values. In our Items table, our key is a combination of `Item` and `Size`. Note that `Item` itself would not be enough, our table has two records with the same value in the `Item` column.

Items				
Item	**Size**	**Price**	**ManufacturerId**	**ManufacturerName**
Just Do It T-Shirt	Women's Medium	29.99	1	Nike
Just Do It T-Shirt	Women's Large	34.99	1	Nike
Nebraska hat	One size fits all	9.99	2	Adidas

Key values

Non-key values

To achieve second normal form, all non-key values must depend on the whole key. Non-key values cannot depend on just part of the key.

This is a problem for our table as certain non-key values, such as `ManufacturerId` and `ManufacturerName` only depend on the `Item` column but not the `Size` column. Both "Just Do It" t-shirts are manufactured by Nike, regardless of the size.

To get into second normal form, we'll restructure as follows (note that the Categories tables have been removed for simplicity):

Items

Item	ManufacturerId	ManufacturerName
Just Do It T-Shirt	1	Nike
Nebraska hat	2	Adidas

ItemPrices

Item	Size	Price
Just Do It T-Shirt	Women's Medium	29.99
Just Do It T-Shirt	Women's Large	34.99
Nebraska hat	One size fits all	9.99

We have removed `Size` and `Price` columns from the Items table and moved them to an ItemsPrices table. Now our Items table's key is `Item`, and all non-key attributes in the Items table depend on the `Item`. The ItemsPrices table contains the various entries for sizes and prices.

Finally, to get to third normal form, there cannot be any transitive dependencies in the table. Basically this means that two columns on the table must be independent of each other.

Notice that we have two columns related to the manufacturer in our table--`ManufacturerId` and `ManufacturerName`. These columns aren't independent of each other. If one changes, so would the other.

Accordingly, let's split them out into a separate table to achieve third normal form:

Items

Item	ManufacturerId
Just Do It T-Shirt	1
Nebraska hat	2

Manufacturers

Id	ManufacturerName
1	Nike
2	Adidas
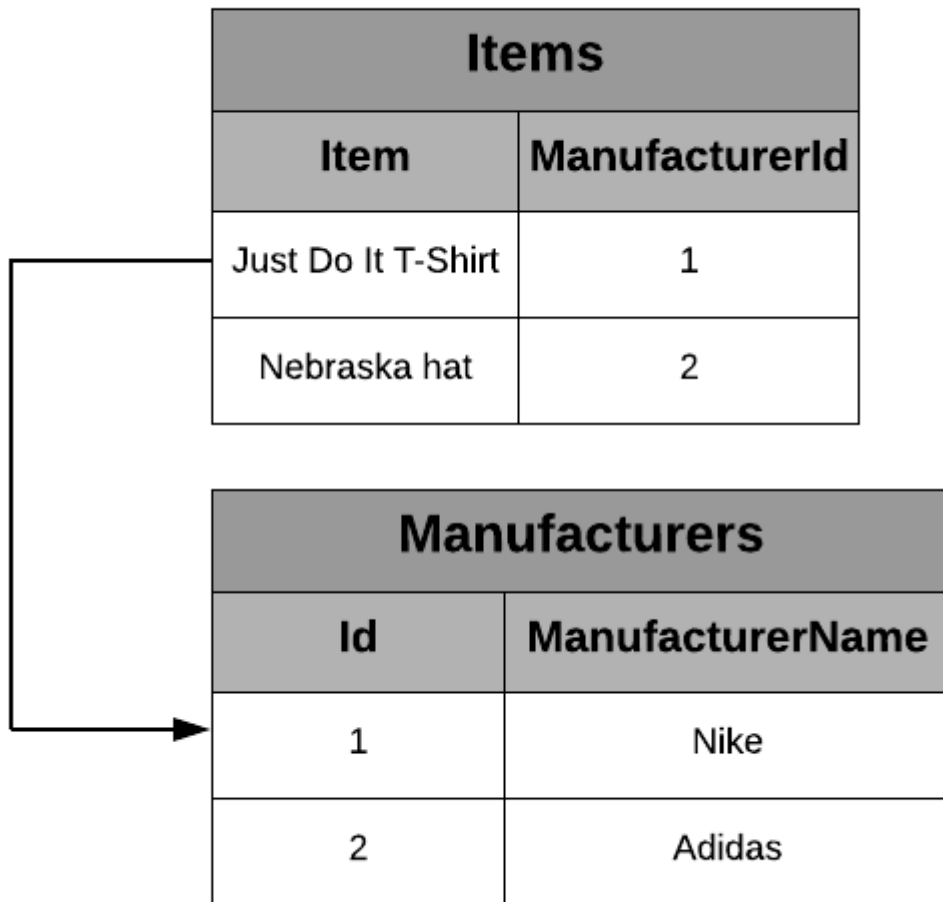
Now there are no transitive dependencies in our tables.

Let's put it all together to see our full data model in third normal form:

Items		
ItemId	**Item**	**ManufacturerId**
1	Just Do It T-Shirt	1
2	Nebraska hat	2

ItemPrices		
ItemId	**Size**	**Price**
1	Women's Medium	29.99
1	Women's Large	34.99
2	One size fits all	9.99

Manufacturers	
ItemId	**ManufacturerName**
1	Nike
2	Adidas

ItemsCategories	
Item	**CategoryId**
1	1
1	2
1	3
2	1
2	2

Categories	
CategoryId	**Category**
1	Womens
2	T-shirt
3	Sportswear
4	Nebraska
5	Hats

Rather than a single Items table, we now have five different tables. There is heavy use of IDs to link different tables together, and there's no duplication of data across rows or columns.

A basic summary of the first three forms of normalization is below.

Form	Definition	Plain english
First normal form (1NF)	Each column value is atomic	Don't include multiple values in a single attribute
Second normal form (2NF)	No partial dependencies	All non-key attributes must depend on the entirety of primary key
Third normal form (3NF)	No transitive dependencies	All non-key attributes depend on *only* the primary key

Table 3. Normalization forms

Now that we understand what normalization is, let's see why it's useful.

Benefits of Normalization

There are two reasons for normalization. One is related to technological limitations, and one is related to complexity in your code base.

The technological reason for normalization is due to the resource constraints of the time when relational databases were growing in popularity. In the 1970s and 1980s, storage was an expensive resource. As such, it made sense to reduce duplication of data by only writing it once and then having other records point to it. You could use joins to reassemble the disaggregated record.

The second reason for normalization is that it helps to maintain data integrity over time. If you have a piece of data duplicated across multiple records in a table, you'll need to make sure you update every record in that table in the event of a change. In contrast, with a normalized database, you can update the record in one place and all records that refer to it will get the benefits of the update.

For example, think of our clothing store example before. Imagine we wanted to put the "Just Do It" shirt on clearance and thus wanted to add a "Clearance" value in the `Categories` column. In our initial, denormalized example, we would have to find all records of the Just Do It t-shirt and add the value to them. If we did this wrong, it's possible we would have inconsistent data across the different sizes of the t-shirt.

With a normalized table, we only need to add a single record to the ItemsCategories linking table, and all sizes of the Just Do It t-shirt will get the update.

Why denormalize with DynamoDB

Normalization requires joins, as you need to reassemble your normalized records. However, we just discussed that DynamoDB doesn't have joins. Because of this, you need to get comfortable with denormalizing your data.

Let's review the two reasons for normalization in the context of modern applications.

The first reason for normalization, that of conserving on storage, is no longer applicable. Storage is cheap and plentiful. As of the time of writing, I can get a GB of SSD storage for $0.10 per month on AWS. As we start to reach the limits of Moore's Law, compute has become the limiting factor in most cases. Given this, it makes sense to optimize for compute by denormalizing your data and storing it together in the format needed by a read operation rather than optimizing for storage by normalizing and re-aggregating your data at query time.

The second reason for normalization—maintaining data integrity—is still a factor, and it's one you will need to consider as you model your data. Data integrity is now an application concern rather than a database concern. Consider where and when your data will be updated and how you will identify the associated records that need to be updated as part of it. In most applications, this won't be a big problem, but it can add significant complexity in certain situations.

7.1.3. Multiple entity types per table

There's one more implication of the lack of joins and the subsequent need to denormalize. In DynamoDB, you will have multiple different types of entities in a single table.

In a relational database, each entity type is put into a different table. Customers will be in one table, CustomerOrders will be in another table, and InventoryItems will be in a third table. Each table has a defined set of columns that are present on every record in the table—Customers have names, dates of birth, and an email address, while CustomerOrders have an order date, delivery date, and total cost.

There are no joins in DynamoDB, and we also saw that you shouldn't fake joins in your application code by making multiple serial requests to your DynamoDB table. So how do you solve a complex access pattern in a single request, such as fetching a customer and all customer orders?

You put multiple entity types in a single table. Rather than having a Customers table and a CustomerOrders table, your application will have a single table that includes both Customers and CustomerOrders (and likely other entities as well). Then, you design your primary key such that you can fetch both a Customer and the matching CustomerOrders in a single request.

There are a few implications of this approach that will feel odd coming from a relational background.

First, the primary keys will contain different information depending on the type of entity. As such, you probably won't be able to give descriptive names to your primary key attributes. In our example, you wouldn't name the partition key `CustomerId` as it may be the CustomerId for the Customer entity but it may show the `OrderId` for the CustomerOrder entity. As such, you'll likely use more generic names for your primary key attributes, such as `PK` for the partition key and `SK` for the sort key.

Second, the attributes on your records will vary. You can't count on each item in a table having the same attributes, as you can in a relational database. The attributes on a Customer are significantly

different than the attributes on a CustomerOrder. This isn't much of an issue for your application code, as the low-level details should be abstracted away in your data access logic. However, it can add some complexity when browsing your data in the console or when exporting your table to an external system for analytical processing.

We'll take a deeper look at the what and why behind single-table design in the next chapter.

7.1.4. Filtering

Data access is one large filtering problem. You rarely want to retrieve all data records at a time. Rather, you want the specific record or records that match a set of conditions.

Filtering with relational databases is glorious. You can specify any condition you want using the magical `WHERE` clause. You can filter on properties of top-level objects or properties of nested objects that have been joined to your data. You can filter using dynamic properties, such as the current time of the request. The `WHERE` clause is supremely powerful and a huge benefit to relational databases.

But, like other aspects of a relational database, the flexibility of the `WHERE` clause is a luxury you can't afford at scale. A `WHERE` clause needs to read and discard a large number of records, and this is wasted compute.

Filtering with DynamoDB is much more limited and, as a result, much more performant. With DynamoDB, filtering is built directly into your data model. The primary keys of your table and your secondary indexes determine how you retrieve data. Rather than arbitrarily filtering on any attribute in your record, you use precise, surgical requests to fetch the exact data you need.

The DynamoDB approach requires more planning and work

upfront but will provide long-term benefits as it scales. This is how DynamoDB is able to provide consistently fast performance as you scale. The sub-10 millisecond response times you get when you have 1 gigabyte of data is the same response time you get as you scale to a terabyte of data and beyond. The same cannot be said about relational databases.

7.2. Steps for Modeling with DynamoDB

In this section, you will learn the steps for modeling with DynamoDB. At a high level, these steps are:

- Understand your application

- Create an entity-relationship diagram ("ERD")

- Write out all of your access patterns

- Model your primary key structure

- Satisfy additional access patterns with secondary indexes and streams

Data modeling in DynamoDB is driven entirely by your access patterns. You will not be able to model your data in a generic way that allows for flexible access in the future. You must shape your data to fit the access patterns.

This process can take some time to learn, particularly for those that are used to modeling data in relational, SQL-based systems. Even for experienced DynamoDB users, modeling your data can take a few iterations as you work to create a lean, efficient table.

Let's walk through the process below.

7.2.1. Create an entity-relationship diagram (ERD)

The first step in your data modeling process is to create an entity-relationship diagram, or ERD. If you got a CS degree in college, ERDs may be old hat to you. If you didn't get a CS degree, don't worry—neither did I! ERDs are learnable and will make it easier for you to think about your data.

An entity-relationship diagram is just like it sounds—a diagram that lists the different entities in your application and how they relate to each other. There's a lot of techno-speak in there, so let's take a look at it by way of an example.

Imagine you have a "Notes" application. Users sign up for your application, and they can save notes. A note has a title, a date when it was created, and a body, which contains the content of the note.

The ERD for your Notes application would look like this:
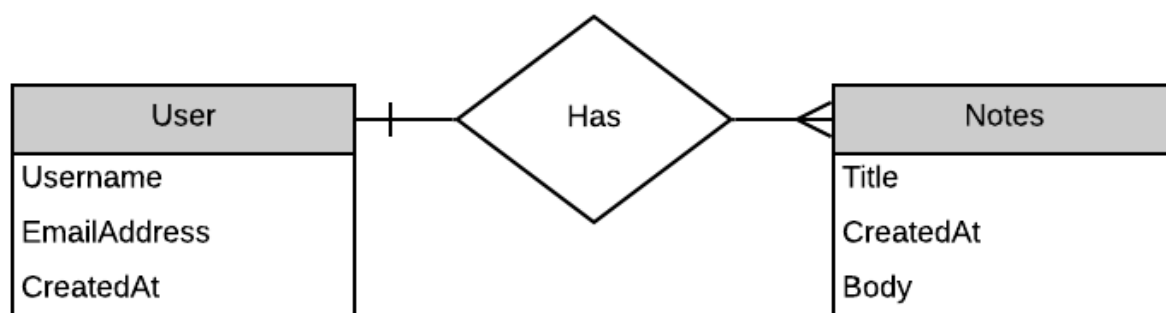
First, note that there are two rectangular boxes, titled "Users" and "Notes". These are the entities in your application. Usually, these are the nouns you use when talking about your application—Users, Notes, Orders, Organizations, etc. Note that each entity lists a number of attributes for that entity. Our User entity has a username, email address, and date created, while the Note entity has a title, a date created, and a body.

Second, notice that there is a diamond with some lines connecting the User and Note entities. This indicates a relationship between the two entities. The line shown indicates a one-to-many relationship, as one User can own many notes.

The Notes application above is an application with a small data model. You will likely have a larger data model with a much larger ERD. The ERD below is from the GitHub data modeling example in Chapter 21:
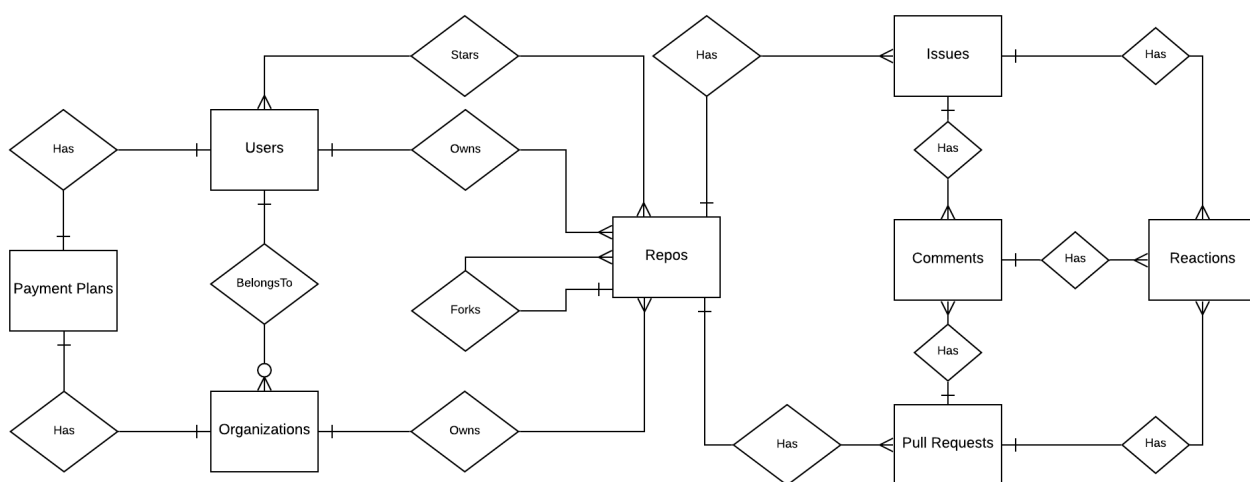
In this ERD, there are eight different entities with fourteen relationships between them. Even with a data model like this with a large number of relationships, you can model this with a single table in DynamoDB. Read the full example in Chapter 21 to learn more.

Even if you have a small data model, I still recommend building an ERD. It forces you to think about your data upfront and provides a nice artifact for those that are new to your application.

7.2.2. Define your access patterns

When using a relational database, you can usually just ship your ERD straight to the database. Your entities become tables, and your

relationships are configured via foreign keys. You design your data in a way that can accomodate flexible query patterns in the future.

This is not the case when data modeling in DynamoDB. You design your data to handle the specific access patterns you have, rather than designing for flexibility in the future.

Thus, your next step after creating an ERD is to define your data access patterns. All of them. Be specific and thorough. Failure to do this correctly may lead to problems down the line as you find your DynamoDB table isn't as flexible to new patterns as your relational database was.

So talk with your PM, engineering manager, business analyst, and other stakeholders in your application. Make sure you know all of the needs you have before you start designing your table.

There are two different strategies you can use for building these access patterns. One is the API-centric approach, which is common if you're planning to implement a REST API. With the API-centric approach, you list out each of the API endpoints you want to support in your application, as well as the expected shape you would like to return in your response.

The second approach for building your access patterns is the UI-centric approach. This is better if you're doing more server-side rendering or if you're using a 'backends-for-frontends' approach to an API. With the UI-centric approach, you look at each of the screens in your application and the URLs that will match those screens. As you do so, identify the different bits of information you need to assemble to build out the screen. Jot those down, and those become your access patterns.

As you gather your requirements, I recommend listing them out in a chart as shown below.

Entity	Access Pattern	Index	Parameters	Notes
Sessions	Create Session			
	Get Session			
	Delete Session (time-based)			
	Delete Session (manual)			

Table 4. Session store access patterns

In the left-hand side of the chart, describe the access pattern you have. If you have unique needs around an access pattern, be sure to list those in the `Notes` column.

As you design your data model, you will fill in the right-hand side of your chart. This column describes the DynamoDB API call you will use and any details about the call—the table or index you use, the parameters used in your API call, and any notes.

I cannot express strongly enough how important this step is. You can handle almost any data model with DynamoDB provided that you design for your access patterns up front. The biggest problem I see users face is failing to account for their patterns up front, then finding themselves stuck once their data model has solidified.

More often than not, these users blame DynamoDB for the problem, when the problem was with how the person used the tool. If you tried to use a screwdriver to rake leaves, would you say the screwdriver is a useless tool, or would you say you used it for the wrong job?

7.2.3. Model your primary key structure

Once you have defined all of your access patterns, it's time to dive in and start modeling your data.

The primary key is the foundation of your table, so you should start there. I model my primary key using the following four steps.

First, I create an 'entity chart' that is used to track the different types of items I'll have in my table. The entity chart tracks the type of item, the primary key structure for each item, and any additional notes and properties.

I usually start by copying each entity in my ERD into the entity chart. Below is the entity chart I started with for the GitHub example in Chapter 21, after I created the ERD:

Entity	PK	SK
Repo		
Issue		
Pull Request		
Fork		
Comment		
Reaction		
User		
Organization		
Payment Plan		

Table 5. GitHub model entity chart

As you build out your data model, the rows in your entity chart may change. You may remove entities that appear in your ERD because they aren't tracked as separate items in DynamoDB. For example, you may use a list or map attribute type to represent related objects on a particular item. This denormalized approach means there's not a separate item for each of the related entities. You can see more about this strategy in Chapter 11 on one-to-many relationship strategies.

Additionally, you may need to add entities to your chart as you design your table. A common example here is with a many-to-

many relationship where you need to add an item type to represent the relationship between entities. You could also add an item type solely to ensure uniqueness on a particular attribute for an item type. You can see the many-to-many example in the GitHub example in Chapter 21, and you can see the uniqueness pattern in the e-commerce example in Chapter 19.

After deciding the entities and relationships to model, the second step is to decide on a simple or composite primary key. Most complex data models use a composite primary key but there are exceptions. In general, a rule of thumb is that if you have any access patterns that require retrieving multiple entities (e.g., Get all Orders for a User) or entity types (e.g., Get a Sensor and the most recent SensorReadings for the Sensor), you will need a composite primary key.

The last step is to start designing the primary key format for each entity type. Make sure you satisfy the uniqueness requirements first. If you have some additional flexibility in your key design after handling uniqueness, try to solve some of the "fetch many" access patterns you have.

There's no one-size-fits-all approach to designing your primary key, but there are a few principles you should keep in mind:

- **Consider what your client will know at read time**

Remember that your client must know the primary key at read time or otherwise make costly additional queries to figure out the primary key.

For example, if the URL to fetch a particular user is https://api.mydomain.com/users/alexdebrie, where `alexdebrie` is the username, you can safely include username in the primary key as the username will be available on the API request.

A common anti-pattern I see people use is to add a `CreatedAt` timestamp into their primary key. This will help to ensure the primary key for your item is unique, but will that timestamp be available when you need to retrieve or update that item? If not, use something that will be available or find how you will make this information available to the client.

- **Use primary key prefixes to distinguish between entity types**

We mentioned in the previous section that you will include multiple entity types per table. This can make it difficult to distinguish between the different types of entities, particularly when looking at your table in the console or exporting to analytics systems. Further, it can prevent accidental overlaps if you have different entity types with common attributes.

One thing I like to do is use a prefixing system on my primary keys to help identify the type of entity that is being referenced. For example, if I have Customers and CustomerOrders in a table, my pattern for Customers might be `CUSTOMER#<CustomerId>` for the partition key and `METADATA#<CustomerId>` on the sort key. For the CustomerOrder, the pattern might be `ORDER#<OrderId>` for the partition key and `METADATA#<OrderId>` for the sort key.

As you're modeling your entities, keep track of primary key templates in your entity chart as shown below:

Entity	PK	SK
Customer	`CUSTOMER#<CustomerId>`	`METADATA#<CustomerId>`
CustomerOrder	`ORDER#<OrderId>`	`METADATA#<OrderId>`

Table 6. E-commerce entity chart

This will help you maintain clarity as you build out your access patterns chart and will serve as a nice artifact for development and post-development.

When you're just starting out with data modeling in DynamoDB, it can be overwhelming to think about where to start. Resist the urge to give up. Dive in somewhere and start modeling. It will take a few iterations, even for experienced DynamoDB users.

As you gain experience, you'll get a feel for which areas in your ERD are going to be the trickiest to model and, thus, the places you should think about first. Once you have those modeled, the rest of the data model often falls into place.

7.2.4. Handle additional access patterns with secondary indexes and streams

Once you've modeled your primary keys, you should see a bunch of access patterns fall into place. And that's great! It's best to do as much as you can with your primary keys. You won't need to pay for additional throughput, and you won't need to consider eventual consistency issues that come with global secondary indexes.

But you won't always be able to model everything with your primary key. That's where you start thinking about secondary indexes. Secondary indexes are a powerful tool for enabling additional read patterns on your DynamoDB table.

New users often want to add a secondary index for each read pattern. This is overkill and will cost more. Instead, you can overload your secondary indexes just like you overload your primary key. Use generic attribute names like `GSI1PK` and `GSI1SK` for your secondary indexes and handle multiple access patterns within a single secondary index.

There are a number of ways to use secondary indexes in your application. Secondary indexes are used heavily in the strategies chapters (Chapters 10-16) as well as in the more advanced data models (Chapters 19-22).

7.3. Conclusion

In this chapter, we saw how to approach data modeling with DynamoDB. First, we took a look at some ways that data modeling with DynamoDB is different than data modeling with a relational database. Forget about joins, get comfortable with denormalization, and reduce the number of tables in your application.

Second, we saw a step-by-step strategy for modeling with DynamoDB. You cannot skip these steps and expect to be successful. Take the time to build out your ERD and figure out your access patterns. Once you've done that, then you can get to the fun part of designing your table.

In the next chapter, we're going to do a deep-dive into single-table design with DynamoDB.

Chapter 8. The What, Why, and When of Single-Table Design in DynamoDB

> ## Chapter Summary
>
> When modeling with DynamoDB, use as few tables as possible. Ideally, you can handle an entire application with a single table. This chapter discusses the reasons behind single-table design.
>
> ## Sections
>
> 1. What is single-table design
> 2. Why is single-table design needed
> 3. The downsides of single-table design
> 4. Two times where the downsides of single-table design outweigh the benefits

When moving from relational databases to DynamoDB, one of the biggest mental hurdles is getting used to single-table design. It has been ingrained deeply in our psyche that each type in our application gets its own table in the database. With DynamoDB, it's flipped. You should use as few tables as possible, ideally just a single table per application or microservice.

In this chapter, we'll do a deep dive on the concepts behind single-table design. You'll learn:

- What is single-table design
- Why is single-table design needed
- The downsides of single-table design
- Two times where the downsides of single-table design outweigh the benefits

Let's get started.

8.1. What is single-table design

Before we get too far, let's define single-table design. To do this, we'll take a quick journey through the history of databases. We'll look at some basic modeling in relational databases, then see why you need to model differently in DynamoDB. With this, we'll see the key reason for using single-table design.

At the end of this section, we'll also do a quick look at some other, smaller benefits of single-table design.

8.1.1. Background on SQL modeling & joins

Let's start with our good friend, the relational database.

With relational databases, you generally normalize your data by creating a table for each type of entity in your application. For example, if you're making an e-commerce application, you'll have one table for Customers and one table for Orders.

Customers		
CustomerId	**CustomerName**	**CustomerBirthdate**
741	Alex DeBrie	05/26/1988
742	Albert Einstein	03/14/1879

Orders		
OrderId	**CustomerId**	**OrderDate**
11578	741	12/20/2019
11579	910	12/21/2019

Each Order belongs to a certain Customer, and you use foreign keys to refer from a record in one table to a record in another. These foreign keys act as pointers—if I need more information about a Customer that placed a particular Order, I can follow the foreign key reference to retrieve items about the Customer.

To follow these pointers, the SQL language for querying relational databases has a concept of *joins*. Joins allow you to combine records from two or more tables at read-time.

8.1.2. The problem of missing joins in DynamoDB

While convenient, SQL joins are also expensive. They require scanning large portions of multiple tables in your relational database, comparing different values, and returning a result set.

DynamoDB was built for enormous, high-velocity use cases, such as the Amazon.com shopping cart. These use cases can't tolerate the inconsistency and slowing performance of joins as a dataset scales.

DynamoDB closely guards against any operations that won't scale, and there's not a great way to make relational joins scale. Rather than working to make joins scale better, DynamoDB sidesteps the problem by removing the ability to use joins at all.

But as an application developer, you still need some of the benefits of relational joins. And one of the big benefits of joins is the ability to get multiple, heterogeneous items from your database in a single request.

In our example above, we want to get both a Customer record and all Orders for the customer. Many developers apply relational design patterns with DynamoDB even though they don't have the relational tools like the join operation. This means they put their items into different tables according to their type. However, since there are no joins in DynamoDB, they'll need to make multiple, serial requests to fetch both the Orders and the Customer record.
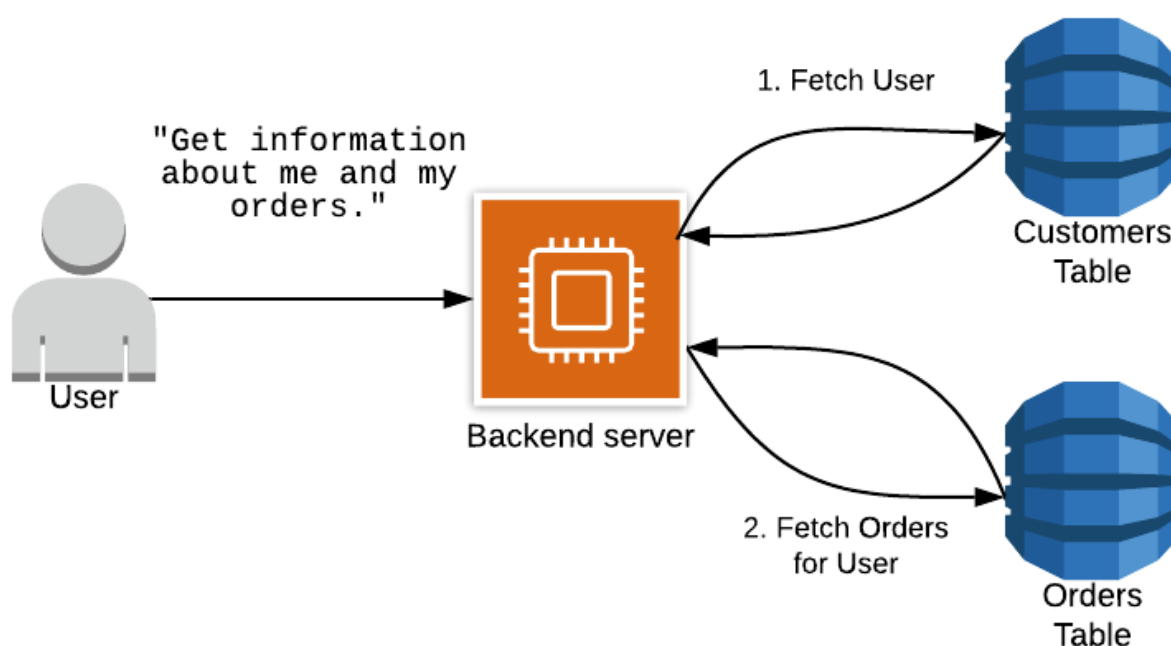
This can become a big issue in your application. Network I/O is likely the slowest part of your application, but now you're making multiple network requests in a waterfall fashion, where one request provides data that is used for subsequent requests. As your application scales, this pattern gets slower and slower.

8.1.3. The solution: pre-join your data into item collections

So how do you get fast, consistent performance from DynamoDB without making multiple requests to your database? By *pre-joining* your data using item collections.

An item collection in DynamoDB refers to all the items in a table or index that share a partition key. In the example below, we have a DynamoDB table that contains actors and the movies in which they have played. The primary key is a composite primary key where the partition key is the actor's name and the sort key is the movie name.

Primary Key		Attributes		
Actor (PARTITION)	**Movie (SORT)**			
Tom Hanks	Cast Away	**Role**	**Year**	**Genre**
		Chuck Noland	2000	Drama
	Toy Story	**Role**	**Year**	**Genre**
		Woody	1995	Children's
Tim Allen	Toy Story	**Role**	**Year**	**Genre**
		Buzz Lightyear	1995	Children's
Natalie Portman	Black Swan	**Role**	**Year**	**Genre**
		Nina Sayers	2010	Drama

You can see there are two items for Tom Hanks—Cast Away and Toy Story. Because they have the same partition key of `Tom Hanks`, they are in the same item collection.

You can use DynamoDB's Query API operation to read multiple items with the same partition key. Thus, if you need to retrieve multiple heterogeneous items in a single request, you organize those items so that they are in the same item collection.

See Chapter 19 for an example of this in action. This example uses an e-commerce application which involves Users and Orders. We have an access pattern where we want to fetch the User record and the Order records. To make this possible in a single request, we

make sure all Order records live in the same item collection as the User record to which they belong.

Primary Key		Attributes				
PK	**SK**					
USER#alexdebrie	#PROFILE#alexdebrie	**Username**	**FullName**	**Email**	**CreatedAt**	**Addresses**
		alexdebrie	Alex DeBrie	alexdebrie1@gmail.com	03/23/2018	{"Home":{"StreetAddress":"1111 1st St","State":"Nebr
	ORDER#5e7272b7	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		alexdebrie	5e7272b7	PLACED	04/21/2019	{"StreetAddress":"1111 1st St","State":"Nebraska","Cc
	ORDER#42ef295e	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		alexdebrie	42ef295e	PLACED	04/25/2019	{"StreetAddress":"1111 1st St","State":"Nebraska","Cc
	ORDER#2e7abecc	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		alexdebrie	2e7abecc	SHIPPED	12/25/2018	{"StreetAddress":"1111 1st St","State":"Nebraska","Cc
USER#nedstark	#PROFILE#nedstark	**Username**	**FullName**	**Email**	**CreatedAt**	**Addresses**
		nedstark	Eddard Stark	lord@winterfell.com	02/27/2016	{"Home":{"StreetAddress":"1234 2nd Ave","City":"Wir
	ORDER#2eae1dee	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		nedstark	2eae1dee	SHIPPED	01/15/2019	{"StreetAddress":"Suite 200, Red Keep","City":"King's L
	ORDER#f4f80a91	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		nedstark	f4f80a91	PLACED	05/12/2019	{"StreetAddress":"Suite 200, Red Keep","City":"King's L

Now when we want to fetch the User and Orders, we can do it in a single request without needing a costly join operation:
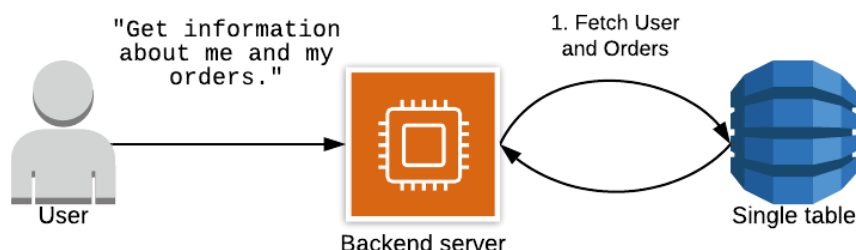
This is what single-table design is all about—tuning your table so that your access patterns can be handled with as few requests to DynamoDB as possible, ideally one.

8.1.4. Other benefits of single-table design

While reducing the number of requests for an access pattern is the main reason for using a single-table design with DynamoDB, there are some other benefits as well. I will discuss those briefly.

150

First, there is some operational overhead with each table you have in DynamoDB. Even though DynamoDB is fully-managed and pretty hands-off compared to a relational database, you still need to configure alarms, monitor metrics, etc. If you have one table with all items in it rather than eight separate tables, you reduce the number of alarms and metrics to watch.

Second, having a single table can save you money as compared to having multiple tables. With each table you have, you need to provision read and write capacity units. Often you will do some back-of-the-envelope math on the traffic you expect, bump it up by X%, and convert it to RCUs and WCUs. If you have one or two entity types in your single table that are accessed much more frequently than the others, you can hide some of the extra capacity for less-frequently accessed items in the buffer for the other items.

While these two benefits are real, they're pretty marginal. The operations burden on DynamoDB is quite low, and the pricing will only save you a bit of money on the margins. Further, if you are using DynamoDB On-Demand pricing, you won't save *any* money by going to a multi-table design.

In general, when thinking about single-table design, the main benefit is the performance improvement by making a single request to retrieve all needed items.

8.2. Downsides of a single-table design

While the single-table pattern is powerful and ridiculously scalable, it doesn't come without costs. In this section, we'll review some of the downsides of a single-table design.

In my mind, there are three downsides of single-table design in

DynamoDB:

- The steep learning curve to understand single-table design;
- The inflexibility of adding new access patterns;
- The difficulty of exporting your tables for analytics.

Let's review each of these in turn.

8.2.1. The steep learning curve of single-table design

The biggest complaint I get from members of the community is around the *difficulty* of learning single-table design in DynamoDB.

A single, over-loaded DynamoDB table looks really weird compared to the clean, normalized tables of your relational database. It's hard to unlearn all the lessons you've learned over years of relational data modeling.

While I empathize with your concerns, I don't find this a sufficient excuse not to learn single-table design.

Software development is a continuous journey of learning, and you can't use the difficulty of learning new things as an excuse to use a new thing poorly. If you want the advantages of DynamoDB— infinite scalability, convenient connection model, and consistent performance—you need to take the time to learn how to use it.

8.2.2. The inflexibility of new access patterns

A second complaint about DynamoDB is the difficulty of accommodating new access patterns in a single-table design. This complaint has more validity.

When modeling a single-table design in DynamoDB, you start with your access patterns first. Think hard (and write down!) how you will access your data, then carefully model your table to satisfy those access patterns. When doing this, you will organize your items into collections such that each access pattern can be handled with as few requests as possible—ideally a single request.

Once you have your table modeled out, then you put it into action and write the code to implement it. Done properly, this will work great. Your application will be able to scale infinitely with no degradation in performance.

However, your table design is narrowly tailored for the exact purpose for which it has been designed. If your access patterns change because you're adding new objects or accessing multiple objects in different ways, you may need to do an ETL process to scan every item in your table and update with new attributes. This process isn't impossible, but it does add friction to your development process.

That said, migrations aren't to be feared. In Chapter 15, we discuss some migration strategies, and in Chapter 22, we implement some of those strategies in a real example.

8.2.3. The difficulty of analytics

DynamoDB is designed for on-line transactional processing (OLTP) use cases—high speed, high velocity data access where you're operating on a few records at a time. But users also have a need for on-line analytics processing (OLAP) access patterns—big, analytical queries over the entire dataset to find popular items, number of orders by day, or other insights.

DynamoDB is not good at OLAP queries. This is intentional. DynamoDB focuses on being ultra-performant at OLTP queries

and wants you to use other, purpose-built databases for OLAP. To do this, you'll need to get your data from DynamoDB into another system.

If you have a single table design, getting it into the proper format for an analytics system can be tricky. You've denormalized your data and twisted it into a pretzel that's designed to handle your exact use cases. Now you need to unwind that table and re-normalize it so that it's useful for analytics.

My favorite quote on this comes from Forrest Brazeal's excellent walkthrough on single-table design:

> [A] well-optimized single-table DynamoDB layout looks more like machine code than a simple spreadsheet

— Forrest Brazeal

Spreadsheets are easy for analytics, whereas a single-table design takes some work to unwind. Your data infrastructure work will need to be pushed forward in your development process to make sure you can reconstitute your table in an analytics-friendly way.

8.3. When not to use single-table design

So far, we know the pros and cons of single-table design in DynamoDB. Now it's time to get to the more controversial part—when, if ever, should you *not* use single-table design in DynamoDB?

At a basic level, the answer is "whenever the benefits don't outweigh the costs." But that generic answer doesn't help us much. The more concrete answer is "whenever I need query flexibility and/or easier analytics more than I need blazing fast performance." And I think there are two occasions where this is most likely:

- in new applications where developer agility is more important than application performance;
- in applications using GraphQL.

We'll explore each of these below. But first I want to emphasize that these are exceptions, not general guidance. When modeling with DynamoDB, you should be following best practices. This includes denormalization, single-table design, and other proper NoSQL modeling principles. And even if you opt into a multi-table design, you should understand single-table design to know why it's not a good fit for your specific application.

8.3.1. New applications that prioritize flexibility

In the past few years, many startups and enterprises are choosing to build on serverless compute like AWS Lambda for their applications. There are a number of benefits to the serverless model, from the ease of deployments to the painless scaling to the pay-per-use pricing model.

Many of these applications use DynamoDB as their database because of the way it fits seamlessly with the serverless model. From provisioning to pricing to permissions to the connection model, DynamoDB is a perfect fit with serverless applications, whereas traditional relational databases are more problematic.

However, it's important to remember that while DynamoDB works great *with* serverless, it was not built *for* serverless.

DynamoDB was built for large-scale, high-velocity applications that were outscaling the capabilities of relational databases. And relational databases can scale pretty darn far! If you're in the situation where you're out-scaling a relational database, you probably have a good sense of the access patterns you need. But if you're making a greenfield application at a startup, it's unlikely you

absolutely require the scaling capabilities of DynamoDB to start, and you may not know how your application will evolve over time.

In this situation, you may decide that the performance characteristics of a single-table design are not worth the loss of flexibility and more difficult analytics. You may opt for a Faux-SQL approach where you use DynamoDB but in a relational way by normalizing your data across multiple tables.

This means you may need to make multiple, serial calls to DynamoDB to satisfy your access patterns. Your application may look as follows:
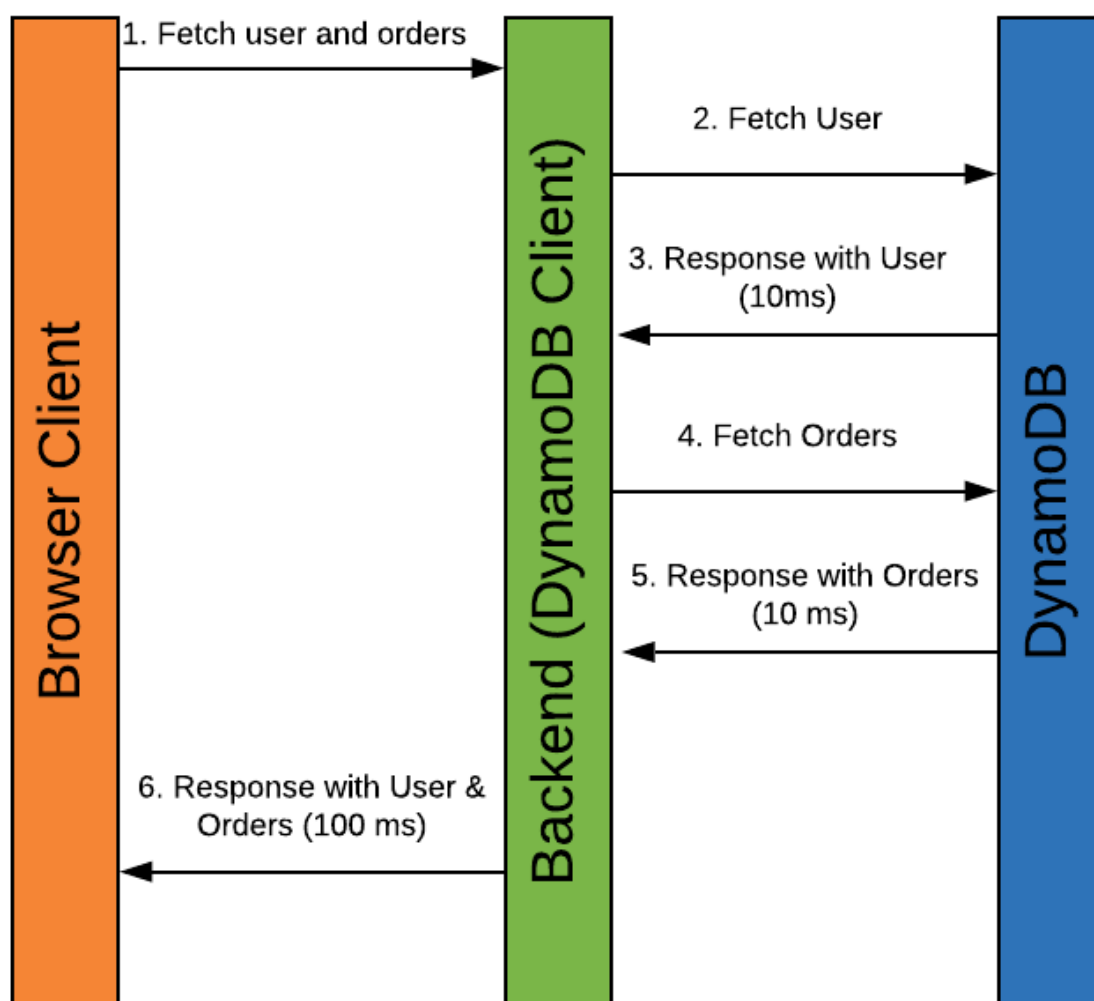
Notice how there are two separate requests to DynamoDB. First, there's a request to fetch the User, then there's a follow up request to fetch the Orders for the given User. Because multiple requests must be made and these requests must be made serially, there's going to be a slower response time for clients of your backend application.

For some use cases, this may be acceptable. Not all applications need to have sub-30ms response times. If your application is fine with 100ms response times, the increased flexibility and easier analytics for early-stage use cases might be worth the slower performance.

8.3.2. GraphQL & Single-table design

The second place where you may want to avoid single-table design with DynamoDB is in GraphQL applications.

Before I get 'Well, actually'-d to death on this one, I want to clarify that yes, I know GraphQL is an execution engine rather than a query language for a specific database. And yes, I know that GraphQL is database agnostic.

My point is not that you *cannot* use a single-table design with GraphQL. I'm saying that because of the way GraphQL's execution works, you're losing most of the benefits of a single-table design while still inheriting all of the costs.

To understand why, let's take a look at how GraphQL works and one of the main problems it aims to solve.

For the past few years, many applications have opted for a REST-based API on the backend and a single-page application on the frontend. It might look as follows:
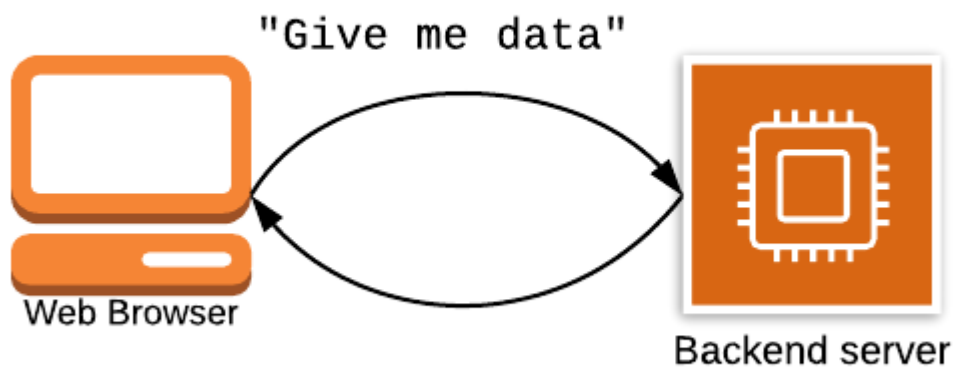
In a REST-based API, you have different *resources* which generally map to an entity in your application, such as Users or Orders. You can perform CRUD-like (Create, Read, Update, Delete) operations on these resources by using different HTTP verbs, like GET, POST, or PUT, to indicate the operation you want to perform.

One common source of frustration for frontend developers when using REST-based APIs is that they may need to make multiple requests to different endpoints to fetch all the data for a given page:
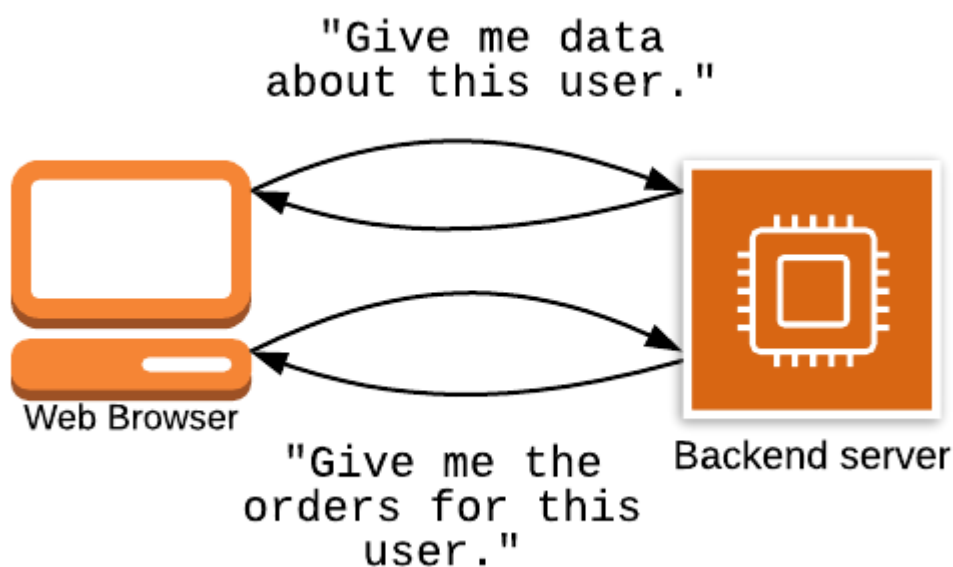
In the example above, the client has to make two requests—one to get the User, and one to get the most recent Orders for a user.

With GraphQL, you can fetch all the data you need for a page in a single request. For example, you might have a GraphQL query that looks as follows:

```
query { User( id:112233 ){
    firstName
    lastName
    addresses
    orders {
      orderDate
      amount
      status
    }
  }
}
```

In the block above, we're making a query to fetch the User with id 112233, then we're fetching certain attributes about the user (including firstName, lastName, and addresses), as well as all of the orders that are owned by that user.

Now our flow looks as follows:

"Please handle the query below."

Web Browser
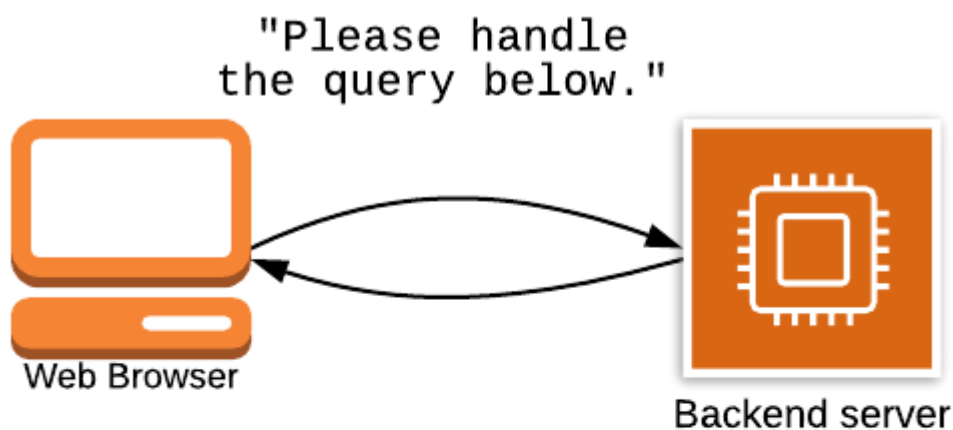
Backend server

```
query {
  User( id:112233 ){
      firstName
      lastName
      addresses
      orders {
        orderDate
        amount
        status
      }
    }
}
```
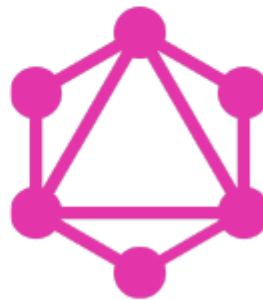
The web browser makes a single request to our backend server. The contents of that request will be our GraphQL query, as shown below the server. The GraphQL implementation will parse the query and handle it.

This looks like a win—our client is only making a single request to the backend! Hurrah!

In a way, this mirrors our discussion earlier about why you want to use single-table design with DynamoDB. We only want to make a single request to DynamoDB to fetch heterogeneous items, just like the frontend wants to make a single request to the backend to fetch

heterogeneous resources. It sounds like a match made in heaven.

The issue is in *how* GraphQL handles those resources in the backend. Each field on each type in your GraphQL schema is handled by a resolver. This resolver understands how to fill in the data for the field.

For different types in your query, such as `User` and `Order` in our example, you would usually have a resolver that would make a database request to resolve the value. The resolver would be given some arguments to indicate which instances of that type should be fetched, and then the resolver will fetch and return the data.

The problem is that resolvers are essentially independent from each other. In the example above, the root resolver would execute first to find the User with ID 112233. This would involve a query to the database. Then, once that data is available, it would be passed to the Order resolver in order to fetch the relevant Orders for this User. This would make subsequent requests to the database to resolve those entities.

Now our flow looks like this:

```
"Please handle
the query below."
```

Web Browser   Backend server

```
query {
    User( id:112233 ){
        firstName
        lastName
        addresses
    orders {
        orderDate
        amount
        status
    }
  }
}
```

1. Resolve User

2. Resolve Orders
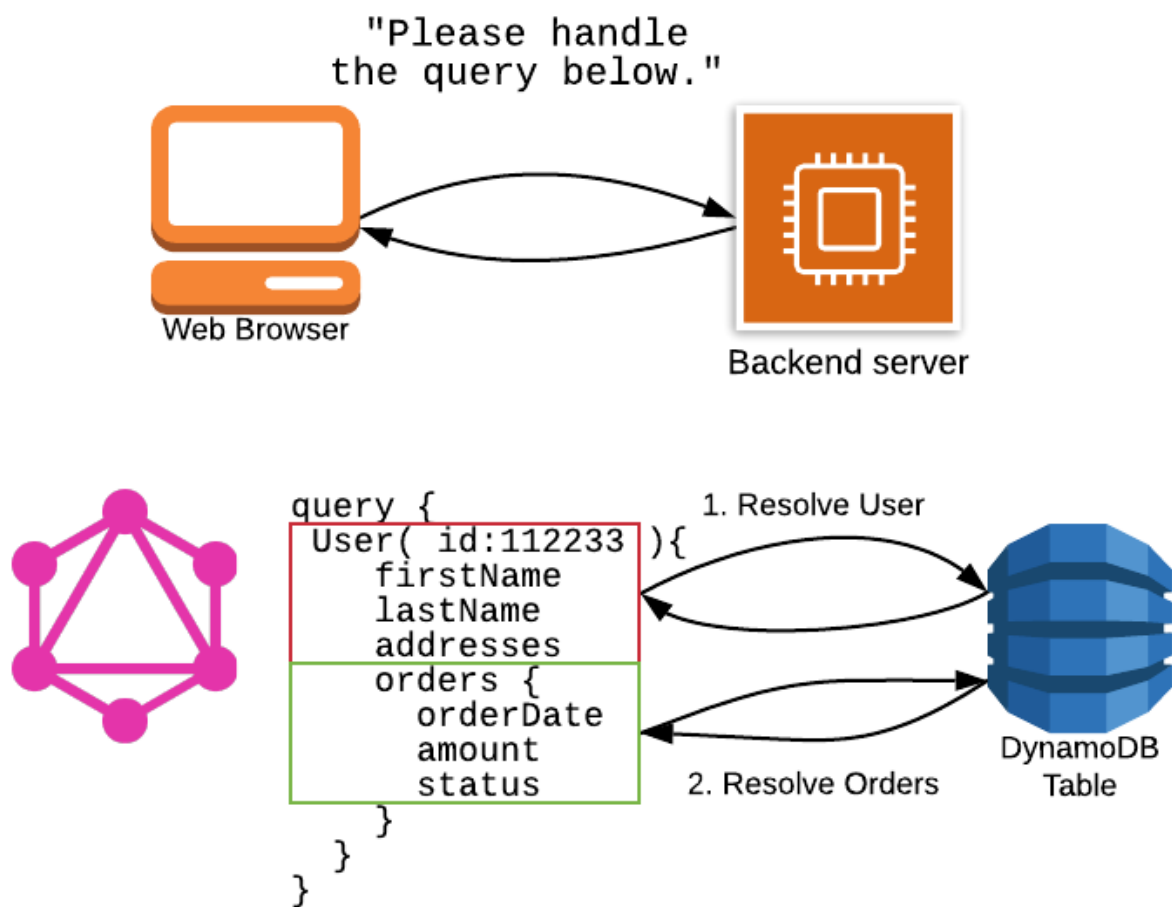
DynamoDB Table

In this flow, our backend is making multiple, serial requests to DynamoDB to fulfill our access pattern. This is exactly what we're trying to avoid with single-table design!

None of this goes to say that you can't use DynamoDB with GraphQL—you absolutely can. I just think it's a waste to spend time on a single-table design when using GraphQL with DynamoDB. Because GraphQL entities are resolved separately, I think it's fine to model each entity in a separate table. It will allow for more flexibility and make it easier for analytics purposes going forward.

8.4. Conclusion

In this chapter, we reviewed the concept of single-table design in DynamoDB. First, we went through some history on how NoSQL & DynamoDB evolved and why single-table design is necessary.

Second, we looked at some downsides to single-table design in DynamoDB. Specifically, we saw how single-table design can make it harder to evolve your access patterns and complicates your analytics work.

Finally, we looked at two situations where the benefits of single-table design in DynamoDB may not outweigh the costs. The first situation is in new, fast-evolving applications using serverless compute where developer agility is paramount. The second situation is when using GraphQL due to the way the GraphQL execution flow works.

I'm still a strong proponent of single-table design in DynamoDB in most use cases. And even if you don't think it's right for your situation, I still think you should learn and understand single-table design before opting out of it. Basic NoSQL design principles will help you even if you don't follow the best practices for large-scale design.

Chapter 9. From modeling to implementation

Chapter Summary

This chapter reviews converting your DynamoDB data model from concept to code. It includes tips for a successful, efficient, and extendable implementation.

Sections

1. Separate application attributes from indexing attributes
2. Implement your data model at the very boundary of your application
3. Don't reuse attributes across multiple indexes
4. Add a "Type" attribute to every item
5. Write scripts to help debug access patterns
6. Shorten attribute names to save storage

This book is heavily focused on data modeling with DynamoDB. And with good reason! Because you need to model your DynamoDB table to specifically handle your access patterns, you must be intentional with your data modeling. 90% of the work of using DynamoDB happens in the planning stage, before you write a single line of code.

At some point, however, you need to move from model to implementation. This chapter includes guidance on how to

implement your DynamoDB data model in your application code.

9.1. Separate application attributes from your indexing attributes

In Chapter 3, we first discussed the concept of overloading the primary key and secondary indexes in your DynamoDB table. To overload your primary key, you use generic names like `PK` and `SK` for your primary key attributes. The values for these generic attributes are specifically designed to create the proper item collections and ordering to handle your queries.

Thus, if you print out an item from DynamoDB, it might look as follows:

```python
response = client.get_item(**kwargs)
print(response['Item'])
"""
{
    "PK": { "S": "USER#alexdebrie" },
    "SK": { "S": "USER#alexdebrie" },
    "GSI1PK": { "S": "ORG#facebook" },
    "GSI1SK": { "S": "USER#alexdebrie" },
    "Username": { "S": "alexdebrie" },
    "FirstName": { "S": "Alex" },
    "LastName": { "S": "DeBrie" },
    "OrganizationName": { "S": "Facebook" },
    ...
}
"""
```

Notice that the first four attributes are all related to my DynamoDB data model but have no meaning in my application business logic. I refer to these as 'indexing attributes', as they're only there for indexing your data in DynamoDB. Additionally, the next three items are properties that are actually useful in my application—attributes like `Username`, `FirstName`, etc.

I advise you to keep a separation between these two kinds of attributes. Your application attributes will often inform parts of your indexing attributes. Here, the `Username` attribute is used to fill in pieces of the `PK`, `SK`, and `GSI1SK` templates.

However, I recommend against going the other way. Don't think that you can remove the `Username` attribute from your item since it's already encoded into your `PK`. It adds complexity and risks data loss if you change your data model and indexing attributes in the future. It will result in slightly larger item sizes due to duplicated data, but I think it's worth it.

9.2. Implement your data model at the very boundary of your application

In the previous example, we printed out one of the User items that may be saved in our DynamoDB table. You'll notice there are two things that are odd about that item. First, it includes the indexing attributes that we just discussed. Second, each attribute value is a map with a single key and value indicating the DynamoDB type and the value for the attribute.

Both of these facts will make it messy to work with DynamoDB items in the business logic portion of your application. You'll be grabbing nested properties, performing proper type conversions, and adding indexing attributes before saving your items.

To avoid this problem, implement your data model at the boundary of your application. Within the core of your application, your application objects should have the main attributes that are relevant to your business logic. That will look something like this:

```python
user = data.get_user(username='alexdebrie')
print(user)
# User(username="alexdebrie", first_name="Alex", ...)
```

In this brief snippet, the core of my application is retrieving a user by its username. The returned value from this function is a User object that is meaningful and useful to my application.

Let's take a look at how we might implement the `data.get_user()` function:

```python
def get_user(username):
    resp = client.get_item(
        TableName='AppTable',
        Key={ 'PK': { 'S': f'USER#{username}' }}
    )
    return User(
        username=resp['Item']['Username']['S'],
        first_name=resp['Item']['FirstName']['S'],
        last_name=resp['Item']['LastName']['S'],
    )
```

When given the username, the `get_user()` function knows how to construct the key for the GetItem API call. Then, it turns the assembled item into the User object that can be used in the application.

All interaction with DynamoDB should be handled in the `data` module that is at the boundary of your application. There's a lot of work to reshape your application object into the format needed by DynamoDB, and it's not something that the core of your application should care about. Write that DynamoDB logic once, at the edge of your application, and operate on application objects the rest of the time.

9.3. Don't reuse attributes across multiple indexes

The next tip I have is partly a data modeling tip but it fits well here.

As discussed above, you will have *indexing attributes* that are solely for properly indexing your data in DynamoDB. Let's print out our User object from above to see those attributes again.

```python
response = client.get_item(**kwargs)
print(response['Item'])
"""
{
    "PK": { "S": "USER#alexdebrie" },
    "SK": { "S": "USER#alexdebrie" },
    "GSI1PK": { "S": "ORG#facebook" },
    "GSI1SK": { "S": "USER#alexdebrie" },
    "Username": { "S": "alexdebrie" },
    "FirstName": { "S": "Alex" },
    "LastName": { "S": "DeBrie" },
    "OrganizationName": { "S": "Facebook" },
    ...
}
"""
```

When you do this, you may notice that `SK` and `GSI1SK` are the same value. And because they're the same value, you may be tempted to skip adding `GSI1SK` altogether and make your `GSI1` index use a key schema of `GSI1PK` and `SK`.

Don't do this.

While you are saving some money in storage by not duplicating attributes, it will make your data modeling more difficult. If you have multiple entity types in your application, you'll tie yourself in knots trying to make the attributes work across multiple different indexes. Further, if you do need to add additional access patterns or migrate data, this will make it more difficult.

Save yourself the pain. For each global secondary index you use, give it a generic name of `GSI<Number>`. Then, use `GSI<Number>PK` and `GSI<Number>SK` for your attribute types.

9.4. Add a 'Type' attribute to every item

As we've seen, your DynamoDB data model will include multiple types of items in a single table. You'll use different patterns for the primary key of each entity to distinguish between entity types. But it can be difficult to easily distinguish between this with a glance or when doing a filter expression.

One tip I suggest is to include a `Type` attribute on every item I write to the table. This attribute will be a simple string declaring the type of entity: User, Order, SensorReading, etc.

In your data access layer to interact with DynamoDB, make sure you include this `Type` attribute when writing to the table:

```python
def save_user(user: User):
    resp = client.put_item(
        TableName='AppTable',
        Item={
            'PK': { 'S': f'USER#{User.username}' },
            'SK': { 'S': f'USER#{User.username}' },
            'GSI1PK': { 'S': f'ORG#{User.org_name}' },
            'GSI1SK': { 'S': f'USER#{User.username}' },
            'Type': { 'S': 'User' },
            'Username': { 'S': User.username },
            'FirstName': { 'S': User.first_name},
            'LastName': { 'S': User.last_name},
            ...
        }
    )
    return user
```

I use this attribute for a few things. First, as mentioned, it makes it easier to orient myself if I am exploring my table in the AWS console. But it's more important for a few other situations.

In the migration strategies discussed in Chapter 15, we discuss that you may need to modify existing items to decorate them with new indexing attributes. To do that, you do a background ETL operation that scans your table, finds the items you need to modify, and adds the attributes.

When doing this, you usually only need to update certain entity types. I like to use a filter expression on the `Type` attribute when scanning my table to ensure I'm only getting the items that I need. It simplifies the logic in my ETL script.

A final reason to add this `Type` attribute is for analytics reasons when you export your data to an external system. DynamoDB is not great at performing ad-hoc OLAP-style queries, so you will probably import your data to Amazon Redshift or export it to Amazon S3 to query with Amazon Athena. But all your entities will be in a single table, which is not how a relational database expects to use your data. After your initial export of data, you'll want to "re-normalize" it by moving your different entity types into their own tables. Having a `Type` attribute makes it easier to write the transformation query and find the right items to move around.

9.5. Write scripts to help debug access patterns

Debugging DynamoDB applications can be difficult for developers. With single-table design, your items are all jumbled together in the same table. You're accessing items from DynamoDB via indexed attributes rather than the application attributes that you're used to in your application. Finally, you may be using shortened attribute names that require translation to map it back to your actual application attribute names.

Rather than use the DynamoDB console, I recommend writing little scripts that can be used to debug your access patterns. These scripts can be called via a command-line interface (CLI) in the terminal. A script should take the parameters required for an access pattern—a username or an order ID—and pass that to your data access code. Then it can print out the results.

A basic example is as follows:

```python
# scripts/get_user.py
import click

import data

@click.command()
@click.option('--username', help='Username of user to retrieve.')
def get_user(username):
    user = data.get_user(username)
    print(user)

if __name__ == '__main__':
    get_user()
```

Then you could run `python scripts/get_user.py --username alexdebrie`, and it would retrieve and print out the retrieved User.

For a simple access pattern to fetch a single item, it may not seem that helpful. However, if you're retrieving multiple related items from a global secondary index with complex conditions on the sort key, these little scripts can be lifesavers. Write them at the same time you're implementing your data model.

9.6. Shorten attribute names to save storage

Above, I mentioned you shouldn't reuse indexing attributes even if it would save you money due to the confusion it adds. However,

there is a different approach you can take to save on storage. This is a pretty advanced pattern that I would recommend only for the largest tables and for those that are heavily into the DynamoDB mindset.

Previously we discussed that all interaction with DynamoDB should be at the very boundary of your application, and it should be handled in a way that is specific to your data needs in DynamoDB.

In line with this, you can abbreviate your attribute names when saving items to DynamoDB to reduce storage costs. For example, imagine the following code to save a User in your application:

```python
def save_user(user: User):
    resp = client.put_item(
        TableName='AppTable',
        Item={
          'PK': { 'S': f'USER#{User.username}' },
          'SK': { 'S': f'USER#{User.username}' },
          'GSI1PK': { 'S': f'ORG#{User.org_name}' },
          'GSI1SK': { 'S': f'USER#{User.username}' },
          'u': { 'S': User.username },
          'fn': { 'S': User.first_name},
          'ln': { 'S': User.last_name},
          ...
        }
    )
    return user
```

This `save_user()` method takes a User object from your application and saves it to DynamoDB. Notice how the application attributes are abbreviated when calling the PutItem operation. The "Username" attribute name has been shortened to "u", and the "FirstName" attribute has been shortened to "fn".

Because your application will never be touching these abbreviated names, it's safe to make these abbreviations. When retrieving the User item from your table, you'll need to rehydrate your User object. That will convert the shortened attribute names to the more meaningful names to make it easy to use in your application.

Again, this is an advanced pattern. For the marginal application, the additional attribute names won't be a meaningful cost difference. However, if you plan on storing billions and trillions of items in DynamoDB, this can make a difference with storage.

9.7. Conclusion

When you're working with DynamoDB, the lion's share of the work will be upfront before you ever write a line of code. You'll build your ERD and design your entities to handle your access patterns as discussed in Chapter 7.

After you've done the hard design work, then you need to convert it to your application code. In this chapter, we reviewed some tips for handling that implementation. We saw how you should implement your DynamoDB code at the very boundary of your application. We also learned about conceptually separating your application attributes from your indexing attributes. Then we saw a few tips around saving your sanity later on by including a `Type` attribute on all items and writing scripts to help explore your data. Finally, we saw an advanced tip for saving costs on storage by abbreviating your attribute names.

In the next chapter, we're going to get started reviewing DynamoDB strategies, which is my favorite part of the book.

Chapter 10. The Importance of Strategies

We've covered a lot of ground already about the basics of DynamoDB, and the second half of this book contains a number of data modeling examples. There's no substitute for seeing a full example of a DynamoDB table in action to really grok how the modeling works.

Before we look at those examples, though, we're going to cover a concept that's essential when it comes to modeling well with DynamoDB—namely, *strategies*. In order to succeed with DynamoDB, it is crucial to understand and be able to apply a variety of strategies for modeling your data.

To illustrate what I mean by strategies, let's compare modeling with DynamoDB to how you approach modeling with a relational database.

When modeling for a relational database, there's basically one correct way to do something. Every problem has a straightforward answer:

- *Duplicating data?* Normalize it by putting it into a separate table.
- *One-to-many relationship?* Add a foreign key to indicate the relationship.
- *Many-to-many relationship?* Use a joining table to connect the two entities.

Data modeling with an RDBMS is like a science—if two people are modeling the same application, they should come up with very similar patterns.

With DynamoDB, on the other hand, there are multiple ways to approach the problem, and you need to use judgment as to which approach works best for your situation. DynamoDB modeling is more art than science—two people modeling the same application can have vastly different table designs.

Let's look at one example from above—modeling a one-to-many relationship. In the next chapter, we cover five different ways to model one-to-many relationships in DynamoDB:

- Denormalizing the data and storing the nested objects as a document attribute
- Denormalizing the data by duplicating it across multiple items
- Using a composite primary key
- Creating a secondary index
- Using a composite sort key to handle hierarchical data

As shown in the GitHub example in Chapter 21, we will often use more than one of these strategies in the same table.

Knowing these strategies is important for success in data modeling. As you gain experience, you'll learn to attack your application data model one access pattern at a time, searching for the right strategy to solve the immediate problem. Your model will come together like the pieces in a puzzle with each different strategy helping to achieve the ultimate goal.

Data modeling with DynamoDB requires flexibility, creativity, and persistence. It can be frustrating, but it can also be a lot of fun.

In the next six chapters, we'll look at some of the strategies you can use in modeling with DynamoDB.

Chapter 11. Strategies for one-to-many relationships

Chapter Summary

Your application objects will often have a parent-child relationship with each other. In this chapter, we'll see different approaches for modeling one-to-many relationships with DynamoDB.

Sections

1. Denormalization by using a complex attribute

2. Denormalization by duplicating data

3. Composite primary key + the Query API action

4. Secondary index + the Query API action

5. Composite sort keys with hierarchical data

A one-to-many relationship is when a particular object is the owner or source for a number of sub-objects. A few examples include:

- **Workplace:** A single office will have many employees working there; a single manager may have many direct reports.

- **E-commerce:** A single customer may make multiple orders over time; a single order may be comprised of multiple items.

- **Software-as-a-Service (SaaS) accounts:** An organization will purchase a SaaS subscription; multiple users will belong to one organization.

With one-to-many relationships, there's one core problem: how do I fetch information about the parent entity when retrieving one or more of the related entities?

In a relational database, there's essentially one way to do this—using a foreign key in one table to refer to a record in another table and using a SQL join at query time to combine the two tables.

There are no joins in DynamoDB. Instead, there are a number of strategies for one-to-many relationships, and the approach you take will depend on your needs.

In this chapter, we will cover five strategies for modeling one-to-many relationships with DynamoDB:

- Denormalization by using a complex attribute
- Denormalization by duplicating data
- Composite primary key + the Query API action
- Secondary index + the Query API action
- Composite sort keys with hierarchical data

We will cover each strategy in depth below—when you would use it, when you wouldn't use it, and an example. The end of the chapter includes a summary of the five strategies and when to choose each one.

11.1. Denormalization by using a complex attribute

Database normalization is a key component of relational database modeling and one of the hardest habits to break when moving to DynamoDB.

The basics of normalization are discussed in Chapter 7, but there are a number of areas where *denormalization* is helpful with DynamoDB.

The first way we'll use denormalization with DynamoDB is by having an attribute that uses a complex data type, like a list or a map. This violates the first tenet of database normalization: to get into first normal form, each attribute value must be atomic. They cannot be broken down any further.

Let's see this by way of an example. In the e-commerce example in Chapter 19, there are Customer entities that represent people that have created an account on our site. A single Customer can have multiple mailing addresses to which they may ship items. Perhaps I have one address for my home, another address for my workplace, and a third address for my parents (a relic from the time I sent them a belated anniversary present).

In a relational database, you would model this with two tables using a foreign key to link the tables together, as follows:

Notice that each record in the `Addresses` table includes a `CustomerId`, which identifies the Customer to which this Address belongs. You can follow the pointer to the record to find information about the Customer.

DynamoDB works differently. From our e-commerce example in Chapter 19, we have a `MailingAddresses` attribute on our Customer item. This attribute is a map and contains all addresses for the given customer:

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
CUSTOMER #alexdebrie	CUSTOMER #alexdebrie	FirstName	LastName	MailingAddresses	
		Alex	DeBrie	{ "Home": { "StreetAddress: "1122 1st Avenue", "PostalCode": "90210", "Country": "UnitedStates" }, "Business": { "StreetAddress: "555 Broadway", "PostalCode": "90211", "Country": "UnitedStates" } }	
CUSTOMER #jeffbezos	CUSTOMER #jeffbezos	FirstName	LastName	MailingAddresses	
		Jeff	Bezos	{ "Home": { "StreetAddress: "123 Spruce Drive", "PostalCode": "69361", "Country": "UnitedStates" } }	

Because `MailingAddresses` contains multiple values, it is no longer atomic and, thus, violates the principles of first normal form.

There are two factors to consider when deciding whether to handle a one-to-many relationship by denormalizing with a complex attribute:

- **Do you have any access patterns based on the values in the complex attribute?**

  All data access in DynamoDB is done via primary keys and secondary indexes. You cannot use a complex attribute like a list or a map in a primary key. Thus, you won't be able to make queries based on the values in a complex attribute.

  In our example, we don't have any access patterns like "Fetch a Customer by his or her mailing address". All use of the `MailingAddress` attribute will be in the context of a Customer, such as displaying the saved addresses on the order checkout page. Given these needs, it's fine for us to save them in a complex attribute.

- **Is the amount of data in the complex attribute unbounded?**

A single DynamoDB item cannot exceed 400KB of data. If the amount of data that is contained in your complex attribute is potentially unbounded, it won't be a good fit for denormalizing and keeping together on a single item.

In this example, it's reasonable for our application to put limits on the number of mailing addresses a customer can store. A maximum of 20 addresses should satisfy almost all use cases and avoid issues with the 400KB limit.

But you could imagine other places where the one-to-many relationship might be unbounded. For example, our e-commerce application has a concept of Orders and Order Items. Because an Order could have an unbounded number of Order Items (you don't want to tell your customers there's a maximum number of items they can order!), it makes sense to split Order Items separately from Orders.

If the answer to either of the questions above is "Yes", then denormalization with a complex attribute is not a good fit to model that one-to-many relationship.

11.2. Denormalization by duplicating data

In the strategy above, we denormalized our data by using a complex attribute. This violated the principles of first normal form for relational modeling. In this strategy, we'll continue our crusade against normalization.

Here, we'll violate the principles of second normal form by duplicating data across multiple items.

In all databases, each record is uniquely identified by some sort of

key. In a relational database, this might be an auto-incrementing primary key. In DynamoDB, this is the primary key that we discussed in previous chapters.

To get to second normal form, *each non-key attribute must depend on the whole key*. This is a confusing way to say that data should not be duplicated across multiple records. If data is duplicated, it should be pulled out into a separate table. Each record that uses that data should refer to it via a foreign key reference.

Imagine we have an application that contains Books and Authors. Each Book has an Author, and each Author has some biographical information, such as their name and birth year. In a relational database, we would model the data as follows:

Authors		
AuthorId	**AuthorName**	**AuthorBirthdate**
1	John Grisham	February 8, 1955
2	Stephen King	September 21, 1947
3	J.K. Rowling	July 31, 1965

Books			
BookId	**AuthorId**	**BookTitle**	**ReleaseYear**
1	2	The Shining	1977
2	2	It	1986
3	3	Harry Potter and the Sorcerer's Stone	1997

Note: In reality, a book can have multiple authors. For simplification of this example, we're assuming each book has exactly one author.

This works in a relational database as you can join those two tables at query-time to include the author's biographical information when retrieving details about the book.

But we don't have joins in DynamoDB. So how can we solve this? We can ignore the rules of second normal form and include the Author's biographical information on each Book item, as shown below.

Primary key		Attributes	
Partition key: AuthorName	Sort key: BookName		
Stephen King	It	AuthorBirthdate	ReleaseYear
		September 21, 1947	1986
	The Shining	AuthorBirthdate	ReleaseYear
		September 21, 1947	1977
J.K. Rowling	Harry Potter and the Sorcerer's Stone	AuthorBirthdate	ReleaseYear
		July 31, 1965	1997

Notice that there are multiple Books that contain the biographical information for the Author Stephen King. Because this information won't change, we can store it directly on the Book item itself. Whenever we retreive the Book, we will also get information about the parent Author item.

There are two main questions you should ask when considering this strategy:

- *Is the duplicated information immutable?*
- *If the data does change, how often does it change and how many items include the duplicated information?*

In our example above, we've duplicated biographical information that isn't likely to change. Because it's essentially immutable, it's OK to duplicate it without worrying about consistency issues when that data changes.

Even if the data you're duplicating does change, you still may decide to duplicate it. The big factors to consider are how often the data changes and how many items include the duplicated information.

If the data changes fairly infrequently and the denormalized items are read a lot, it may be OK to duplicate to save money on all of

those subsequent reads. When the duplicated data does change, you'll need to work to ensure it's changed in all those items.

Which leads us to the second factor—how many items contain the duplicated data. If you've only duplicated the data across three items, it can be easy to find and update those items when the data changes. If that data is copied across thousands of items, it can be a real chore to discover and update each of those items, and you run a greater risk of data inconsistency.

Essentially, you're balancing the benefit of duplication (in the form of faster reads) against the costs of updating the data. The costs of updating the data includes both factors above. If the costs of either of the factors above are low, then almost any benefit is worth it. If the costs are high, the opposite is true.

11.3. Composite primary key + the Query API action

The next strategy to model one-to-many relationships—and probably the most common way—is to use a composite primary key plus the Query API to fetch an object and its related sub-objects.

In Chapter 2, we discussed the notion of *item collections*. Item collections are all the items in a table or secondary index that share the same partition key. When using the Query API action, you can fetch multiple items within a single item collection. This can include items of different types, which can give you join-like behavior with much better performance characteristics.

Let's use one of the examples from the beginning of this section. In a SaaS application, Organizations will sign up for accounts. Then,

multiple Users will belong to an Organization and take advantage of the subscription.

Because we'll be including different types of items in the same table, we won't have meaningful attribute names for the attributes in our primary key. Rather, we'll use generic attribute names, like `PK` and `SK`, for our primary key.

We have two types of items in our table—Organizations and Users. The patterns for the `PK` and `SK` values are as follows:

Entity	PK	SK
Organizations	ORG#<OrgName>	METADATA#<OrgName>
Users	ORG#<OrgName>	USER#<UserName>

Table 7. SaaS App entity chart

The table below shows some example items:

In this table, we've added five items—two Organization items for Microsoft and Amazon, and three User items for Bill Gates, Satya

Nadella, and Jeff Bezos.

Outlined in red is the item collection for items with the partition key of `ORG#MICROSOFT`. Notice how there are two different item types in that collection. In green is the Organization item type in that item collection, and in blue is the User item type in that item collection.

This primary key design makes it easy to solve four access patterns:

1. **Retrieve an Organization.** Use the GetItem API call and the Organization's name to make a request for the item with a PK of `ORG#<OrgName>` and an SK of `METADATA#<OrgName>`.

2. **Retrieve an Organization and all Users within the Organization.** Use the Query API action with a key condition expression of `PK = ORG#<OrgName>`. This would retrieve the Organization and all Users within it, as they all have the same partition key.

3. **Retrieve only the Users within an Organization.** Use the Query API action with a key condition expression of `PK = ORG#<OrgName> AND begins_with(SK, "USER#")`. The use of the `begins_with()` function allows us to retrieve only the Users without fetching the Organization object as well.

4. **Retrieve a specific User.** If you know both the Organization name and the User's username, you can use the GetItem API call with a PK of `ORG#<OrgName>` and an SK of `USER#<Username>` to fetch the User item.

While all four of these access patterns can be useful, the second access pattern—Retrieve an Organization and all Users within the Organization—is most interesting for this discussion of one-to-many relationships. Notice how we're emulating a join operation in SQL by locating the parent object (the Organization) in the same item collection as the related objects (the Users). We are pre-joining our data by arranging them together at write time.

This is a pretty common way to model one-to-many relationships and will work for a number of situations. For examples of this strategy in practice, check out the e-commerce example in Chapter 19 or the GitHub example in Chapter 21.

11.4. Secondary index + the Query API action

A similar pattern for one-to-many relationships is to use a global secondary index and the Query API to fetch multiple items in a single request. This pattern is almost the same as the previous pattern, but it uses a secondary index rather than the primary keys on the main table.

You may need to use this pattern instead of the previous pattern because the primary keys in your table are reserved for another purpose. It could be some write-specific purpose, such as to ensure uniqueness on a particular property, or it could be because you have hierarchical data with a number of levels.

For the latter situation, let's go back to our most recent example. Imagine that in your SaaS application, each User can create and save various objects. If this were Google Drive, it might be a Document. If this were Zendesk, it might be a Ticket. If it were Typeform, it might be a Form.

Let's use the Zendesk example and go with a Ticket. For our cases, let's say that each Ticket is identified by an ID that is a combination of a timestamp plus a random hash suffix. Further, each ticket belongs to a particular User in an Organization.

If we wanted to find all Tickets that belong to a particular User, we could try to intersperse them with the existing table format from

the previous strategy, as follows:

Partition key: PK	Sort key: SK	Attributes	
ORG#MICROSOFT	METADATA#MICROSOFT	OrgName	PlanType
		Microsoft	Enterprise
	USER#BILLGATES	UserName	UserType
		Bill Gates	Member
	USER#BILLGATES#TICKET#1569468714-JM14	CreatedAt	
		2019-09-25 22:31:54	
	USER#BILLGATES#TICKET#1570952398-MQR0	CreatedAt	
		2019-10-13 02:39:58	
	USER#SATYANADELLA	UserName	UserType
		Satya Nadella	Admin
ORG#AMAZON	METADATA#AMAZON	OrgName	PlanType
		Amazon	Pro
	USER#JEFFBEZOS	OrgName	UserType
		Amazon	Admin

Ticket items (label on the two highlighted ticket rows)

Notice the two new Ticket items outlined in red.

The problem with this is that it really jams up my prior use cases. If I want to retrieve an Organization and all its Users, I'm also retrieving a bunch of Tickets. And since Tickets are likely to vastly exceed the number of Users, I'll be fetching a lot of useless data and making multiple pagination requests to handle our original use case.

Instead, let's try something different. We'll do three things:

1. We'll model our Ticket items to be in a separate item collection altogether in the main table. For the `PK` and `SK` values, we'll use a pattern of `TICKET#<TicketId>` which will allow for direct lookups of the Ticket item.

2. Create a global secondary index named `GSI1` whose keys are `GSI1PK` and `GSI1SK`.

3. For both our Ticket and User items, add values for `GSI1PK` and `GSI1SK`. For both items, the `GSI1PK` attribute value will be

`ORG#<OrgName>#USER#<UserName>`.

For the User item, the `GSI1SK` value will be `USER#<UserName>`.

For the Ticket item, the `GSI1SK` value will be `TICKET#<TicketId>`.

Now our base table looks as follows:

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
ORG#MICROSOFT	METADATA#MICROSOFT	OrgName	PlanType	*GSI1PK & GSI1SK added to User items*	
		Microsoft	Enterprise		
	USER#BILLGATES	UserName	UserType	GSI1PK	GSI1SK
		Bill Gates	Member	ORG#MICROSOFT#USER#BILLGATES	USER#BILLGATES
	USER#SATYANADELLA	UserName	UserType	GSI1PK	GSI1SK
		Satya Nadella	Admin	ORG#MICROSOFT#USER#SATYANADELLA	USER#SATYANADELLA
ORG#AMAZON	METADATA#AMAZON	OrgName	PlanType		
		Amazon	Pro		
	USER#JEFFBEZOS	UserName	UserType	GSI1PK	GSI1SK
		Jeff Bezos	Admin	ORG#AMAZON#USER#JEFFBEZOS	USER#JEFFBEZOS
TICKET#1569468714-JM14	TICKET#1569468714-JM14	CreatedAt	GSI1PK	GSI1SK	
Ticket items		2019-09-25 22:31:54	ORG#MICROSOFT#USER#BILLGATES	TICKET#1569468714-JM14	
TICKET#1570952398-MQR0	TICKET#1570952398-MQR0	CreatedAt	GSI1PK	GSI1SK	
		2019-10-13 02:39:58	ORG#MICROSOFT#USER#BILLGATES	TICKET#1570952398-MQR0	

Notice that our Ticket items are no longer interspersed with their parent Users in the base table. Further, the User items now have additional `GSI1PK` and `GSI1SK` attributes that will be used for indexing.

If we look at our GSI1 secondary index, we see the following:

Primary key		Attributes			
Partition key: GSI1PK	**Sort key: GSI1SK**				
ORG#MICROSOFT#USER#BILLGATES	TICKET#1569468714-JM14	PK	SK	CreatedAt	
		TICKET#1569468714-JM14	TICKET#1569468714-JM14	2019-09-25 22:31:54	
Bill Gates item collection	TICKET#1570952398-MQR0	PK	SK	CreatedAt	
		TICKET#1570952398-MQR0	TICKET#1570952398-MQR0	2019-10-13 02:39:58	
	USER#BILLGATES	PK	SK	UserName	UserType
		ORG#MICROSOFT	USER#BILLGATES	Bill Gates	Member
ORG#MICROSOFT#USER#SATYANADELLA	USER#SATYANADELLA	PK	SK	UserName	UserType
		ORG#MICROSOFT	USER#SATYANADELLA	Satya Nadella	Admin
ORG#AMAZON#USER#JEFFBEZOS	USER#JEFFBEZOS	PK	SK	UserName	UserType
		ORG#AMAZON	USER#JEFFBEZOS	Jeff Bezos	Admin

This secondary index has an item collection with both the User item and all of the user's Ticket items. This enables the same access patterns we discussed in the previous section.

One last note before moving on—notice that I've structured it so that the User item is the last item in the partition. This is because the Tickets are sorted by timestamp. It's likely that I'll want to fetch a User and the User's *most recent* Tickets, rather than the oldest tickets. As such, I order it so that the User is at the end of the item collection, and I can use the `ScanIndexForward=False` property to indicate that DynamoDB should start at the end of the item collection and read backwards.

For more examples of the secondary index pattern in action, see Chapters 19, 20, and 21.

11.5. Composite sort keys with hierarchical data

In the last two strategies, we saw some data with a couple levels of hierarchy—an Organization has Users, which create Tickets. But what if you have more than two levels of hierarchy? You don't want to keep adding secondary indexes to enable arbitrary levels of fetching throughout your hierarchy.

A common example in this area is around location-based data. Let's keep with our workplace theme and imagine you're tracking all the locations of Starbucks around the world. You want to be able to filter Starbucks locations on arbitrary geographic levels—by country, by state, by city, or by zip code.

We could solve this problem by using a composite sort key. This term is a little confusing, because we're using a composite primary

key on our table. The term composite sort key means that we'll be smashing a bunch of properties together in our sort key to allow for different search granularity.

Let's see how this looks in a table. Below are a few items:

Primary key		Attributes			
Partition key: Country	Sort key: STATE#CITY#ZIP				
USA	NE#OMAHA#68118	StreetAddress		SquareFeet	
		15821 W Dodge Rd #100		921	
	NY#NEWYORKCITY#10001	StreetAddress		SquareFeet	
		875 6th Ave		1211	
	NY#NEWYORKCITY#10019	StreetAddress		SquareFeet	
		1500 Broadway		1924	
FRANCE	ILE-DE-FRANCE#PARIS#75001	StreetAddress		SquareFeet	
		26 Avenue de l'Opéra		2102	

In our table, the partition key is the country where the Starbucks is located. For the sort key, we include the State, City, and ZipCode, with each level separated by a `#`. With this pattern, we can search at four levels of granularity using just our primary key!

The patterns are:

1. **Find all locations in a given country.** Use a Query with a key condition expression of `PK = <Country>`, where Country is the country you want.

2. **Find all locations in a given country and state.** Use a Query with a condition expression of `PK = <Country> AND begins_with(SK, '<State>#'`.

3. **Find all locations in a given country, state, and city.** Use a Query with a condition expression of `PK = <Country> AND`

```
begins_with(SK, '<State>#<City>'.
```

4. **Find all locations in a given country, state, city, and zip code.**
   Use a Query with a condition expression of `PK = <Country> AND`
   `begins_with(SK, '<State>#<City>#<ZipCode>'`.

This composite sort key pattern won't work for all scenarios, but it can be great in the right situation. It works best when:

- You have many levels of hierarchy (>2), and you have access patterns for different levels within the hierarchy.

- When searching at a particular level in the hierarchy, you want all subitems in that level rather than just the items in that level.

  For example, recall our SaaS example when discussing the primary key and secondary index strategies. When searching at one level of the hierarchy—find all Users—we didn't want to dip deeper into the hierarchy to find all Tickets for each User. In that case, a composite sort key will return a lot of extraneous items.

If you want a detailed walkthrough on this example, I wrote up the Starbucks example on DynamoDBGuide.com.

11.6. Summary of one-to-many relationship strategies

In this chapter, we discussed five different strategies you can implement when modeling data in a one-to-many relationship with DynamoDB. The strategies are summarized in the table below.

Strategy	Notes	Relevant examples
Denormalize + **complex attribute**	Good when nested objects are bounded and are not accessed directly	Chapter 18
Denormalize + **duplicate**	Good when duplicated data is immutable or infrequently changing	Movie roles table
Primary key + **Query API**	Most common. Good for multiple access patterns both the parent and related entities.	Chapters 19 & 20
Secondary index + **Query API**	Similar to primary key strategy. Good when primary key is needed for something else.	Chapters 19 & 20
Composite sort key	Good for deeply nested hierarchies where you need to search through multiple levels of the hierarchy	Chapter 20

Table 8. One-to-many relationship strategies

Chapter 12. Strategies for many-to-many relationships

> ## Chapter Summary
>
> This chapter contains four strategies for modeling many-to-many relationships in your DynamoDB table.
>
> ## Sections
>
> 1. Shallow duplication
> 2. Adjacency list
> 3. Materialized graph
> 4. Normalization & multiple requests

A many-to-many relationship is one in which one type of object may belong to multiple instances of a different type of object and vice versa. For example:

- **Students and classes:** A student may take multiple classes, and each class may have multiple students.

- **Movies and actors:** A movie has multiple actors that perform roles, and an actor will perform roles in multiple movies.

- **Social media friendships:** In a social media application like Twitter, each user can follow and be followed by multiple other users.

Many-to-many relationships are tricky because you often want to

query both sides of the relationship. If your table has students & classes, you may have one access pattern where you want to fetch a student and the student's schedule, and you may have a different access pattern where you want to fetch a class and all the students in the class. This is the main challenge of many-to-many access patterns.

In a relational database, you often handle this by using a linking table which serves as an intermediary between your two objects. Each object table will have a one-to-many relationship with the linking table, and you can traverse these relationships to find all related records for a particular entity.

Remember that one of the main difficulties in DynamoDB is how to properly 'pre-join' your data such that you can fetch multiple, different types of entities in a single request. There are no joins in DynamoDB so spreading them across multiple tables and combining them at query time won't work. Many-to-many relationships are one of the more difficult areas for DynamoDB to handle.

In this chapter, we will cover four strategies for modeling many-to-many relationships with DynamoDB:

- Shallow duplication

- Adjacency list

- Materialized graph

- Normalization & multiple requests

We will cover each strategy in depth below—when you would use it, when you wouldn't use it, and an example. The end of the chapter includes a summary of the four strategies and when to choose each one.

12.1. Shallow duplication

The first strategy we will use is one that I call "shallow duplication". Let's see an example where this would be helpful and then discuss when to use it.

Imagine you have an application with classes and students as mentioned above. A student is enrolled in many classes, and a class has many students.

One of your access patterns is to fetch a class and all of the students in the class. However, when fetching information about a class, you don't need detailed information about each student in the class. You only need a subset of information, such as a name or an ID. Our user interface will then provide a link to click on the student for someone that wants more detailed information about the student.

To handle this, we could model our Class items so that information about the students enrolled in the class would be duplicated into the item. It may look something like this:

Primary key		Attributes		
Partition key: PK	**Sort key: SK**			
STUDENT#alexdebrie	STUDENT#alexdebrie	StudentName	GPA	GraduationDate
		Alex DeBrie	3.12	2021-05-26
STUDENT#alberteinstein	STUDENT#alberteinstein	StudentName	GPA	GraduationDate
		Albert Einstein	4.00	2020-05-14
STUDENT#dracomalfoy	STUDENT#dracomalfoy	StudentName	GPA	
		Draco Malfoy	1.87	
CLASS#Physics101	SEMESTER#Spring2020	ClassName	Semester	Students
		Physics 101	Spring 2020	["Albert Einstein","Alex DeBrie"]
CLASS#Gym202	SEMESTER#Spring2020	ClassName	Semester	Students
		Gym 202	Spring 2020	["Draco Malfoy","Alex DeBrie"]

In this example, there are five items. The top three items are Student items that have records about the students enrolled at the school. The bottom two items are two Class items that track information about a particular class.

Notice that the Class items have a `Students` attribute (outlined in red). This is an attribute of type list and contains the names of the students in the class. Now we can easily handle this particular access pattern—fetch a class and all students in the class—with a single GetItem request to DynamoDB.

The shallow duplication strategy works when both of the following properties are true:

1. **There is a limited number of related entities in the duplicated relationship.** DynamoDB items have a 400KB limit. Similar to the "denormalization with a complex attribute" strategy in the one-to-many relationship section, this will not work if there is a high or unlimited number of related entities that you need to maintain.

2. **The duplicated information is immutable.** This strategy works if the information that is duplicated does not change. If the information is frequently changing, you will spend a lot of time looking for all the references to the data and updating accordingly. This will result in additional write capacity needs and potential data integrity issues.

   In our example, notice that we only copy the student's name, which is immutable (or close to immutable). If our access pattern needed more mutable data about the student, like the student's GPA or graduation date, then this strategy wouldn't work as well.

Note that this strategy only handles one side of the many-to-many relationship, as you still need to model out how to fetch a student and all classes in which the student is enrolled. However, by using

this pattern, you have taken out one side of the many-to-many relationship. You can now handle the other side using one of the one-to-many relationship strategies from the previous chapter.

12.2. Adjacency list

A second strategy for handling many-to-many relationships is an adjacency list. With this strategy, you model each top-level entity as an item in your table. You also model the relationship between entities as an item in your table. Then, you organize the items so that you can fetch both the top-level entity and information about the relationship in a single request.

Let's see this with an example. Imagine we are storing information about movies and actors in an application. An actor can perform in multiple movies, and a movie will have performances by multiple actors.

In our example table, we'll have three types of items with the following primary key patterns:

Movies:

- **PK:** `MOVIE#<MovieName>`
- **SK:** `MOVIE#<MovieName>`

Actors:

- **PK:** `ACTOR#<ActorName>`
- **SK:** `ACTOR#<ActorName>`

Roles:

- **PK:** `MOVIE#<MovieName>`

- **SK:** `ACTOR#<ActorName>`

The Movie and Actor items are top-level items, and the Role item represents the many-to-many relationship between Movies and Actors.

Our example table will look as follows:

Primary key		Attributes				
Partition key: PK	**Sort key: SK**					
MOVIE#ToyStory	ACTOR#TimAllen	MovieName	ActorName	RoleName		
		Toy Story	Tim Allen	Buzz Lightyear		
	ACTOR#TomHanks	MovieName	ActorName	RoleName		
		Toy Story	Tom Hanks	Woody		
	MOVIE#ToyStory	MovieName	Year	Genre	BoxOfficeReceipts	IMDBScore
		Toy Story	1995	Children's	$222,498,679	8.3
MOVIE#CastAway	ACTOR#TomHanks	MovieName	ActorName	RoleName		
		Cast Away	Tom Hanks	Chuck Noland		
	MOVIE#CastAway	MovieName	Year	Genre	BoxOfficeReceipts	IMDBScore
		Cast Away	2000	Drama	$233,632,142	7.8
MOVIE#BlackSwan	ACTOR#NataliePortman	MovieName	ActorName	RoleName		
		Black Swan	Natalie Portman	Nina Sayers		
	MOVIE#BlackSwan	MovieName	Year	Genre	BoxOfficeReceipts	IMDBScore
		Black Swan	2010	Drama	$106,954,678	8.0
ACTOR#TomHanks	ACTOR#TomHanks	ActorName	Upvotes	BirthDate		
		Tom Hanks	469396	July 9, 1956		
ACTOR#TimAllen	ACTOR#TimAllen	ActorName	Upvotes	BirthDate		
		Tim Allen	125376	June 13, 1953		

With this base table configuration, notice that our Movie and Role items are in the same item collection. This allows us to fetch a movie and actor roles that played in the movie with a single request by making a Query API call that uses `PK = MOVIE#<MovieName>` in the key condition expression.

We can then add a global secondary index that flips the composite key elements. In the secondary index, the partition key is `SK` and

the sort key is `PK`. Our index looks as follows:

Primary key		Attributes				
Partition key: SK	Sort key: PK					
MOVIE#CastAway	MOVIE#CastAway	MovieName	Year	Genre	BoxOfficeReceipts	IMDBScore
		Cast Away	2000	Drama	$233,632,142	7.8
MOVIE#BlackSwan	MOVIE#BlackSwan	MovieName	Year	Genre	BoxOfficeReceipts	IMDBScore
		Black Swan	2010	Drama	$106,954,678	8.0
ACTOR#TomHanks	ACTOR#TomHanks	ActorName	Upvotes	BirthDate		
		Tom Hanks	469396	July 9, 1956		
	MOVIE#CastAway	MovieName	ActorName	RoleName		
		Cast Away	Tom Hanks	Chuck Noland		
	MOVIE#ToyStory	MovieName	ActorName	RoleName		
		Toy Story	Tom Hanks	Woody		
ACTOR#TimAllen	ACTOR#TimAllen	ActorName	Upvotes	BirthDate		
		Tim Allen	125376	June 13, 1953		
	MOVIE#ToyStory	MovieName	ActorName	RoleName		
		Toy Story	Tim Allen	Buzz Lightyear		
ACTOR#NataliePortman	ACTOR#NataliePortman	ActorName	Upvotes	BirthDate		
		Natalie Portman	551462	June 9, 1981		
	MOVIE#BlackSwan	MovieName	ActorName	RoleName		
		Black Swan	Natalie Portman	Nina Sayers		

Now our Actor item is in the same item collection as the actor's Role items, allowing us to fetch an Actor and all roles in a single request.

The nice part about this strategy is that you can combine mutable information with immutable information in both access patterns. For example, the Movie item has some attributes that will change over time, such as the total box office receipts or the IMDB score. Likewise, the Actor item has a mutable attribute like the total number of upvotes they have received. With this setup, we can edit the mutable parts—the Movie and Actor items—without editing the immutable Role items. The immutable items are copied into both item collections, giving you a full look at the data while keeping

updates to a minimum.

This pattern works best when the information about the *relationship* between the two is immutable. In this case, nothing changes about the role that an actor played after the fact. This makes it an ideal fit for this pattern.

In this example, we flipped the `PK` and `SK` for our secondary index, and you might hear that called an *inverted index* as well. However, if you have other items in your table, you may not want to flip the `PK` and `SK` for those items, as this may not enable the access patterns you want. You could create two new attributes, `GSI1PK` and `GSI1SK` that have the flipped values from `PK` and `SK` for the items in your many-to-many relationship. Then when you create your GSI1 index, it will be the same as if you flipped the `PK` and `SK` for those items.

12.3. Materialized graph

A powerful but less-commonly used strategy for many-to-many relationships is the materialized graph. Let's first review some basics of graphs, and then see how to use the materialized graph.

A graph is made up of *nodes* and *edges*. Usually, a node is an object or concept, such as a person, place, or thing. Various nodes are then connected via edges, which indicate relationships between nodes. Thus, a person would be a node and the city of Omaha, Nebraska would be a node. One person might live in Omaha, Nebraska and that relationship would be represented by an edge.

To use a materialized graph in DynamoDB, first you create your nodes as an item collection in your base table. See the example below.

Primary key		Attributes		
Partition key: PK	Sort key: SK			
156	DATE|2011-05-28|MARRIED	NodeId	Date	GSI1PK
		156	2011-05-28	DATE|2011-05-28
	JOB|Developer|DeBrie, LLC	NodeId	Job	GSI1PK
		156	Developer	JOB|Developer
	PERSON|156	NodeId	Name	GSI1PK
		156	Alex DeBrie	PERSON|156
247	DATE|2011-05-28|MARRIED	NodeId	Date	GSI1PK
		247	2011-05-28	DATE|2011-05-28
	JOB|Attorney|Koley Jessen	NodeId	Job	GSI1PK
		247	Attorney	JOB|Attorney
	PERSON|247	NodeId	Name	GSI1PK
		247	Elsie DeBrie	PERSON|247
958	JOB|Attorney|A.C. Lee	Job	GSI1PK	
		Attorney	JOB|Attorney	
	PERSON|958	NodeId	Name	GSI1PK
		958	Atticus Finch	PERSON|958

Node 156 = Alex DeBrie

Notice that Node ID of `156` is the Person node for Alex DeBrie. Instead of including all attributes about me in a single item, I've broken it up across multiple items. I have one item that includes information about the day I was married and another item that includes information about my job. I've done similar things with my wife and with Atticus Finch.

Then, you can use a secondary index to reshuffle those items and group them according to particular relationships, as shown below:

Primary key		Attributes		
Partition key: GSI1PK	**Sort key: SK**			
PERSON|156	PERSON|156	PK	NodeId	Name
		156	156	Alex DeBrie
DATE|2011-05-28	DATE|2011-05-28|MARRIED	PK	NodeId	Date
		156	156	2011-05-28
	DATE|2011-05-28|MARRIED	PK	NodeId	Date
		247	247	2011-05-28
JOB|Developer	JOB|Developer|DeBrie, LLC	PK	NodeId	Job
		156	156	Developer
PERSON|247	PERSON|247	PK	NodeId	Name
		247	247	Elsie DeBrie
JOB|Attorney	JOB|Attorney|A.C. Lee	PK	Job	
		958	Attorney	
	JOB|Attorney|Koley Jessen	PK	NodeId	Job
		247	247	Attorney
PERSON|958	PERSON|958	PK	NodeId	Name
		958	958	Atticus Finch

Notice that there are new groupings of nodes and edges. In the partition for the date of May 28, 2011, both my wife and I have an edge in there to represent our wedding. You could imagine other items in there to represent births, deaths, or other important events.

Likewise, there are two items in the `JOB|Attorney` item to indicate the two persons that have jobs as attorneys. Notice that the `NodeId` is present on both items, so you could make follow-up requests to reconstitute the parent node by querying the base table for the given Node Id.

The materialized graph pattern can be useful for highly-connected data that has a variety of relationships. You can quickly find a particular type of entity and all the entities that relate to it. That said, I don't have a deeper example that shows the materialized graph in practice, as it's a pretty niche pattern.

12.4. Normalization and multiple requests

I mentioned at the beginning of this chapter that many-to-many relationships are one of the more difficult areas for DynamoDB to handle. This is particularly true when there is information that is highly mutable and heavily duplicated across your related items. In this situation, you might need to bite the bullet and make multiple requests to your database.

The most common example I use for this one is in a social media application like Twitter. A user can follow multiple users, and one user can be followed by multiple users. Twitter has an access pattern where they need to fetch all people that a given user is following. For example, here is what I see when I look at people I'm following:

Alex DeBrie
@alexbdebrie

Followers | **Following**

Michael Chan
@mchancloud Follows you   Following

Father of two boys, guitar and uke player. Developer Advocate for
@AWSIdentity. I for @AWSCloud & opinions are my own.

Paul Chin Jr.
@paulchinjr Follows you   Following

Curious Human. #Serverless #JavaScript Dev @ arc.codes Dev Rel at
begin.com . Prophet for the One True God. #PraiseCage

marco marandiz
@allthingsmarco   Following

Co-founder & Head of Marketing @hello_iamelliot 🐦 DTC Strategist.

Gillian Armstrong
@virtualgill Follows you   Following

Technology wrangler, Christian, passionate about the tech, psych &
philosophy of #AI, #CUX, #VUX, #Chatbots, #Serverless. AWS Machine
Learning Hero.

Mark McCann
@MarkMcCann Follows you   Following

Married to @OtherGill, father of 2 awesome Girls, Software Architect
@Liberty_IT, AWS Certified Professional.

Notice that the inimitable Paul Chin, Jr. shows up here along with
some other great AWS and Serverless folks. Importantly, it contains
information that will change over time. For each person I'm
following, it includes their display name (which can change) as well
as their profile description.

Imagine that I made a separate item in DynamoDB for each
following relationship that contained the information that needed
to be displayed on this screen. Some of it, like the username or the
timestamp when I started to follow the person, may not change. But
the other bits of information would. If we wanted to keep that

information fresh, we would need to update a user's follower items each time the user changed their display name or profile. This could add a ton of write traffic as some users have thousands or even millions of followers!

Rather than having all that write thrashing, we can do a little bit of normalization (eek!) in our DynamoDB table. First, we'll store two types of items in our table: Users and Following items. The structure will look as follows:

Users:

- **PK:** `USER#<Username>`
- **SK:** `USER#<Username>`

Following:

- **PK:** `USER#<Username>`
- **SK:** `FOLLOWING#<Username>`

Our table would look as follows:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
USER#alexbdebrie	FOLLOWING#paulchinjr	FollowingUser	FollowedUser	FollowedAt
		alexbdebrie	paulchinjr	2020-02-15 11:13:06
	FOLLOWING#virtualgill	FollowingUser	FollowedUser	FollowedAt
		alexbdebrie	virtualgill	2020-02-10 11:26:01
	USER#alexbdebrie	Username	DisplayName	Profile
		alexbdebrie	Alex DeBrie	⭐AWS Data Hero 📖 Writing http://DynamoDBBook.com. ...
USER#paulchinjr	USER#paulchinjr	Username	DisplayName	Profile
		paulchinjr	Paul Chin Jr.	Curious Human. #Serverless #JavaScript Dev @ http://arc.codes ...
USER#virtualgill	USER#virtualgill	Username	DisplayName	Profile
		virtualgill	Gillian Armstrong	Technology wrangler, Christian, passionate about the tech, psych & philosophy of ...

Notice that we have a User item record for Alex DeBrie, Paul Chin Jr., and Gillian Armstrong. We also have a Following item indicating that Alex DeBrie is following Paul Chin Jr. and Gillian

Armstrong. However, that Following item is pretty sparse, as it contains only the basics about the relationship between the two users.

When Alex DeBrie wants to view all the users he's following, the Twitter backend will do a two-step process:

1. Use the Query API call to fetch the User item and the initial Following items in Alex DeBrie's item collection to find information about the user and the first few users he's following.

2. Use the BatchGetItem API call to fetch the detailed User items for each Following item that was found for the user. This will provide the authoritative information about the followed user, such as the display name and profile.

Note that this isn't ideal as we're making multiple requests to DynamoDB. However, there's no better way to handle it. If you have highly-mutable many-to-many relationships in DynamoDB, you'll likely need to make multiple requests at read time.

Social media is my go-to example for this one, but a second one is in an online e-commerce store's shopping cart. You have customers that can purchase multiple products, and a product can be purchased by multiple customers. As a customer adds an item to the cart, you can duplicate some information about the item at that time, such as the size, price, item number, etc. However, as the user goes to check out, you need to go back to the authoritative source to find the current price and whether it's in stock. In this case, you've done a shallow duplication of information to show users the number of items in their cart and the estimated price, but you don't check the final price until they're ready to check out.

12.5. Conclusion

In this chapter, we discussed different many-to-many relationship strategies for DynamoDB. Those strategies are summarized below.

Strategy	Notes	Relevant examples
Shallow duplication	Good when a parent entity only needs minimal information about related entites	Chapter 20
Adjacency list	Good when information about the relationship is immutable or infrequently changing	Chapter 21
Materialized graph	Good for highly-interconnected data with a variety of relationships	Knowledge graph
Normalization & multiple requests	Fallback option for when you have highly-mutable data contained in the relationship	Social network recommendations

Table 9. Many-to-many relationship strategies

Chapter 13. Strategies for filtering

> ## Chapter Summary
>
> Filtering is one of the key properties of any database. This chapter shows common filtering patterns in DynamoDB.
>
> ## Sections
>
> 1. Filtering with the partition key
> 2. Filtering with the sort key
> 3. Composite sort key
> 4. Sparse indexes
> 5. Filter expressions
> 6. Client-side filtering

Database design is one giant exercise in filtering. You have a huge mess of data in your application. At certain times you want this piece of data but not that piece, while at others you want that piece but not this piece. How do you properly design your table so that you get fast, consistent performance?

That is the focus of this chapter. We'll go through a few different strategies on how to filter your data to find the needle in your haystack. But before we dive into specifics, the key point you must understand is that *filtering in DynamoDB is almost exclusively focused on your primary key*. You have to understand how to model, query,

and index your primary keys in order to get the most out of DynamoDB. Note that in this sense, I use 'primary key' to include both the primary key of your base table and of any secondary indexes.

With that note in mind, let's get started.

13.1. Filtering with the partition key

The first and easiest way to filter data in DynamoDB is with the partition key of your primary key. Recall from Chapter 3 that the partition key is responsible for determining which storage node will store your item. When a request is received by DynamoDB, it will hash the partition key to find which storage node should be used to save or read the requested item.
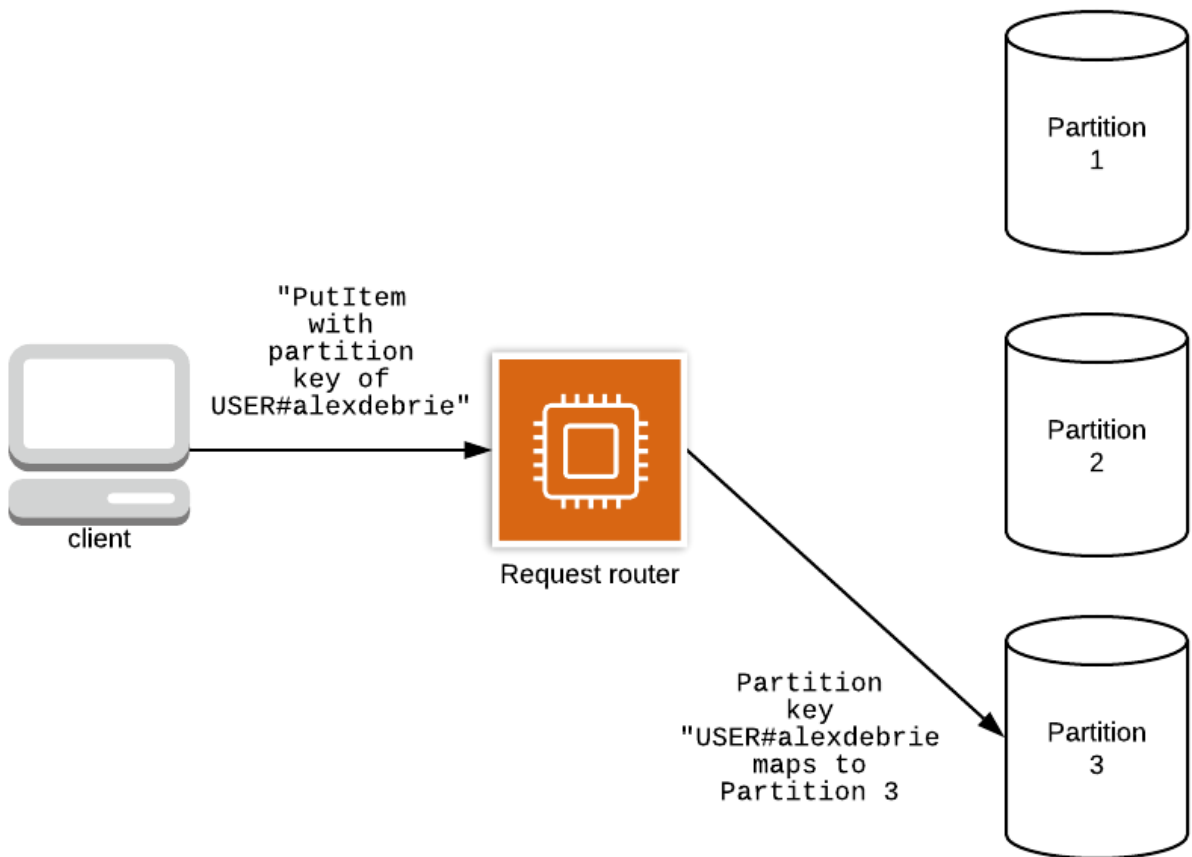
This partitioning is done to keep data access fast, no matter the size of your data. By relying heavily on this partition key, DynamoDB starts every operation with an `O(1)` lookup that reduces your dataset from terabytes or more down to a single storage node that contains a maximum of 10GB. As your data continues to grow, you'll benefit from this constant-time operation.

Be sure to use this partition key as you think about data design and filtering. In previous chapters, we've used the example of a table that includes Movies and Actors. In this example table, the partition key was the Actor name:

Primary key		Attributes		
Partition key: Actor	Sort key: Movie			
Tom Hanks	Cast Away	Role	Year	Genre
		Chuck Noland	2000	Drama
	Toy Story	Role	Year	Genre
		Woody	1995	Children's
Tim Allen	Toy Story	Role	Year	Genre
		Buzz Lightyear	1995	Children's
Natalie Portman	Black Swan	Role	Year	Genre
		Nina Sayers	2010	Drama

In our Query API requests, we can use the partition key to filter down to the specific actor we want. This makes it easy to quickly retrieve all of Natalie Portman's movies, if needed.

Likewise, we've shown how to make a secondary index to enable additional access patterns. With the Movies & Actors table, we've created a secondary index that used Movie as the partition key:

Primary key		Attributes		
Partition key: Movie	Sort key: Actor			
Cast Away	Tom Hanks	Role	Year	Genre
		Chuck Noland	2000	Drama
Toy Story	Tim Allen	Role	Year	Genre
		Buzz Lightyear	1995	Children's
	Tom Hanks	Role	Year	Genre
		Woody	1995	Children's
Black Swan	Natalie Portman	Role	Year	Genre
		Nina Sayers	2010	Drama

With this secondary index, we can use the Query API to find all actors that have been in a particular movie by specifying the movie as the partition key in our request.

The partition key has to be the starting point for your filtering. Other than the costly Scan operation, all DynamoDB APIs require the partition key in the request. Make sure you assemble items that are retrieved together in the same partition key.

13.2. Filtering with the sort key

The second common filtering strategy is to add conditions on the sort key. Just like filtering on the partition key, this strategy works for both your base table and any secondary indexes, as long as they use a composite primary key. This won't work when using a simple primary key as there is no sort key to speak of.

There are a couple of different ways that you can filter with the sort key. Let's look at two of them:

- Using simple filters
- Assembling different collections of items

13.2.1. Simple filters with the sort key

The first is pretty straightforward—if you have a clear, simple item collection that's grouped by a particular parent entity, you can just use the sort key to filter on something inherently meaningful like dates or scores.

For example, in Chapter 6, we had an example of an e-commerce application where a customer may make multiple orders over time. The table looked as follows:

Primary key		Attributes	
Partition key: CustomerId	Sort key: OrderTime		
aef7159cd662	2020-01-06 14:22:48	Amount	Status
		42.19	SHIPPED
36ab55a589e4	2020-01-08 02:27:04	Amount	Status
		179.98	CANCELLED
	2020-01-11 04:24:58	Amount	Status
		66.21	SHIPPED
	2020-01-16 02:01:36	Amount	Status
		87.77	PLACED
f7f2cb482b74	2020-01-15 14:28:29	Amount	Status
		12.44	SHIPPED

Notice that the orders use the `CustomerId` as the partition key, which groups them by customer. You could then use the sort key to find orders for a customer within a certain time range. For example, if you wanted all orders for customer `36ab55a589e4` that were placed between January 11 and February 1, 2020, you could write the following Query:

```
result = dynamodb.query(
    TableName='CustomerOrders',
    KeyConditionExpression="#c = :c AND #ot BETWEEN :start and :end",
    ExpressionAttributeNames={
        "#c": "CustomerId",
        "#ot": "OrderTime"
    },
    ExpressionAttributeValues={
        ":c": { "S": "36ab55a589e4" },
        ":start": { "S": "2020-01-11T00:00:00.000000" },
        ":end": { "S": "2020-02-01T00:00:00.000000" }
    }
)
```

This would return the following two items outlined in red:

Primary key		Attributes	
Partition key: CustomerId	**Sort key: OrderTime**		
aef7159cd662	2020-01-06 14:22:48	Amount	Status
		42.19	SHIPPED
36ab55a589e4	2020-01-08 02:27:04	Amount	Status
		179.98	CANCELLED
	2020-01-11 04:24:58	Amount	Status
		66.21	SHIPPED
	2020-01-16 02:01:36	Amount	Status
		87.77	PLACED
f7f2cb482b74	2020-01-15 14:28:29	Amount	Status
		12.44	SHIPPED

The key point with this strategy is that the sort key itself is inherently meaningful. It shows a date, and you're filtering within a certain date range.

13.2.2. Assembling different collections of items

The second pattern is slightly different in that the sort key also encodes some specific information about how you've happened to arrange the data in your table. This is hard to explain without an example, so let's start there.

In the GitHub example in Chapter 21, there are many different entities. Three of those entities are Repos, Issues, and Stars. Both Issues and Stars have a one-to-many relationship with Repos. Both Issues and Stars have an access pattern where they want to retrieve the parent Repo item as well as a number of related items (Issues or Stars).

To conserve on secondary indexes, we model all three of these items in the same item collection. A selection of the table looks as follows:

Primary key		Attributes						
Partition key: PK	Sort key: SK							
REPO#alexdebrie#dynamodb-book	ISSUE#2	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-10-20 15:32:32	2	Open	ISSUE#alexdebrie#dynamodb-book#2	ISSUE#alexdebrie#dynamodb-book#2
	ISSUE#3	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-10-22 19:35:37	3	Closed	ISSUE#alexdebrie#dynamodb-book#3	ISSUE#alexdebrie#dynamodb-book#3
	ISSUE#4	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-11-07 03:43:42	4	Open	ISSUE#alexdebrie#dynamodb-book#4	ISSUE#alexdebrie#dynamodb-book#4
	REPO#alexdebrie#dynamodb-book	RepoName	RepoOwner	CreatedAt	GSI1PK	GSI1SK		
		dynamodb-book	alexdebrie	2019-10-07 23:38:31	REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book		
	STAR#danny-developer	RepoName	RepoOwner	CreatedAt	StarringUser			
		dynamodb-book	alexdebrie	2019-12-25 01:57:12	danny-developer			
	STAR#sally-scala	RepoName	RepoOwner	CreatedAt	StarringUser			
		dynamodb-book	alexdebrie	2020-01-07 10:35:13	sally-scala			

Notice that there is an item collection that uses `REPO#alexdebrie/dynamodb-book` as the partition key. The first three items in that table are Issue items, where the sort key pattern is `ISSUE#<IssueNumber>`. Then there is a Repo item whose sort key is the same as the partition key (`REPO#alexdebrie/dynamodb-book`). Finally, there are two Star items whose sort key pattern is `STAR#<Username>`.

When we want to fetch the Repo item and all its Issue items, we need to add conditions on the sort key so that we are filtering out the Star items. To do this, we'll write a Query like the following:

```
result = dynamodb.query(
    TableName='GitHubTable',
    KeyConditionExpression="#pk = :pk AND #sk <= :sk",
    ExpressionAttributeNames={
        "#pk": "PK",
        "#sk": "SK"
    },
    ExpressionAttributeValues={
        ":pk": { "S": "REPO#alexdebrie#dynamodb-book" },
        ":sk": { "S": "REPO#alexdebrie#dynamodb-book" }
    },
    ScanIndexForward=True
)
```

With the condition on the sort key value, we're saying to only look for items whose sort key is less than or equal to REPO#alexdebrie#dynamodb-book. Further, we're using ScanIndexForward=True to read the items in descending order. This means we'll start at the Repo item and work backwards to remove all Star items from our query.

Primary key		Attributes						
Partition key: PK	**Sort key: SK**							
REPO#alexdebrie#dynamodb-book	ISSUE#2	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-10-20 15:32:32	2	Open	ISSUE#alexdebrie#dynamodb-book#2	ISSUE#alexdebrie#dynamodb-book#2
	ISSUE#3	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-10-22 19:35:37	3	Closed	ISSUE#alexdebrie#dynamodb-book#3	ISSUE#alexdebrie#dynamodb-book#3
	ISSUE#4	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-11-07 03:43:42	4	Open	ISSUE#alexdebrie#dynamodb-book#4	ISSUE#alexdebrie#dynamodb-book#4
	REPO#alexdebrie#dynamodb-book	RepoName	RepoOwner	CreatedAt	GSI1PK	GSI1SK		
		dynamodb-book	alexdebrie	2019-10-07 23:38:31	REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book		
	STAR#danny-developer	RepoName	RepoOwner	CreatedAt	StarringUser			
		dynamodb-book	alexdebrie	2019-12-25 01:57:12	danny-developer			
	STAR#sally-scala	RepoName	RepoOwner	CreatedAt	StarringUser			
		dynamodb-book	alexdebrie	2020-01-07 10:35:13	sally-scala			

Start here and read backward

When you want to fetch a Repo and all of its Stars, you would use

the opposite pattern: assert that the sort key is *greater than* or equal to `REPO#alexdebrie#dynamodb-book` to start at the Repo and work through the Star items.

Primary key		Attributes						
Partition key: PK	Sort key: SK							
	ISSUE#2	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-10-20 15:32:32	2	Open	ISSUE#alexdebrie#dynamodb-book#2	ISSUE#alexdebrie#dynamodb-book#2
	ISSUE#3	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-10-22 19:35:37	3	Closed	ISSUE#alexdebrie#dynamodb-book#3	ISSUE#alexdebrie#dynamodb-book#3
REPO#alexdebrie#dynamodb-book	ISSUE#4	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-11-07 03:43:42	4	Open	ISSUE#alexdebrie#dynamodb-book#4	ISSUE#alexdebrie#dynamodb-book#4
	REPO#alexdebrie#dynamodb-book	RepoName	RepoOwner	CreatedAt	GSI1PK	GSI1SK		
		dynamodb-book	alexdebrie	2019-10-07 23:38:31	REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book		
	STAR#danny-developer	RepoName	RepoOwner	CreatedAt	StarringUser			
		dynamodb-book	alexdebrie	2019-12-25 01:57:12	danny-developer			
	STAR#sally-scala	RepoName	RepoOwner	CreatedAt	StarringUser			
		dynamodb-book	alexdebrie	2020-01-07 10:35:13	sally-scala			

Start here and read forward — REPO#alexdebrie#dynamodb-book

The key difference between this pattern and the simple filtering pattern with the sort key is that there's no inherent meaning in the sort key values. Rather, the way that I'm sorting is a function of how I decided to arrange my items within a particular item collection.

13.3. Composite sort key

A third strategy for filtering in DynamoDB is to use a composite sort key. A composite sort key is when you combine multiple data values in a sort key that allow you to filter on both values.

> The terminology can get confusing here. A *composite primary key* is a technical term of when a primary key has two elements: a partition key and a sort key. A *composite sort key* is a term of art to indicate a sort key value that contains two or more data elements within it.

Let's see an example that uses a composite sort key, then we'll discuss when it's helpful.

Imagine you have an online e-commerce store where customers can make orders. An order can be in one of four statuses: PLACED, SHIPPED, DELIVERED, or CANCELLED. In your user interface, you give users a way to generate reports on past orders, including looking for orders in a specific time period that have a particular status (e.g.: "Give me all CANCELLED orders between July 1, 2018 and September 30, 2018".)

For customers that have placed a lot of orders, this could be an expensive operation to retrieve all orders and filter out the ones that don't match. Rather than wasting a bunch of read capacity, we can use a composite sort key to handle this complex pattern.

Imagine your table of orders looks as follows:

Primary key		Attributes		
Partition key: CustomerId	**Sort key: OrderId**			
2b5a41c0	0d397f6ca4ba	Amount	OrderStatus	OrderDate
		99.14	CANCELLED	2020-01-14
	33ae6f099c76	Amount	OrderStatus	OrderDate
		78.03	DELIVERED	2018-02-14
	5dd7d9d6e862	Amount	OrderStatus	OrderDate
		20.45	CANCELLED	2019-07-25
	954d5cc59f88	Amount	OrderStatus	OrderDate
		19.67	CANCELLED	2019-01-19
b6bc729a	5efef0259abc	Amount	OrderStatus	OrderDate
		14.12	PLACED	2020-02-18

To enable your pattern, first you make an attribute that is a

combination of the `OrderStatus` attribute and the `OrderDate` attribute. The value for this `OrderStatusDate` attribute is equal to those two attributes separated by a `#`, as shown below.

Primary key		Attributes				
Partition key: CustomerId	Sort key: OrderId					
2b5a41c0	0d397f6ca4ba	Amount	OrderStatus	OrderDate		OrderStatusDate
		99.14	CANCELLED +	2020-01-14	=	CANCELLED#2020-01-14
	33ae6f099c76	Amount	OrderStatus	OrderDate		OrderStatusDate
		78.03	DELIVERED +	2018-02-14	=	DELIVERED#2018-02-14
	5dd7d9d6e862	Amount	OrderStatus	OrderDate		OrderStatusDate
		20.45	CANCELLED +	2019-07-25	=	CANCELLED#2019-07-25
	954d5cc59f88	Amount	OrderStatus	OrderDate		OrderStatusDate
		19.67	CANCELLED +	2019-01-19	=	CANCELLED#2019-01-19
b6bc729a	5efef0259abc	Amount	OrderStatus	OrderDate		OrderStatusDate
		14.12	PLACED +	2020-02-18	=	PLACED#2020-02-18

Then, you create a secondary index using that attribute as a sort key. In this example, we'll continue to use the `CustomerId` as the partition key. Our secondary index looks as follows:

Primary key		Attributes			
Partition key: CustomerId	Sort key: OrderStatusDate				
2b5a41c0	CANCELLED#2019-01-19	OrderId	Amount	OrderStatus	OrderDate
		954d5cc59f88	19.67	CANCELLED	2019-01-19
	CANCELLED#2019-07-25	OrderId	Amount	OrderStatus	OrderDate
		5dd7d9d6e862	20.45	CANCELLED	2019-07-25
	CANCELLED#2020-01-14	OrderId	Amount	OrderStatus	OrderDate
		0d397f6ca4ba	99.14	CANCELLED	2020-01-14
	DELIVERED#2018-02-14	OrderId	Amount	OrderStatus	OrderDate
		33ae6f099c76	78.03	DELIVERED	2018-02-14
b6bc729a	PLACED#2020-02-18	OrderId	Amount	OrderStatus	OrderDate
		5efef0259abc	14.12	PLACED	2020-02-18

Now we can use the Query API to quickly find what we want. To find all CANCELLED orders for customer `2b5a41c0` between July 1, 2019 and September 30, 2019, you would write the following

Query:

```
result = dynamodb.query(
    TableName='CustomerOrders',
    IndexName="OrderStatusDateGSI",
    KeyConditionExpression="#c = :c AND #osd BETWEEN :start and :end",
    ExpressionAttributeNames={
        "#c": "CustomerId",
        "#ot": "OrderStatusDate"
    },
    ExpressionAttributeValues={
        ":c": { "S": "2b5a41c0" },
        ":start": { "S": "CANCELLED#2019-07-01T00:00:00.000000" },
        ":end": { "S": "CANCELLED#2019-10-01T00:00:00.000000" },
    }
)
```

This would return the following items:

Primary key		Attributes			
Partition key: CustomerId	Sort key: OrderStatusDate				
2b5a41c0	CANCELLED#2019-01-19	OrderId	Amount	OrderStatus	OrderDate
		954d5cc59f88	19.67	CANCELLED	2019-01-19
	CANCELLED#2019-07-25	OrderId	Amount	OrderStatus	OrderDate
		5dd7d9d6e862	20.45	CANCELLED	2019-07-25
	CANCELLED#2020-01-14	OrderId	Amount	OrderStatus	OrderDate
		0d397f6ca4ba	99.14	CANCELLED	2020-01-14
	DELIVERED#2018-02-14	OrderId	Amount	OrderStatus	OrderDate
		33ae6f099c76	78.03	DELIVERED	2018-02-14
b6bc729a	PLACED#2020-02-18	OrderId	Amount	OrderStatus	OrderDate
		5efef0259abc	14.12	PLACED	2020-02-18

This is much more efficient than looking through all of the order items to find the proper ones.

The composite sort key pattern works well when the following statements are true:

1. **You always want to filter on two or more attributes in a particular access pattern.**

2. **One of the attributes is an enum-like value.**

In this example, we want to allow users to filter on `OrderStatus` *plus* `OrderDate`, which means the first property is true. Second, the `OrderStatus` attribute has a limited set of potential values.

Notice how our items are sorted in our secondary index. They are sorted *first* by the `OrderStatus`, *then* by the `OrderDate`. This means we can do an exact match on that value and use more fine-grained filtering on the second value.

This pattern would not work in reverse. If you made your composite sort key to be `<OrderDate>#<OrderStatus>`, the high cardinality of the `OrderDate` value would intersperse items such that the `OrderStatus` property would be useless.

13.4. Sparse indexes

The next filtering strategy (and one of my favorites) is to make use of sparse indexes. First, we'll take a look at what a sparse index is and then we'll see where it's helpful.

When creating a secondary index, you will define a key schema for the index. When you write an item into your base table, DynamoDB will copy that item into your secondary index if it has the elements of the key schema for your secondary index. Crucially, if an item doesn't have those elements, it won't be copied into the secondary index.

This is the important concept behind a sparse index. A sparse index is one that intentionally excludes certain items from your table to help satisfy a query. And this can be an instrumental pattern when modeling with DynamoDB.

Note: if you're using overloaded secondary indexes, many of your secondary indexes might technically be sparse indexes. Imagine you have a table that has three different entity types — Organization, User, and Ticket. If two of those entities have two access patterns while the third entity only has one, the two entities will likely be projected into an overloaded secondary index. Technically, this is a sparse index because it doesn't include all items in your base table. However, it's a less-specific version of the sparse index pattern.

I like to call it a sparse index when there is an explicit intention to use the sparseness of the index for data modeling. This shows up most clearly in two situations:

- Filtering within an entity type based on a particular condition
- Projecting a single type of entity into a secondary index.

Let's look at each of these in turn.

13.4.1. Using sparse indexes to provide a global filter on an item type

The first example of using a sparse index is when you filter within an entity type based on a particular condition.

Imagine you had a SaaS application. In your table, you have Organization items that represent the purchasers of your product. Each Organization is made up of Members which are people that have access to a given Organization's subscription.

Your table might look as follows:

Primary key			
Partition key: PK	Sort key: SK		
ORG#BERKSHIRE	ORG#BERKSHIRE	OrgName	SubscriptionLevel
		Berkshire Hathaway	Enterprise
	USER#CHARLIEMUNGER	UserName	Role
		Charlie Munger	Member
	USER#WARRENBUFFETT	UserName	Role
		Warren Buffett	Admin
ORG#FACEBOOK	ORG#FACEBOOK	OrgName	SubscriptionLevel
		Facebook	Pro
	USER#SHERYLSANDBERG	UserName	Role
		Sheryl Sandberg	Admin

Notice that our Users have different roles in our application. Some are Admins and some are regular Members. Imagine that you had an access pattern that wanted to fetch all Users that had Administrator privileges within a particular Organization.

If an Organization had a large number of Users and the condition we want is sufficiently rare, it would be very wasteful to read all Users and filter out those that are not administrators. The access pattern would be slow, and we would expend a lot of read capacity on discarded items.

Instead, we could use a sparse index to help. To do this, we would add an attribute to only those User items which have Administrator privileges in their Organization.

To handle this, we'll add `GSI1PK` and `GSI1SK` attributes to Organizations and Users. For Organizations, we'll use `ORG#<OrgName>` for both attributes. For Users, we'll use

`ORG#<OrgName>` as the `GSI1PK`, and we'll include `Admin` as the `GSI1SK` *only if the user is an administrator in the organization.*

Our table would look as follows:

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
ORG#BERKSHIRE	ORG#BERKSHIRE	OrgName	SubscriptionLevel	GSI1PK	GSI1SK
		Berkshire Hathaway	Enterprise	ORGANIZATIONS	Berkshire Hathaway
	USER#CHARLIEMUNGER	UserName	Role		
		Charlie Munger	Member		
	USER#WARRENBUFFETT	UserName	Role	GSI1PK	GSI1SK
		Warren Buffett	Admin	ORG#BERKSHIRE	Admin
ORG#FACEBOOK	ORG#FACEBOOK	OrgName	SubscriptionLevel	GSI1PK	GSI1SK
		Facebook	Pro	ORGANIZATIONS	Facebook
	USER#SHERYLSANDBERG	UserName	Role	GSI1PK	GSI1SK
		Sheryl Sandberg	Admin	ORG#FACEBOOK	Admin

Notice that both Warren Buffett and Sheryl Sandberg have values for GSI1SK but Charlie Munger does not, as he is not an admin.

Let's take a look at our secondary index:

Primary key		Attributes			
Partition key: GSI1PK	**Sort key: GSI1SK**				
ORGANIZATIONS	Berkshire Hathaway	PK	SK	OrgName	SubscriptionLevel
		ORG#BERKSHIRE	ORG#BERKSHIRE	Berkshire Hathaway	Enterprise
	Facebook	PK	SK	OrgName	SubscriptionLevel
		ORG#FACEBOOK	ORG#FACEBOOK	Facebook	Pro
ORG#BERKSHIRE	Admin	PK	SK	UserName	Role
		ORG#BERKSHIRE	USER#WARRENBUFFETT	Warren Buffett	Admin
ORG#FACEBOOK	Admin	PK	SK	UserName	Role
		ORG#FACEBOOK	USER#SHERYLSANDBERG	Sheryl Sandberg	Admin

Notice we have an overloaded secondary index and are handling multiple access patterns. The Organization items are put into a single partition for fast lookup of all Organizations, and the User

items are put into partitions to find Admins in an Organization.

The key to note here is that we're intentionally using a sparse index strategy to filter out User items that are not Administrators. We can still use non-sparse index patterns for other entity types.

The next strategy is a bit different. Rather than using an overloaded index, it uses a dedicated sparse index to handle a single type of entity.

13.4.2. Using sparse indexes to project a single type of entity

A second example of where I like to use a sparse index is if I want to project a single type of entity into an index. Let's see an example of where this can be useful.

Imagine I have an e-commerce application. I have several different entity types in my application, including Customers that make purchases, Orders that indicate a particular purchase, and InventoryItems that represent products I have available for sale.

My table might look as follows:

Primary key		Attributes	
Partition key: PK	Sort key: SK		
CUSTOMER#ALEXDEBRIE	CUSTOMER#ALEXDEBRIE	CustomerName	JoinedAt
		Alex DeBrie	2020-01-12 08:02:57
CUSTOMER#BARRYBONDS	CUSTOMER#BARRYBONDS	CustomerName	JoinedAt
		Barry Bonds	2020-01-18 19:53:59
ORDER#1MLKM4	ORDER#1MLKM4	OrderDate	OrderCustomer
		2020-01-19 23:08:31	Barry Bonds
ORDER#OIQ410	ORDER#OIQ410	OrderDate	OrderCustomer
		2020-01-24 22:27:43	Alex DeBrie
INVENTORY#PQOIUA0	INVENTORY#PQOIUA0	ItemName	Price
		Spalding basketball	29.99
INVENTORY#DAJR10	INVENTORY#DAJR10	ItemName	Price
		A Tale of Two Cities	9.99

Notice that the table includes Customers, Orders, and InventoryItems, as discussed, and these items are interspersed across the table.

My marketing department occasionally wants to send marketing emails to all Customers to alert them of hot sales or new products. To find all my Customers in my base table is an expensive task, as I would need to scan my entire table and filter out the items that aren't Customers. This is a big waste of time and of my table's read capacity.

Instead of doing that, I'll add an attribute called `CustomerIndexId` on my Customer items. Now my table looks as follows:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
CUSTOMER#ALEXDEBRIE	CUSTOMER#ALEXDEBRIE	CustomerName	JoinedAt	CustomerIndexId
		Alex DeBrie	2020-01-12 08:02:57	Alex DeBrie
CUSTOMER#BARRYBONDS	CUSTOMER#BARRYBONDS	CustomerName	JoinedAt	CustomerIndexId
		Barry Bonds	2020-01-18 19:53:59	Barry Bonds
ORDER#1MLKM4	ORDER#1MLKM4	OrderDate	OrderCustomer	
		2020-01-19 23:08:31	Barry Bonds	
ORDER#OIQ410	ORDER#OIQ410	OrderDate	OrderCustomer	
		2020-01-24 22:27:43	Alex DeBrie	
INVENTORY#PQOIUA0	INVENTORY#PQOIUA0	ItemName	Price	
		Spalding basketball	29.99	
INVENTORY#DAJR10	INVENTORY#DAJR10	ItemName	Price	
		A Tale of Two Cities	9.99	

Notice the Customer items now have an attribute named `CustomerIndexId` as outlined in red.

Then, I create a secondary index called `CustomerIndex` that uses `CustomerIndexId` as the partition key. Only Customer items have that attribute, so they are the only ones projected into that index.

The secondary index looks as follows:

Primary key	Attributes			
Partition key: CustomerIndexId				
Alex DeBrie	PK	SK	CustomerName	JoinedAt
	CUSTOMER#ALEXDEBRIE	CUSTOMER#ALEXDEBRIE	Alex DeBrie	2020-01-12 08:02:57
Barry Bonds	PK	SK	CustomerName	JoinedAt
	CUSTOMER#BARRYBONDS	CUSTOMER#BARRYBONDS	Barry Bonds	2020-01-18 19:53:59

Only the Customer items are projected into this table. Now when the marketing department wants to find all Customers to send marketing emails, they can run a Scan operation on the `CustomerIndex`, which is much more targeted and efficient. By isolating all items of a particular type in the index, our sparse index makes finding all items of that type much faster.

Again, notice that this strategy does not work with index overloading. With index overloading, we're using a secondary index to index different entity types in different ways. However, this strategy relies on projecting only a single entity type into the secondary index.

Both sparse index patterns are great for filtering out non-matching items entirely.

13.5. Filter Expressions

In Chapter 6, we reviewed the notion of filter expressions. You can include filter expressions in Query and Scan operations to remove items from your results that don't match a given condition.

As discussed in that chapter, filter expressions seem powerful but are much more limited than they appear. This is because a filter expression is applied *after* items are read, meaning you pay for all of the items that get filtered out and you are subject to the 1MB results limit before your filter is evaluated. Because of this, you cannot count on filter expressions to save a bad model. Filter expressions are, at best, a way to slightly improve the performance of a data model that already works well.

I mentioned in Chapter 6 that I like to use filter expressions in the following situations:

1. *Reducing response payload size*. DynamoDB can return up to 1MB in a single response. That's a lot of data to send over the wire, particularly if you're just going to filter out a bunch of it once it's returned. Server-side filtering can reduce the data transferred and speed up response times.

2. *Easier application filtering*. If you'll retrieve some results from

DynamoDB and immediately run a `filter()` method to throw some away, it can be easier to handle than in your API request to DynamoDB. This is more of a personal preference.

3. *Better validation around time-to-live (TTL) expiry.* When using DynamoDB TTL, AWS states that items are generally deleted within 48 hours of their TTL expiry. This is a wide range! If you're counting on expiration as a part of your business logic, you could get incorrect results. To help guard against this, you could write a filter expression that removes all items that should have been expired by now, even if they're not quite expired yet. To see an example of this in action, check out Chapter 18.

For the first two examples, I consider a filter expression an acceptable tactic if I get at least a 30-40% hit rate on my filter OR if the total result size before the filter is pretty small (definitely under 1MB and likely under a 100KB or so). If your hit rate is lower than that and you have a large result set, you're wasting a ton of extra read capacity just to throw it all away.

I want to address one more issue with filter expressions before moving on. Reliance on filter expressions can make it more difficult to get a specific number of items. Let's see this with an example.

We've used the example of an e-commerce application a few times in this book. Imagine that the `GET /orders` API endpoint promises to return ten orders to the client. Further, imagine that you can add a filter to the API based on order status such that you can return only "SHIPPED" orders or only "CANCELLED" orders.

If you implement this filter via a filter expression, you don't know how many items you will need to fetch to ensure you get ten orders to return to the client. Accordingly, you'll likely need to vastly overfetch your items or have cases where you make follow-up requests to retrieve additional items.

If you have a table where the order status is built into the access pattern, like the composite sort key example shown above, you know that you can add a `Limit=10` parameter into your request and retrieve exactly ten items.

13.6. Client-side filtering

The final filtering strategy is a little different, as it's not a *DynamoDB* filter strategy at all. With client-side filtering, you delay all filtering to the client rather than to DynamoDB.

There are a couple places I like to use client-side filtering:

1. When filtering is difficult to model in the database
2. When the dataset isn't that big.

For the first example, imagine you have a calender application. You want to make it easy to find where certain gaps exist in the calendar in order to schedule new appointments.

This can be a tricky access pattern to handle in the database directly. Appointments can be for different time ranges, and they can start at odd times. The constraints on a `PutItem` or `UpdateItem` request can be quite difficult to handle.

Instead of doing that, just invert it by pulling all calendar information into the client. The client can use an algorithm to determine where openings exist and then save the updated results to the database accordingly.

For the second factor, I often see people ask how to implement many different filter conditions on a dataset. The canonical example here is a table view in a web application. Perhaps the table has 10 different columns, and users can specify conditions, filters,

or sort conditions for each of the columns.

If the data set that is being manipulated isn't that large (sub-1MB), just send the entire data set to the client and let all the manipulation happen client-side. You'll be able to handle any number of conditions that are thrown at you, including full-text search, and you won't be pounding your database to handle it.

Again, this only works if the dataset is small to begin with. If you are talking about Amazon.com's inventory, it's probably not a good fit. Their inventory is so broad with so many filters (categories, price, brand, size, etc.) that you cannot pull down all that data. But if we're talking about showing a table view of a particular user's orders from your e-commerce site, it's much more manageable. At that point, you've already done the work to narrow the vast dataset down to a single customer, so you can let the client handle the rest.

To quote the excellent Rick Houlihan: "the browser is sitting in a 99% idle loop. Give it something to do!" Client-side filtering can be a great way to get some free compute without adding complexity to your access patterns.

13.7. Conclusion

In this chapter, we discussed six different strategies for filtering in DynamoDB. Those strategies are summarized below.

Strategy	Notes	Relevant examples
Filtering with the partition key	Good for initial segmentation of data into smaller groups	All examples
Filtering with the sort key	Good for grabbing smaller subsets (e.g. timerange)	E-commerce orders

Strategy	Notes	Relevant examples
Composite sort key	Good for filtering on multiple fields	E-commerce filtering on order status and time
Sparse index	Good for providing a global filter on a set of data	Deals example
Filter expressions	Good for removing a few extraneous items from a narrow result set	GitHub example
Client-side filtering	Good for flexible filtering and sorting on a small (<1MB) set of data	Table views

Table 10. Filtering strategies

Chapter 14. Strategies for sorting

Chapter Summary

When considering your DynamoDB access patterns up front, this includes any sorting requirements you have. In this chapter, we'll review different strategies for properly sorting your data.

Sections

1. Basics of sorting
2. Sorting on changing attributes
3. Ascending vs. descending
4. Two relational access patterns in a single item collection
5. Zero-padding with numbers
6. Faking ascending order

As we've discussed over and over, almost everything in DynamoDB flows through your primary keys. It's true for relationships, it's true for filtering, and, as we'll see in this chapter, it's true for sorting. Like with joins and filtering, you need to arrange your items so they're sorted in advance.

If you need to have specific ordering when retrieving multiple items in DynamoDB, there are two main rules you need to follow. First, you must use a composite primary key. Second, all ordering

must be done with the sort key of a particular item collection.

Think back to our discussion in Chapter 4 about how DynamoDB enforces efficiency. First, it uses the partition key to isolate item collections into different partitions and enables an `O(1)` lookup to find the proper node. Then, items within an item collection are stored as a B-tree which allow for `O(log n)` time complexity on search. This B-tree is arranged in lexicographical order according to the sort key, and it's what you'll be using for sorting.

In this chapter, we'll review some strategies for sorting. We'll cover:

- Basics of sorting (what is lexicographical sorting, how to handle timestamps, etc.)
- Sorting on changing attributes
- Ascending vs. descending
- Two one-to-many access patterns in a single item collection
- Zero-padding with numbers
- Faking ascending order

It's a lot to cover, so let's get started!

14.1. Basics of sorting

Before we dive into actual sorting strategies, I want to cover the basics of how DynamoDB sorts items.

As mentioned, sorting happens only on the sort key. You can only use the scalar types of string, number, and binary for a sort key. Thus, we don't need to think about how DynamoDB would sort a map attribute!

For sort keys of type number, the sorting is exactly as you would expect—items are sorted according to the value of the number.
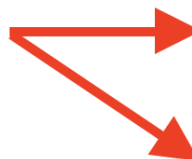
For sort keys of type string or binary, they're sorted in order of UTF-8 bytes. Let's take a deeper look at what that means.

14.1.1. Lexicographical sorting

A simplified version of sorting on UTF-8 bytes is to say the ordering is *lexicographical*. This order is basically dictionary order with two caveats:

1. All uppercase letters come before lowercase letters
2. Numbers and symbols (e.g. `#` or `$`) are relevant too.

The biggest place I see people get tripped up with lexicographical ordering is by forgetting about the uppercase rule. For example, my last name is "DeBrie" (note the capital "B" in the middle). Imagine you had Jimmy Dean, Laura Dern, and me in an item collection using our last names. If you forgot about capitalization, it might turn out as follows:

| Primary key | | Attributes |
Partition key: Initial	Sort key: LastName	
DeBrie before Dean?? → D	DeBrie	Name
		Alex DeBrie
	Dean	Name
		Jimmy Dean
	Dern	Name
		Laura Dern

You might be surprised to see that DeBrie came before Dean! This is due to the casing—uppercase before lowercase.

To avoid odd behavior around this, you should standardize your sort keys in all uppercase or all lowercase values:

Primary key		Attributes
Partition key: Initial	Sort key: LastName	
D	DEAN	Name
		Jimmy Dean
	DEBRIE	Name
		Alex DeBrie
	DERN	Name
		Laura Dern

With all last names in uppercase, they are now sorted as we would expect. You can then hold the properly-capitalized value in a different attribute in your item.

14.1.2. Sorting with Timestamps

A second basic point I want to cover with sorting is how to handle timestamps. I often get asked the best format to use for timestamps. Should we use an epoch timestamp (e.g. `1583507655`) with DynamoDB's number type? Or should we use ISO-8601 (e.g. `2020-03-06T15:14:15`)? Or something else?

First off, your choice needs to be sortable. In this case, either epoch timestamps or ISO-8601 will do. What you absolutely cannot do is use something that's not sortable, such as a display-friendly format like "May 26, 1988". This won't be sortable in DynamoDB, and you'll be in a world of hurt.

Beyond that, it doesn't make a huge difference. I prefer to use ISO-8601 timestamps because they're human-readable if you're debugging items in the DynamoDB console. That said, it can be

tough to decipher items in the DynamoDB console if you have a single-table design. As mentioned in Chapter 9, you should have some scripts to aid in pulling items that you need for debugging.

14.1.3. Unique, sortable IDs

A common need is to have unique, sortable IDs. This comes up when you need a unique identifier for an item (and ideally a mechanism that's URL-friendly) but you also want to be able to sort a group of these items chronologically. This problem comes up in both the Deals example (Chapter 17) and the GitHub Migration example (Chapter 19).

There are a few options in this space, but I prefer the KSUID implementation from the folks at Segment. A KSUID is a K-Sortable Unique Identifier. Basically, it's a unique identifier that is prefixed with a timestamp but also contains enough randomness to make collisions very unlikely. In total, you get a 27-character string that is more unique than a UUIDv4 while still retaining lexicographical sorting.

Segment released a CLI tool for creating and inspecting KSUIDs. The output is as follows:

```
ksuid -f inspect

REPRESENTATION:

  String: 1YnlHOfSSk3DhX4BR6lMAceAo1V
     Raw: 0AF14D665D6068ACBE766CF717E210D69C94D115

COMPONENTS:

       Time: 2020-03-07T13:02:30.000Z
  Timestamp: 183586150
    Payload: 5D6068ACBE766CF717E210D69C94D115
```

The "String" version of it (`1YnlHOfSSk3DhX4BR6lMAceAo1V`) shows the actual value you would use in your application. The various

components below show the time and random payload that were used.

There are implementations of KSUIDs for many of the popular programming languages, and the algorithm is pretty straightforward if you do need to implement it yourself.

> ℹ️ Thanks to Rick Branson and the folks at Segment for the implementation and description of KSUIDs. For more on this, check out Rick's blog post on the implementation of KSUIDs.

14.2. Sorting on changing attributes

The sort key in DynamoDB is used for sorting items within a given item collection. This can be great for a number of purposes, including viewing the most recently updated items or a leaderboard of top scores. However, it can be tricky if the value you are sorting on is frequently changing. Let's see this with an example.

Imagine you have a ticket tracking application. Organizations sign up for your application and create tickets. One of the access patterns is to allow users to view tickets in order by the most recently updated.

You decide to model your table as follows:

Primary key		Attributes	
Partition key: OrgName	**Sort key: UpdatedAt**		
Amazon	2020-01-07 06:18:24	TicketId	1340
	2020-01-10 02:09:14	TicketId	1381
	2020-01-18 16:47:32	TicketId	2248
Trek10	2020-01-01 15:28:20	TicketId	841
	2020-01-23 03:08:25	TicketId	144

In this table design, the organization name is the partition key, which gives us the 'group by' functionality. Then the timestamp for when the ticket was last updated is the sort key, which gives us 'order by' functionality. With this design, we could use DynamoDB's Query API to fetch the most recent tickets for an organization.

However, this design causes some problems. When updating an item in DynamoDB, you may not change any elements of the primary key. In this case, your primary key includes the UpdatedAt field, which changes whenever you update a ticket. Thus, anytime you update a ticket item, we would need first to delete the existing ticket item, then create a new ticket item with the updated primary key.

We have caused a needlessly complicated operation and one that could result in data loss if you don't handle your operations correctly.

Instead, let's try a different approach. For our primary key, let's use two attributes that won't change. We'll keep the organization name as the partition key but switch to using TicketId as the sort key.

Now our table looks as follows:

Primary key		Attributes	
Partition key: OrgName	Sort key: TicketId		
Amazon	1340	UpdatedAt	2020-01-07 06:18:24
	1381	UpdatedAt	2020-01-10 02:09:14
	2248	UpdatedAt	2020-01-18 16:47:32
Trek10	144	UpdatedAt	2020-01-23 03:08:25
	841	UpdatedAt	2020-01-01 15:28:20

Now we can add a secondary index where the partition key is OrgName and the sort key is UpdatedAt. Each item from the base table is copied into the secondary index, and it looks as follows:

Primary key		Attributes
Partition key: OrgName	Sort key: UpdatedAt	
Amazon	2020-01-07 06:18:24	TicketId
		1340
	2020-01-10 02:09:14	TicketId
		1381
	2020-01-18 16:47:32	TicketId
		2248
Trek10	2020-01-01 15:28:20	TicketId
		841
	2020-01-23 03:08:25	TicketId
		144

Notice that this is precisely how our original table design used to look. We can use the Query API against our secondary index to satisfy our 'Fetch most recently updated tickets' access pattern. More importantly, we don't need to worry about complicated delete + create logic when updating an item. We can rely on DynamoDB to handle that logic when replicating the data into a secondary index.

14.3. Ascending vs. descending

Now that we've got the basics out of the way, let's look at some more advanced strategies.

As we discussed in Chapter 5, you can use the `ScanIndexForward` property to tell DynamoDB how to order your items. By default, DynamoDB will read items in *ascending* order. If you're working with words, this means starting at aardvark and going toward zebra.

If you're working with timestamps, this means starting at the year 1900 and working toward the year 2020.

You can flip this by using `ScanIndexForward=False`, which means you'll be reading items in *descending* order. This is useful for a number of occasions, such as when you want to get the *most recent* timestamps or you want to find the *highest* scores on the leaderboard.

One complication arises when you are combining the one-to-many relationship strategies from Chapter 9 with the sorting strategies in this chapter. With those strategies, you are often combining multiple types of entities in a single item collection and using it to read both a parent and multiple related entities in a single request.

When you do this, you need to consider the common sort order you'll use in this access pattern to know where to place the parent item.

For example, imagine you have an IoT device that is sending back occasional sensor readings. One of your common access patterns is to fetch the Device item and the most recent 10 Reading items for the device.

You could model your table as follows:

Primary key		Attributes
Partition key: PK	**Sort key: SK**	
Device item → DEVICE#123	DEVICE#123	DeviceLocation
		Omaha, NE
Readings in ascending order ↓ DEVICE#123	READING#2020-03-14T10:33:00	Temperature
		32.1
	READING#2020-03-14T10:34:00	Temperature
		32.2
	READING#2020-03-14T10:35:00	Temperature
		32.2
	READING#2020-03-14T10:36:00	Temperature
		32.4
	READING#2020-03-14T10:37:00	Temperature
		32.4

Notice that the parent Device item is located before any of the Reading items because "DEVICE" comes before "READING" in the alphabet. Because of this, our Query to get the Device and the Readings would retrieve the oldest items. If our item collection was big, we might need to make multiple pagination requests to get the most recent items.

To avoid this, we can add a # prefix to our Reading items. When we do that, our table looks as follows:

Primary key		Attributes
Partition key: PK	**Sort key: SK**	
DEVICE#123	#READING#2020-03-14T10:33:00	Temperature
		32.1
	#READING#2020-03-14T10:34:00	Temperature
		32.2
	#READING#2020-03-14T10:35:00	Temperature
		32.2
	#READING#2020-03-14T10:36:00	Temperature
		32.4
	#READING#2020-03-14T10:37:00	Temperature
		32.4
	DEVICE#123	DeviceLocation
		Omaha, NE

Readings in descending order

Device item

Now we can use the Query API to fetch the Device item and the most recent Reading items by starting at the end of our item collection and using the `ScanIndexForward=False` property.

When you are co-locating items for one-to-many or many-to-many relationships, be sure to consider the order in which you want the related items returned so that your parent itself is located accordingly.

14.4. Two relational access patterns in a single item collection

Let's take that last example up another level. If we can handle a one-to-many relationship where the related items are located before or after the parent items, can we do both in one item collection?

We sure can! To do this, you'll need to have one access pattern in which you fetch the related items in ascending order and another access pattern where you fetch the related items in descending order. It also works if order doesn't really matter for one of the two access patterns.

For example, imagine you had a SaaS application. Organizations sign up and pay for your application. Within an Organization, there are two sub-concepts: Users and Teams. Both Users and Teams have a one-to-many relationship with Organizations.

For each of these relationships, imagine you had a relational access pattern of "Fetch Organization and all {Users|Teams} for the Organization". Further, you want to fetch Users in alphabetical order, but Teams can be returned in any order because there shouldn't be that many of them.

You could model your table as follows:

Primary key		Attributes
Partition key: PK	Sort key: SK	
ORG#MCDONALDS	#TEAM#FANSOFISH *Team items*	TeamName
		Fans O' Fish
	#TEAM#TEAMBURGER	TeamName
		Team Burger
	ORG#MCDONALDS *Org item*	OrgName
		McDonalds
	USER#HAMBURGLAR	UserName
		Hamburglar
	USER#RONALD *User items*	UserName
		Ronald McDonald

In this table, we have our three types of items. All share the same `PK` value of `ORG#<OrgName>`. The Team items have a `SK` of

#TEAM#<TeamName>. The Org items have an SK of ORG#<OrgName>. The User items have an SK of USER#<UserName>.

Notice that the Org item is right between the Team items and the User items. We had to specifically structure it this way using a # prefix to put the Team item ahead of the Org item in our item collection.

Now we could fetch the Org and all the Team items with the following Query:

```
result = dynamodb.query(
    TableName='SaaSTable',
    KeyConditionExpression="#pk = :pk AND #sk <= :sk",
    ExpressionAttributeNames={
        "#pk": "PK",
        "#sk": "SK"
    },
    ExpressionAttributeValues={
        ":pk": { "S": "ORG#MCDONALDS" },
        ":sk": { "S": "ORG#MCDONALDS" }
    },
  ScanIndexForward=False
)
```

This goes to our partition and finds all items less than or equal to than the sort key value for our Org item. Then it scans backward to pick up all the Team items.

Below is an image of how it works on the table:

Primary key		Attributes
Partition key: PK	**Sort key: SK**	
ORG#MCDONALDS	#TEAM#FANSOFISH	TeamName
		Fans O' Fish
	#TEAM#TEAMBURGER	TeamName
		Team Burger
	ORG#MCDONALDS	OrgName
		McDonalds
	USER#HAMBURGLAR	UserName
		Hamburglar
	USER#RONALD	UserName
		Ronald McDonald

2. Read backwards

1. Start here

You can do the reverse to fetch the Org item and all User items: look for all items greater than or equal to our Org item sort key, then read forward to pick up all the User items.

This is a more advanced pattern that is by no means necessary, but it will save you additional secondary indexes in your table. You can see a pattern of this in action in Chapter 21.

14.5. Zero-padding with numbers

You may occasionally want to order your items numerically even when the sort key type is a string. A common example of this is when you're using prefixes to indicate your item types. Your sort key might be `<ItemType>#<Number>`. While this is doable, you need to be careful about lexicographic sorting with numbers in strings.

As usual, this is best demonstrated with an example. Imagine in our IoT example above that, instead of using timestamps in the sort key, we used a reading number. Each device kept track of what

reading number it was on and sent that up with the sensor's value.

You might have a table that looks as follows:

Primary key		Attributes		
Partition key: PK	**Sort key: SK**			
DEVICE#123	#READING#1	Temperature		ReadingNumber
		32.1		1
	#READING#10	Temperature		ReadingNumber
		32.2		10
	#READING#2	Temperature		ReadingNumber
		32.2		2
	DEVICE#123	DeviceLocation		
		Omaha, NE		

Yikes, our readings are out of order. Reading number 10 is placed ahead of Reading number 2. This is because lexicographic sorting evalutes one character at a time, from left to right. When it is compared "10" to "2", the first digit of 10 ("1") is before the first digit of 2 ("2"), so 10 was placed before 2.

To avoid this, you can zero-pad your numbers. To do this, you make the number part of your sort key a fixed length, such as 5 digits. If your value doesn't have that many digits, you use a "0" to make it longer. Thus, "10" becomes "00010" and "2" becomes "00002".

Now our table looks as follows:

Primary key		Attributes	
Partition key: PK	Sort key: SK		
DEVICE#123	#READING#00001	Temperature	ReadingNumber
		32.1	1
	#READING#00002	Temperature	ReadingNumber
		32.2	2
	#READING#00010	Temperature	ReadingNumber
		32.2	10
	DEVICE#123	DeviceLocation	
		Omaha, NE	

Now Reading number 2 is placed before Reading number 10, as expected.

The big factor here is to make sure your padding is big enough to account for any growth. In our example, we used a fixed length of 5 digits which means we can go to Reading number 99999. If your needs are bigger than that, make your fixed length longer.

I'd recommend going to the maximum number of related items you could ever imagine someone having, then adding 2-3 digits beyond that. The cost is just an extra few characters in your item, but the cost of underestimating will add complexity to your data model later on. You may also want to have an alert condition in your application code that lets you know if a particular count gets to more than X% of your maximum, where X is probably 30 or so.

14.6. Faking ascending order

This last sorting pattern is a combination of a few previous ones, and it's pretty wild. Imagine you had a parent entity that had two

one-to-many relationships that you want to query. Further, imagine that both of those one-to-many relationships use a number for identification but that you want to fetch both relationships in the same order (either descending or ascending).

The problem here is that because you want to fetch the relationships in the same order, you would need to use two different item collections (and thus secondary indexes) to handle it. One item collection would handle the parent entity and the most recent related items of entity A, and the other item collection would handle the parent entity and the most recent related items of entity B.

However, we could actually get this in the same item collection using a *zero-padded difference*. This is similar to our zero-padded number, but we're storing the difference between the highest number available and the actual number of our item.

For example, imagine we were again using a zero-padded number with a width of 5. If we had an item with an ID of "157", the zero-padded number would be "00157".

To find the zero-padded difference, we subtract our number ("157") from the highest possible value ("99999"). Thus, the zero-padded difference would be "99842".

If we put this in our table, our items would look as follows:

Primary key		Attributes	
Partition key: PK	Sort key: SK		
DEVICE#123	DEVICE#123	DeviceLocation	
		Omaha, NE	
	READING#99989	Temperature	ReadingNumber
		32.2	10
	READING#99997	Temperature	ReadingNumber
		32.2	2
	READING#99998	Temperature	ReadingNumber
		32.1	1

Notice that I changed the `SK` structure of the Reading items so that the parent Device item is now at the top of our item collection. Now we can fetch the Device and the most recent Readings by starting at Device and reading *forward*, even though we're actually getting the readings in descending order according to their ReadingId.

This is a pretty wacky pattern, and it actually shows up in the GitHub Migration example in Chapter 19. That said, you may not ever have a need for this in practice. The best takeaway you can get from this strategy is how flexible DynamoDB can be if you combine multiple strategies. Once you learn the basics, you can glue them together in unique ways to solve your problem.

14.7. Conclusion

Strategy	Notes	Relevant examples
Using KSUIDs for sortable, unique IDs	Good for unique identifiers that still need sorted chronologically	GitHub migration example

Strategy	Notes	Relevant examples
Sorting on changing attributes	Use secondary indexes to avoid delete + create workflow	Leaderboards; most recently updated
Ascending vs. descending	Consider the order in which you'll access items when modeling relations	All relational examples
Two relational access patterns in a single item collection	Save money on secondary indexes by reusing an item collection	GitHub migration
Zero-padding with numbers	If sorting with integers, make sure to zero-pad if numbers are represented in a string	GitHub example

Table 11. Sorting strategies

Chapter 15. Strategies for Migrations

Chapter Summary

You need to model your DynamoDB table to match your access patterns. So what do you do when your access patterns change? This chapter contains strategies on adding new access patterns and migrating your data.

Sections

1. Adding new attributes to an existing entity

2. Adding a new entity without relations

3. Adding a new entity type into an existing item collection

4. Adding a new entity type into a new item collection

5. Joining existing items into a new item collection

6. Using parallel scans

In the previous chapters, I've harped over and over about how you need to know your access patterns before you model your table in DynamoDB. And it's true—DynamoDB is not flexible, but it's powerful. You can do almost anything with it, as long as you model it that way up front.

In the process of writing this book, I've had a number of people ask me about how to handle migrations. It's just not realistic to assume that your data model will be frozen in time. It's going to evolve.

In this chapter, we'll cover strategies for migrations as your data model changes. Migrations are intimidating at first, but they're entirely manageable. Once you've done one or two migrations, they will seem less scary.

To me, the fundamental question to ask yourself about a migration is whether it is *purely additive* or whether you need to *edit existing items*. A purely additive change means you can start writing the new attributes or items without changes to existing items. A purely additive change is much easier, while editing existing items usually requires a batch job to scan existing items and decorate them with new attributes.

Recall our discussion of indexing attributes and application attributes in Chapter 9. When considering whether a change is additive, you only need to consider indexing attributes, not application attributes. We'll walk through this in the subsequent sections.

Let's review some strategies for migrations.

15.1. Adding new attributes to an existing entity

The first, and easiest, type of migration is to add new attributes to existing entities. Perhaps you want to allow users to enter their birthday, so you add a `Birthdate` attribute to your existing User items. Or maybe you want to allow businesses to add a fax number to their contact information by adding a `FaxNumber` attribute.

When adding new attributes, you can simply add this in your application code without doing any large scale ETL process on your DynamoDB database. Recall from Chapter 9 that interacting with

DynamoDB should be at the very boundary of your application. As you retrieve data from DynamoDB, you should transform the returned items into objects in your application. As you transform them, you can add defaults for new attributes.

For example, look at the following code that may involve retrieving a User item in your application:

```python
def get_user(username):
    resp = client.get_item(
        TableName='ApplicationTable',
        Key={
            'PK': f"USER#{username}"
        }
    )
    return User(
        username=item['Username']['S'],
        name=item['Name']['S'],
        birthdate=item.get('Birthdate', {}).get('S') # Handles missing values
    )
```

Notice that we fetch the User item from our DynamoDB table and then return a User object for our application. When constructing the User object, we are careful to handle the absence of a `Birthdate` attribute for users that haven't added it yet.

Note that this strategy works even with one-to-many relationships when you're using a denormalization strategy. Because this strategy models a relationship as an attribute on the parent entity, you can add these relationships lazily in your application rather than worrying about changes to your DynamoDB items.

Adding application attributes is the easiest type of migration because you don't really think about application attributes while modeling your data in DynamoDB. You are primarily concerned with the attributes that are used for indexing within DynamoDB. The schemaless nature of DynamoDB makes it simple to add new attributes in your application without doing large-scale migrations.

For more on this strategy, check out the Code of Conduct addition

in the GitHub Migration example in Chapter 22.

15.2. Adding a new entity type without relations

While adding new attributes does happen, a far more common migration issue is when you're adding new entity types in your application. This happens as your application evolves and adds additional features, and you need new types of items to store details about those features.

There are a few different scenarios you see when adding a new entity type. The first one I want to discuss is when your new entity has no relations with existing entity types.

Now, it's rare that an entity won't have any relations at all on your ERD for your application. As Rick Houlihan says, all data is relational. It doesn't float around independently out in the ether. This scenario is more about when you don't have an access pattern that needs to *model* a relation in DynamoDB.

Imagine you have a new type of nested item in a one-to-many relationship, but you don't have any access patterns that are "Fetch the parent entity and its related entities". In this case, you don't need to worry about creating an item collection with an existing entity. You can simply start writing your new entity type and handle those access patterns directly.

In the previous chapter, we had a SaaS application that had Organizations and Users. Our table looked as follows:

Primary key		Attributes
Partition key: PK	Sort key: SK	
ORG#MCDONALDS	ORG#MCDONALDS	OrgName
		McDonalds
	USER#HAMBURGLAR	UserName
		Hamburglar
	USER#RONALDMCDONALD	UserName
		Ronald McDonald

Now imagine we have a new entity type: Projects. A Project belongs to an Organization, and an Organization can have multiple Projects. Thus, there is a one-to-many relationship between Organizations and Projects.

When reviewing our access patterns, we have a "Get Projects for Organization" access pattern, but we don't need to fetch the parent Organization item as part of it. Because of that, we can model the Projects into a new item collection altogether.

We might model our table as follows:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
ORG#MCDONALDS	ORG#MCDONALDS	OrgName		
		McDonalds		
	USER#HAMBURGLAR	UserName		
		Hamburglar		
	USER#RONALDMCDONALD	UserName		
		Ronald McDonald		
ORGPROJECTS#MCDONALDS	PROJECT#DUCKBURGER	OrgName	ProjectName	ProjectTasks
		McDonalds	DuckBurger	43
	PROJECT#NEWWEBSITE	OrgName	ProjectName	ProjectTasks
		McDonalds	New Website	14

We've added our Project items as outlined in red at the bottom. Notice that all the Project items are in the same item collection as they share the same partition key, but they aren't in the same item collection as any existing items. We didn't have to make any changes to *existing* items to handle this new entity and its access patterns, so it's a purely additive change. Like the addition of an attribute, you just update your application code and move on.

15.3. Adding a new entity type into an existing item collection

Let's move on to a more common but still straightforward example. This one is similar to the last in that you're adding a new entity type into your table, but it's different in that you do have a relational access pattern of "Fetch parent entity and its related entities".

In this scenario, you can try to *reuse* an existing item collection. This is great if your parent entity is in an item collection that's not being used for an existing relationship.

As an example, think about an application like Facebook before they introduced the "Like" button. Their DynamoDB table may have had Post items that represented a particular post that a user made.

The basic table might look as follows:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
POST#02f67ffa9a8a	POST#02f67ffa9a8a	PostOwner	PostContent	PostDate
		Alex DeBrie	https://....	2020-01-12 19:30:39
POST#faa8eaf4af67	POST#faa8eaf4af67	PostOwner	PostContent	PostDate
		Mark Zuckerberg	https://....	2020-02-07 21:59:38
POST#9f14e02cf927	POST#9f14e02cf927	PostOwner	PostContent	PostDate
		Lindsay Lohan	https://...	2020-02-23 21:17:49

Then someone has the brilliant idea to allow users to "like" posts via the Like button. When we want to add this feature, we have an access pattern of "Fetch post and likes for post".

In this situation, the Like entity is a completely new entity type, and we want to fetch it at the same time as fetching the Post entity. If we look at our base table, the item collection for the Post entity isn't being used for anything. We can add our Like items into that collection by using the following primary key pattern for Likes:

- **PK**: `POST#<PostId>`
- **SK**: `LIKE#<Username>`

When we add a few items to our table, it looks like this:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
POST#02f67ffa9a8a	LIKE#elsiedebrie	LikingUser		
		Elsie DeBrie		
	LIKE#markzuckerberg	LikingUser		
		Mark Zuckerberg		
	POST#02f67ffa9a8a	PostOwner	PostContent	PostDate
		Alex DeBrie	https://....	2020-01-12 19:30:39
POST#faa8eaf4af67	POST#faa8eaf4af67	PostOwner	PostContent	PostDate
		Mark Zuckerberg	https://....	2020-02-07 21:59:38
POST#9f14e02cf927	POST#9f14e02cf927	PostOwner	PostContent	PostDate
		Lindsay Lohan	https://...	2020-02-23 21:17:49

Outlined in red is an item collection with both a Post and the Likes
for that Post. Like our previous examples, this is a purely additive
change that didn't require making changes to our existing Post
items. All that was required is that we modeled our data
intentionally to locate the Like items into the existing item
collection for Posts.

15.4. Adding a new entity type into a new item collection

The easy stuff is over. Now it's time to look at something harder.

This section is similar to the previous one. We have a new item type
and a relational access pattern with an existing type. However,
there's a twist—we don't have an existing item collection where we
can handle this relational access pattern.

Let's continue to use the example from the last section. We have a social application with Posts and Likes. Now we want to add Comments. Users can comment on a Post to give encouragement or argue about some pedantic political point.

With our new Comment entity, we have a relational access pattern where we want to fetch a Post and the most recent Comments for that Post. How can we model this new pattern?

The Post item collection is already being used on the base table. To handle this access pattern, we'll need to create a new item collection in a global secondary index.

To do this, let's add the following attributes to the Post item:

- **GSI1PK:** `POST#<PostId>`
- **GSI1SK:** `POST#<PostId>`

And we'll create a Comment item with the following attributes:

- **PK:** `COMMENT#<CommentId>`
- **SK:** `COMMENT#<CommentId>`
- **GSI1PK:** `POST#<PostId>`
- **GSI1SK:** `COMMENT#<Timestamp>`

When we do that, our base table looks as follows:

Primary key		Attributes					
Partition key: PK	Sort key: SK						
POST#02f67ffa9a8a	POST#02f67ffa9a8a	PostOwner	PostContent	PostDate	GSI1PK	GSI1SK	
		Alex DeBrie	https://....	2020-01-12 19:30:39	POST#02f67ffa9a8a	POST#02f67ffa9a8a	
POST#faa8eaf4af67	POST#faa8eaf4af67	PostOwner	PostContent	PostDate	GSI1PK	GSI1SK	
		Mark Zuckerberg	https://....	2020-02-07 21:59:38	POST#faa8eaf4af67	POST#faa8eaf4af67	
POST#9f14e02cf927	POST#9f14e02cf927	PostOwner	PostContent	PostDate	GSI1PK	GSI1SK	
		Lindsay Lohan	https://...	2020-02-23 21:17:49	POST#9f14e02cf927	POST#9f14e02cf927	
COMMENT#f0e27e50-8144-4c17-9a2a-c50bdb94fb0f	COMMENT#f0e27e50-8144-4c17-9a2a-c50bdb94fb0f	CreatedAt	CommentingUser	GSI1PK	GSI1SK		
		2020-02-02 10:16:46	Sheryl Sandberg	POST#02f67ffa9a8a	COMMENT#2020-02-02 10:16:46		
COMMENT#710bdfc5-9b5f-4e9e-b00e-825674eb9314	COMMENT#710bdfc5-9b5f-4e9e-b00e-825674eb9314	CreatedAt	CommentingUser	GSI1PK	GSI1SK		
		2020-02-09 13:51:34	Bill Gates	POST#02f67ffa9a8a	COMMENT#2020-02-09 13:51:34		

Notice that we've added two Comment items at the bottom outlined in blue. We've also add two attributes, `GSI1PK` and `GSI1SK`, to our Post items, outlined in red .

When we switch to our secondary index, it looks as follows:

Primary key		Attributes				
Partition key: GSI1PK	Sort key: GSI1SK					
POST#02f67ffa9a8a	COMMENT#2020-02-02 10:16:46	PK	SK	CreatedAt	CommentingUser	
		COMMENT#f0e27e50-8144-4c17-9a2a-c50bdb94fb0f	COMMENT#f0e27e50-8144-4c17-9a2a-c50bdb94fb0f	2020-02-02 10:16:46	Sheryl Sandberg	
	COMMENT#2020-02-09 13:51:34	PK	SK	CreatedAt	CommentingUser	
		COMMENT#710bdfc5-9b5f-4e9e-b00e-825674eb9314	COMMENT#710bdfc5-9b5f-4e9e-b00e-825674eb9314	2020-02-09 13:51:34	Bill Gates	
	POST#02f67ffa9a8a	PK	SK	PostOwner	PostContent	PostDate
		POST#02f67ffa9a8a	POST#02f67ffa9a8a	Alex DeBrie	https://....	2020-01-12 19:30:39
POST#faa8eaf4af67	POST#faa8eaf4af67	PK	SK	PostOwner	PostContent	PostDate
		POST#faa8eaf4af67	POST#faa8eaf4af67	Mark Zuckerberg	https://....	2020-02-07 21:59:38
POST#9f14e02cf927	POST#9f14e02cf927	PK	SK	PostOwner	PostContent	PostDate
		POST#9f14e02cf927	POST#9f14e02cf927	Lindsay Lohan	https://...	2020-02-23 21:17:49

Our Post item is in the same item collection as our Comment items, allowing us to handle our relational use case.

This looks easy in the data model, but I've skipped over the hard part. How do you decorate the existing Post items to add `GSI1PK` and `GSI1SK` attributes?

You will need to run a table scan on your table and update each of the Post items to add these new attributes. A simplified version of the code is as follows:

```python
last_evaluated = ''

params = {
    "TableName": "SocialNetwork",
    "FilterExpression": "#type = :type",
    "ExpressionAttributeNames": {
        "#type": "Type"
    },
    "ExpressionAttributeValues": {
        ":type": { "S": "Post" }
    }
}

while True:
    # 1. Scan to find our items
    if last_evaluated:
        params['ExclusiveStartKey'] = last_evaluated

    results = client.scan(**params)

    # 2. For each item found, update with new attributes.
    for item in results['Items']:
        client.update_item(
            TableName='SocialNetwork',
            Key={
                'PK': item['PK'],
                'SK': item['SK']
            },
            UpdateExpression="SET #gsi1pk = :gsi1pk, #gsi1sk = :gsi1sk",
            ExpressionAttributeNames={
                '#gsi1pk': 'GSI1PK',
                '#gsi1sk': 'GSI1SK'
            }
            ExpressionAttributeValues={
                ':gsi1pk': item['PK'],
                ':gsi1sk': item['SK']
            }
        )

    # 3. Check to see if there are additional pages to process in Scan.
    if not results['LastEvaluatedKey']:
        break
    last_evaluated = results['LastEvaluatedKey']
```

This script is running a Scan API action against our DynamoDB table. It's using a filter expression to filter out any items whose `Type` attribute is not equal to `Post`, as we're only updating the Post items in this job.

As we receive items from our Scan result, we iterate over those items and make an UpdateItem API request to add the relevant

properties to our existing items.

There's some additional work to handle the `LastEvaluatedKey` value that is received in a Scan response. This indicates whether we have additional items to scan or if we've reached the end of the table.

There are a few things you'd want to do to make this better, including using parallel scans, adding error handling, and updating multiple items in a BatchWriteItem request, but this is the general shape of your ETL process. There is a note on parallel scans at the end of this chapter.

This is the hardest part of a migration, and you'll want to test your code thoroughly and monitor the job carefully to ensure all goes well. However, there's really not that much going on. A lot of this can be parameterized:

- How do I know which items I want?
- Once I get my items, what new attributes do I need to add?.

From there, you just need to take the time for the whole update operation to run.

15.5. Joining existing items into a new item collection

So far, all of the examples have involved adding a new item type into our application. But what if we just have a new access pattern on existing types? Perhaps we want to filter existing items in a different way. Or maybe we want to use a different sorting mechanism. We may even want to join two items that previously were separate.

The pattern here is similar to the last section. Find the items you want to update and design an item collection in a new or existing secondary index. Then, run your script to add the new attributes so they'll be added to the secondary index.

15.6. Using parallel scans

I mentioned above that you may want to use parallel scans in your ETL scripts. This might sound scary—how do I do this in parallel? Am I manually maintaining state across multiple independent scanners?

Fortunately, DynamoDB makes parallel scans easy. When you're doing a Scan operation, there are two optional properties:

- **Total Segments:** This indicates the total number of segments you want to split your Scan across

- **Segment:** This indicates the segment to be scanned by *this particular worker*.

In the ETL examples shown above, you would update your Scan parameters to have the following:

```
params = {
    "TableName": "SocialNetwork",
    "FilterExpression": "#type = :type",
    "ExpressionAttributeNames": {
        "#type": "Type"
    },
    "ExpressionAttributeValues": {
        ":type": "Post"
    },
    "TotalSegments": 10,
    "Segment": 0
}
```

With the two new parameters at the bottom, I'm indicating that I

want to split my Scan across 10 workers and that this is worker 0 processing the table. I would have nine other workers processing segments 1 - 9 as well.

This will greatly speed up your ETL processing without adding a lot of complexity. DynamoDB will handle all of the state management for you to ensure every item is handled.

15.7. Conclusion

In this chapter, we reviewed how to handle migrations. Below is a summary of migration situations and strategies.

Situation	Notes	Relevant examples
Adding new attributes to existing entity	Lazily add attributes	GitHub Migration (Codes of Conduct)
Adding a new entity without relations	Write items to new item collection	SaaS table with Projects
Adding a new entity into an existing item collection	Model items to go to existing item collection	GitHub Migration (Gists)
Adding a new entity into a new item collection	Update existing items to add new attributes. Model new items into same item collection	GitHub Migration (Apps)
Joining existing items into a new item collection	Update existing items to add new attributes.	GitHub Migration (Issues & Pull Requests)

Table 12. Migration situations

Chapter 16. Additional strategies

Chapter Summary

This chapter contains additional strategies that are useful but don't fit into one of the previous categories.

Sections

1. Ensuring uniqueness on two or more attributes
2. Handling sequential IDs
3. Pagination
4. Singleton items
5. Reference counts

In the last five chapters, we've covered strategies across a number of areas. Relationships, filtering, sorting, migrations—these are the core of data modeling.

There are a few other strategies I want to cover that don't fit neatly into one of the five previous chapters. Hence, you get an "additional strategies" chapter. They're a bit of a grab bag but still useful.

The strategies we'll cover in this chapter are:

- Ensuring uniqueness on two or more attributes
- Handling sequential IDs

- Pagination

- Singleton items

- Reference counts

Let's have some fun.

16.1. Ensuring uniqueness on two or more attributes

In DynamoDB, if you want to ensure a particular attribute is unique, you need to build that attribute directly into your primary key structure.

An easy example here is a user signup flow for an application. You want the username to be unique across your application, so you build a primary key that includes the username.

Primary key		Attributes			
Partition key: PK	Sort key: SK				
USER#alexdebrie	USER#alexdebrie	Username	FirstName	LastName	DateOfBirth
		alexdebrie	Alex	DeBrie	May 26, 1988
USER#thevoice	USER#thevoice	Username	FirstName	LastName	DateOfBirth
		thevoice	Whitney	Houston	August 9, 1963
USER#olblueeyes	USER#olblueeyes	Username	FirstName	LastName	DateOfBirth
		olblueeyes	Frank	Sinatra	December 12, 1915

In the table above, our PK and SK values both include the username so that it will be unique. When creating a new user, I'll use a condition expression to ensure a user with the same username doesn't exist.

But what if you also want to ensure that a given email address is

unique across your system so that you don't have people signing up for multiple accounts to the same email address?

You might be tempted to add email into your primary key as well:

Primary key		Attributes			
Partition key: PK	Sort key: SK				
USER#alexdebrie	USER#alex@debrie.com	Username	FirstName	LastName	DateOfBirth
		alexdebrie	Alex	DeBrie	May 26, 1988
USER#thevoice	USER#hello@whitney.com	Username	FirstName	LastName	DateOfBirth
		thevoice	Whitney	Houston	August 9, 1963
USER#olblueeyes	USER#frankthetank@sinatra.com	Username	FirstName	LastName	DateOfBirth
		olblueeyes	Frank	Sinatra	December 12, 1915

Now our table includes the username in the PK and the email in the SK.

However, this won't work. It's the *combination* of a partition key and sort key that makes an item unique within the table. Using this key structure, you're confirming that an email address will only be used once for a given username. Now you've lost the original uniqueness properties on the username, as someone else could sign up with the same username and a different email address!

If you want to ensure that both a username and an email address are unique across your table, you need to use a transaction.

We discussed transactions back in Chapter 4 but here's a quick refresher. In a DynamoDB transaction, you can include write operations on multiple items in a single request, and the entire request will succeed or fail together.

In this example, we'll create two items in a transaction: one that tracks the user by username, and one that tracks the email by username.

The code to write such a transaction would be as follows:

```
response = client.transact_write_items(
    TransactItems=[
        {
            'Put': {
                'TableName': 'UsersTable',
                'Item': {
                    'PK': { 'S': 'USER#alexdebrie' },
                    'SK': { 'S': 'USER#alexdebrie' },
                    'Username': { 'S': 'alexdebrie' },
                    'FirstName': { 'S': 'Alex' },
                    ...
                },
                'ConditionExpression': 'attribute_not_exists(PK)
            }
        },
        {
            'Put': {
                'TableName': 'UsersTable',
                'Item': {
                    'PK': { 'S': 'USEREMAIL#alex@debrie.com' },
                    'SK': { 'S': 'USEREMAIL#alex@debrie.com' },
                },
                'ConditionExpression': 'attribute_not_exists(PK)
            }
        }
    ]
)
```

For each write operation, we're including a condition expression that ensures that an item with that primary key doesn't exist. This confirms that the username is not currently in use and that the email address is not in use.

And now your table would look as follows:

Primary key		Attributes			
Partition key: PK	Sort key: SK				
USER#alexdebrie	USER#alexdebrie	Username	FirstName	LastName	DateOfBirth
		alexdebrie	Alex	DeBrie	May 26, 1988
USER#thevoice	USER#thevoice	Username	FirstName	LastName	DateOfBirth
		thevoice	Whitney	Houston	August 9, 1963
USER#olblueeyes	USER#olblueeyes	Username	FirstName	LastName	DateOfBirth
		olblueeyes	Frank	Sinatra	December 12, 1915
USEREMAIL#alex@debrie.com	USEREMAIL#alex@debrie.com				
USEREMAIL#hello@whitney.com	USEREMAIL#hello@whitney.com				
USEREMAIL#frankthetank@sinatra.com	USEREMAIL#frankthetank@sinatra.com				

Notice that the item that stores a user by email address doesn't have any of the user's properties on it. You can do this if you will only access a user by a username and never by an email address. The email address item is essentially just a marker that tracks whether the email has been used.

If you will access a user by email address, then you need to duplicate all information across both items. Then your table might look as follows:

PK (Partition key) : String ⇕	SK (Sort key) : String ⇕	Username : String ⇕	FirstName : String ⇕	LastName : String ⇕	DateOfBirth : String ⇕
USER#alexdebrie	USER#alexdebrie	alexdebrie	Alex	DeBrie	May 26, 1988
USER#thevoice	USER#thevoice	thevoice	Whitney	Houston	August 9, 1963
USER#olblueeyes	USER#olblueeyes	olblueeyes	Frank	Sinatra	December 12, 1915
USEREMAIL#alex@...	USEREMAIL#alex@...	alexdebrie	Alex	DeBrie	May 26, 1988
USEREMAIL#hello@...	USEREMAIL#hello@...	thevoice	Whitney	Houston	August 9, 1963
USEREMAIL#frankth...	USEREMAIL#frankt...	olblueeyes	Frank	Sinatra	December 12, 1915

I'd avoid this if possible. Now every update to the user item needs to be a transaction to update both items. It will increase the cost of your writes and the latency on your requests.

To see this strategy in action, check out the E-commerce example

in Chapter 19.

16.2. Handling sequential IDs

In relational database systems, you often use a sequential ID as a primary key identifier for each row in a table. With DynamoDB, this is not the case. You use meaningful identifiers, like usernames, product names, etc., as unique identifiers for your items.

That said, sometimes there are user-facing reasons for using sequential identifiers. Perhaps your users are creating entities, and it's easiest to assign them sequential identifiers to keep track of them. Examples here are Jira tickets, GitHub issues, or order numbers.

DynamoDB doesn't have a built-in way to handle sequential identifiers for new items, but you can make it work through a combination of a few building blocks.

Let's use a project tracking software like Jira as an example. In Jira, you create Projects, and each Project has multiple Issues. Issues are given a sequential number to identify them. The first issue in a project would be Issue #1, the second issue would be Issue #2, etc.

When a new issue is created within a project, we'll do a two-step process.

First, we will run an UpdateItem operation on the Project item to increment the `IssueCount` attribute by 1. We'll also set the `ReturnValues` parameter to `UPDATED_NEW`, which will return the current value of all updated attributes in the operation. At the end of this operation, we will know what number should be used for our new issue. Then, we'll create our new Issue item with the new issue number.

The full operation looks as follows:

```python
resp = client.update_item(
    TableName='JiraTable',
    Key={
        'PK': { 'S': 'PROJECT#my-project' },
        'SK': { 'S': 'PROJECT#my-project' }
    },
    UpdateExpression="SET #count = #count + :incr",
    ExpressionAttributeNames={
        "#count": "IssueCount",
    },
    ExpressionAttributeValues={
        ":incr": { "N": "1" }
    },
    ReturnValues='UPDATED_NEW'
)

current_count = resp['Attributes']['IssueCount']['N']

resp = client.put_item(
    TableName='JiraTable',
    Item={
        'PK': { 'S': 'PROJECT#my-project' },
        'SK': { 'S': f"ISSUE#{current_count}" },
        'IssueTitle': { 'S': 'Build DynamoDB data model' }
        ... other attributes ...
    }
)
```

First we increment the `IssueCount` on our Project item. Then we use the updated value in the PutItem operation to create our Issue item.

This isn't the best since you're making two requests to DynamoDB in a single access pattern. However, it can be a way to handle auto-incrementing IDs when you need them.

16.3. Pagination

Pagination is a common requirement for APIs. A request will often return the first 10 or so results, and you can receive additional results by making a follow-up request and specifying the page.

In a relational database, you may use a combination of `OFFSET` and `LIMIT` to handle pagination. DynamoDB does pagination a little differently, but it's pretty straightforward.

When talking about pagination with DynamoDB, you're usually paginating within a single item collection. You're likely doing a Query operation within an item collection to fetch multiple items at once.

Let's walk through this with an example. Imagine you have an e-commerce application where users make multiple orders. Your table might look like this:

Primary key		Attributes	
Partition key: PK	Sort key: SK		
USER#alexdebrie	ORDER#1YRfXRYymimDqUhRZJCAWDHV7Nq	OrderId	Amount
		1YRfXRYymimDqUhRZJCAWDHV7Nq	23.88
	ORDER#1YRfXS14inXwIJEf9tO5hWnL2pi	OrderId	Amount
		1YRfXS14inXwIJEf9tO5hWnL2pi	400.54
	ORDER#1YRfXSd5AvBnEB3G6MOFt6UeMKW	OrderId	Amount
		1YRfXSd5AvBnEB3G6MOFt6UeMKW	78.14
	ORDER#1YRfXTGWS0i9DcC0UajS1dn92Ou	OrderId	Amount
		1YRfXTGWS0i9DcC0UajS1dn92Ou	61.58
	ORDER#1YRfXVZ39ORLkmJKlfSNvQSxA6Q	OrderId	Amount
		1YRfXVZ39ORLkmJKlfSNvQSxA6Q	14.99
	ORDER#1YRfXVhH1mFsKjbLZ9b0bTtgDYg	OrderId	Amount
		1YRfXVhH1mFsKjbLZ9b0bTtgDYg	68.14

Our table includes Order items. Each Order item uses a PK of `USER#<username>` for grouping and then uses an OrderId in the sort key. The OrderId is a KSUID (see the notes in Chapter 14 for notes on KSUIDs) which gives them rough chronological ordering.

A key access pattern is to fetch the most recent orders for a user. We only return five items per request, which seems very small but it makes this example must easier to display. On the first request, we would write a Query as follows:

```python
resp = client.query(
    TableName='Ecommerce',
    KeyConditionExpression='#pk = :pk, #sk < :sk',
    ExpressionAttributeNames={
        '#pk': 'PK',
        '#sk': 'SK'
    },
    ExpressionAttributeValues={
        ':pk': 'USER#alexdebrie',
        ':sk': 'ORDER$'
    },
    ScanIndexForward=False,
    Limit=5
)
```

This request uses our partition key to find the proper partition, and it looks for all values where the sort key is less than `ORDER$`. This will come immediately *after* our Order items, which all start with `ORDER#`. Then, we use `ScanIndexForward=False` to read items backward (reverse chronological order) and set a limit of 5 items.

It would retrieve the items outlined in red below:

This works for the first page of items but what if a user makes follow-up requests?

In making that follow-up request, the URL would be something like

`https://my-ecommerce-store.com/users/alexdebrie/orders?before=1YRfXS14inXwIJEf9tO5hWnL2pi`. Notice how it includes both the username (in the URL path) as well as the last seen OrderId (in a query parameter).

We can use these values to update our Query:

```python
resp = client.query(
    TableName='Ecommerce',
    KeyConditionExpression='#pk = :pk, #sk < :sk',
    ExpressionAttributeNames={
        '#pk': 'PK',
        '#sk': 'SK'
    },
    ExpressionAttributeValues={
        ':pk': 'USER#alexdebrie',
        ':sk': 'ORDER#1YRfXS14inXwIJEf9tO5hWnL2pi'
    },
    ScanIndexForward=False,
    Limit=5
)
```

Notice how the `SK` value we're comparing to is now the Order item that we last saw. This will get us the next item in the item collection, returning the following result:

Partition key: PK	Sort key: SK	Attributes	
USER#alexdebrie	ORDER#1YRfXRYymimDqUhRZJCAWDHV7Nq	OrderId	Amount
		1YRfXRYymimDqUhRZJCAWDHV7Nq	23.88
	ORDER#1YRfXS14inXwIJEf9tO5hWnL2pi	OrderId	Amount
		1YRfXS14inXwIJEf9tO5hWnL2pi	400.54
	ORDER#1YRfXSd5AvBnEB3G6MOFt6UeMKW	OrderId	Amount
		1YRfXSd5AvBnEB3G6MOFt6UeMKW	78.14
	ORDER#1YRfXTGWS0i9DcC0UajS1dn92Ou	OrderId	Amount
		1YRfXTGWS0i9DcC0UajS1dn92Ou	61.58
	ORDER#1YRfXVZ39ORLkmJKlfSNvQSxA6Q	OrderId	Amount
		1YRfXVZ39ORLkmJKlfSNvQSxA6Q	14.99
	ORDER#1YRfXVhH1mFsKjbLZ9b0bTtgDYg	OrderId	Amount
		1YRfXVhH1mFsKjbLZ9b0bTtgDYg	68.14

By building these hints into your URL, you can discover where to

start for your next page through your item collection.

16.4. Singleton items

In most of our examples, we create an item pattern that is reused for the primary key across multiple kinds of items.

If you're creating a User item, the pattern might be:

- **PK:** `USER#<Username>`
- **SK:** `USER#<Username>`

If you're creating an Order item for the user, the pattern might be:

- **PK:** `USER#<Username>`
- **SK:** `ORDER#<OrderId>`

Notice that each item is customized for the particular user or order by virtue of the username and/or order id.

But sometimes you don't need a customized item. Sometimes you need a single item that applies across your entire application.

For example, maybe you have a table that is tracking some background jobs that are running in your application. Across the entire application, you want to have at most 100 jobs running at any given time.

To track this, you could create a *singleton item* that is responsible for tracking all jobs in progress across the application. Your table might look as follows:

Primary key		Attributes	
Partition key: PK	Sort key: SK		
JOBS	JOBS	JobsInProgress	
		["15cfea119a65","ff8a2f338373"]	
JOB#15cfea119a65	JOB#15cfea119a65	JobId	JobStatus
		15cfea119a65	IN_PROGRESS
JOB#ff8a2f338373	JOB#ff8a2f338373	JobId	JobStatus
		ff8a2f338373	IN_PROGRESS
JOB#818ec6a97dee	JOB#818ec6a97dee	JobId	JobStatus
		818ec6a97dee	WAITING

We have three Job items outlined in red. Further, there is a singleton item with a PK and SK of `JOBS` that tracks the existing jobs in progress via a `JobsInProgress` attribute of type string set. When we want to start a new job, we would do a transaction with two write operations:

1. Update the `JobsInProgress` attribute of the singleton Jobs item to add the new Job ID to the set *if* the current length of the set is less than 100, the max number of jobs in progress.

2. Update the relevant Job item to set the status to `IN_PROGRESS`.

Another example for using singleton items is in the Big Time Deals example in Chapter 20 where we use singleton items for the front page of the application.

16.5. Reference counts

We talked about one-to-many relationships and many-to-many relationships in previous chapters. These are very common

patterns in your application.

Often you'll have a parent item in your application that has a large number of related items through a relationship. It could be the number of retweets that a tweet receives or the number of stars for a GitHub repo.

As you show that parent item in the UI, you may want to show a reference count of the number of related items for the parent. We want to show the total number of retweets and likes for the tweet and the total number of stars for the GitHub repo.

We could query and count the related items each time, but that would be highly inefficient. A parent could have thousands or more related items, and this would burn a lot of read capacity units just to receive an integer indicating the total.

Instead, we'll keep a count of these related items as we go. Whenever we add a related item, we usually want to do two things:

1. Ensure the related item doesn't already exist (e.g. this particular user hasn't already starred this repo);
2. Increase the reference count on the parent item.

Note that we only want to allow it to proceed if *both* portions succeed. This is a great case for DynamoDB Transactions! Remember that DynamoDB Transactions allow you to combine multiple operations in a single request, and the operations will only be applied if *all* operations succeed.

The code to handle this would look as follows:

```
result = dynamodb.transact_write_items(
    TransactItems=[
        {
            "Put": {
                "Item": {
                    "PK": { "S": "REPO#alexdebrie#dynamodb-book" },
                    "SK": { "S": "STAR#danny-developer" }
                    ...rest of attributes ...
                },
                "TableName": "GitHubModel",
                "ConditionExpression": "attribute_not_exists(PK)"
            }
        },
        {
            "Update": {
                "Key": {
                    "PK": { "S": "REPO#alexdebrie#dynamodb-book" },
                    "SK": { "S": "#REPO#alexdebrie#dynamodb-book" }
                },
                "TableName": "GitHubModel",
                "ConditionExpression": "attribute_exists(PK)",
                "UpdateExpression": "SET #count = #count + :incr",
                "ExpressionAttributeNames": {
                    "#count": "StarCount"
                },
                "ExpressionAttributeValues": {
                    ":incr": { "N": "1" }
                }
            }
        }
    ]
)
```

Notice that in our transaction, we're doing a PutItem operation to insert the Star entity to track this user starring a repo. We include a condition expression to ensure the user hasn't already starred the repo. Additionally, we have an UpdateItem operation that increments the `StarCount` attribute on our parent Repo item.

I use this reference count strategy quite a bit in my applications, and DynamoDB Transactions have made it much easier than the previous mechanism of editing multiple items and applying manual rollbacks in the event of failure.

16.6. Conclusion

In this chapter we covered additional strategies for DynamoDB. The strategies are summarized below.

Strategy	Notes	Relevant examples
Ensuring uniqueness on two or more attributes	Create a tracking item and use DynamoDB Transactions	E-commerce example (Users with unique email and username)
Handling sequential IDs	Track current ID in attribute. Increment that attribute and use returned value to create new item.	GitHub Issues & Pull Requests
Pagination	Build pagination elements into URL structure	E-commerce orders
Singleton items	Use for tracking global state or for assembling a meta view	Deals example
Reference counts	Use a transaction to maintain a count of related items on a parent item	Deals example; GitHub example

Table 13. Additional strategies

Chapter 17. Data modeling examples

In the remaining chapters, we'll do deep dives into actual data modeling examples. I think these are the most important chapters in the book, and there are two things I want you to get from these chapters.

The first takeaway is about process. Take the steps from Chapter 7 and apply them to an actual problem.

Understand your application.

Create your ERD.

Understand your access patterns.

Design your primary key and secondary indexes to satisfy those access patterns.

Do your best to try to build the data model with me. Your approach might not look exactly like mine, but it's worth going through the exercise.

The second thing I want you to get from these examples is the various strategies applied in the chapters. It is unlikely that any example you find in this book or anywhere else will exactly match the needs of your application. However, you can take the strategies applied in the following chapters and apply them to sections of your data model. Collect these strategies and learn how to apply them in the proper situations.

17.1. Notes on the data modeling examples

Below are a few notes on how I've organized the examples to help orient you.

Each example starts off with an explanation of the problem we're trying to solve. I'm trying to get you in the head of a developer who is working with a PM or a business analyst to figure out what needs to be built. This includes notes on what we're building, any constraints we have, and images or mockups of our finished product.

As you read through those, get used to listing out the access patterns you'll have in your application. Try to understand the needs of your user and meld it with the possibilties of DynamoDB.

As I list the access patterns out, I don't always include *every* potential access pattern. If there are basic Read / Write / Update access patterns on individual items, I won't list them out *unless* there's something interesting.

If you've worked with ERDs before, you may see ERDs that include all attributes on each entity. I don't do that in my ERDs for a few reasons. First, it adds a lot of clutter to the ERDs. Some of these diagrams are large, complicated models with more than ten entities and over fifteen relationships. To add attributes for each of those would be more trouble than it's worth.

Second, I don't add attributes on my ERD because it's just not relevant to data modeling with DynamoDB. The attributes are more about what's needed in my *application*, whereas the ERD is more about understanding how these different objects relate to each other.

Likewise, when showing examples of items in DynamoDB, I don't include all the attributes on an item. Recall in Chapter 9 the difference between *application attributes* and *indexing attributes*. Application attributes are useful in your application, whereas indexing attributes are added solely for data access in DynamoDB.

In the data modeling phase of building an application, we're mostly concerned about the indexing attributes. We're trying to build a model that allows for fast, efficient data access in DynamoDB. Don't think that the absence of application attributes means that those attributes won't be present on our items—we're just hiding them to reduce complexity.

17.2. Conclusion

The data modeling examples are a lot of fun. You finally get to put all your learning to work. Try your best, and refer back to the strategies chapters where necessary to understand the examples more deeply.

Chapter 18. Building a Session Store

It's time for our first data modeling example. We'll start with something simple and get progressively more complex in subsequent examples.

This example will show how to build a session store for an application that includes an authentication element. To model this out, we'll use a simple primary key and a single secondary index in our DynamoDB table. Simple primary keys can be great for basic key-value access patterns where you're only working with a single item at a time. However, you are limited in the amount of complexity you can handle with a simple primary key.

Because the concepts of simple primary keys are pretty straightforward, this is the only example that uses a simple primary key.

18.1. Introduction

Imagine you are building the next hot social network application. As part of your application, you need user accounts with authentication and authorization. To handle this, you'll use a session token.

The basic flow is as follows. First, there is the token creation flow, as shown below.

"Can I log in? My
username is
alexdebrie and my
password is
ilovedynamodb"

user

Backend service

Your password is
correct! Here, use
this token the next
time you make a
request:
"j14j0fl.."
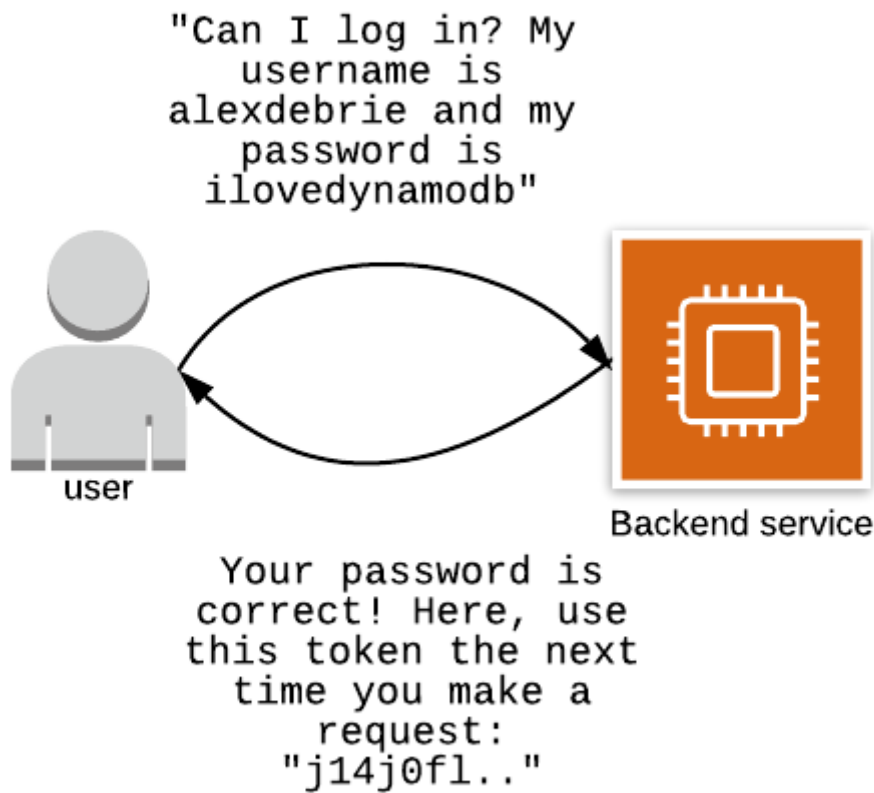
Users will login to your application by providing a username and password. Your backend service will verify the username and password. If the username and password are correct, the backend will create a session token, store it in a database, and return it to the user.

On subsequent requests, your application clients will include the session token in the Authorization header. This flow looks as follows:
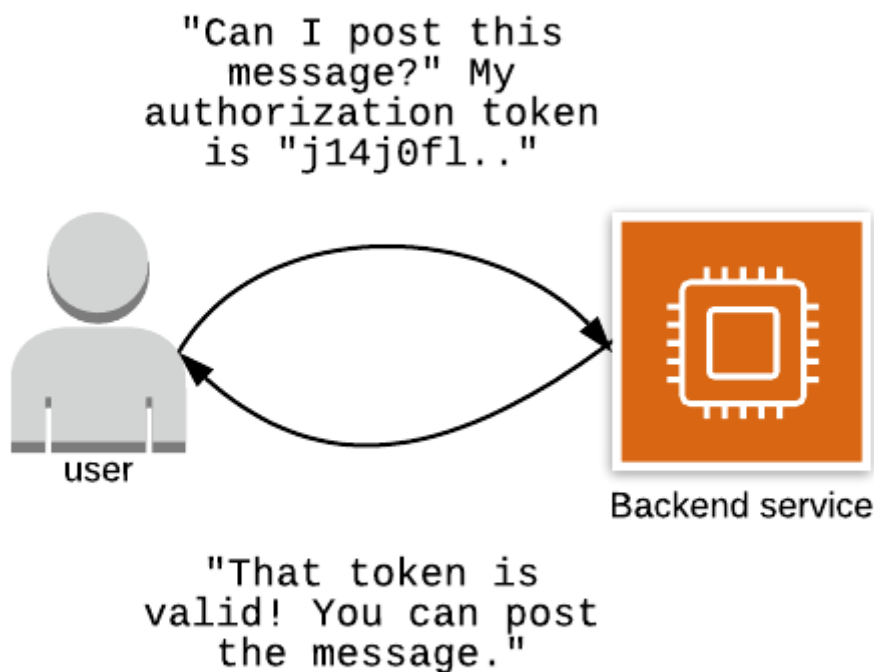
Your backend will check the session token in the database to verify that the token exists and validate the user making the request.

You don't want these tokens to live forever. The longer a token lives, the higher chance that a malicious actor will get it. As such, your application will invalidate a token in two instances:

- **Time-based:** A session token will automatically expire 7 days after it is created.

- **Manual revocation:** A user or application admin can choose to invalidate all outstanding tokens for the user if there is concern that the account has been compromised.

With these requirements in mind, let's build our ERD and list our access patterns.

18.2. ERD and Access Patterns

The first step to modeling in DynamoDB is to build an ERD. Our ERD for the session store is as follows:
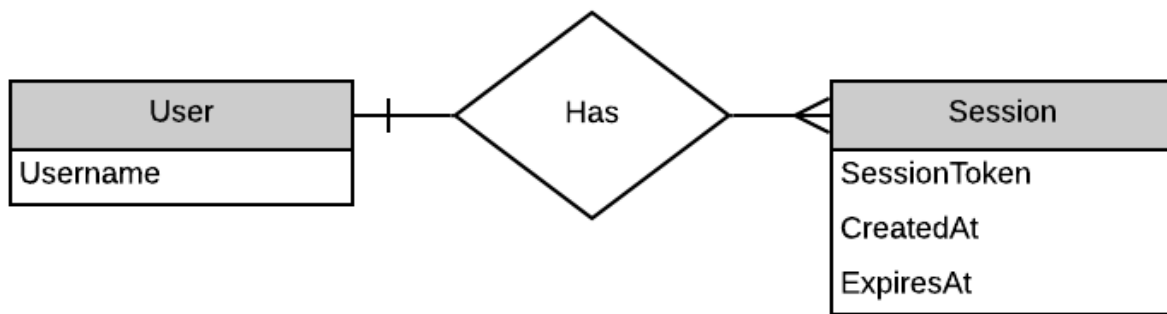
This is a pretty simple ERD. The core of the data model is going to be the Session. A Session will be primarily identified and accessed by its `SessionToken`. Additionally, a Session belongs to a User, and a User can have multiple Sessions, such as if the User is using the application on multiple devices.

After building my ERD, I make an entity chart that will be used to track the primary key patterns of each entity in my ERD. The entity chart for this application is as follows:

Entity	PK	SK
Session		
User		

Table 14. Session store entity chart

Now that we have our ERD and entity chart, let's think through our access patterns. I came up with four:

- Create Session
- Get Session

- Delete Session (time-based)

- Delete Sessions for User (manual revocation)

In this example, I've included all write-based access patterns, such as Create Session and Delete Session. This is mostly to get a feel for how DynamoDB works. In future examples, I won't include write-based access patterns if they're straight-forward without complications.

With our ERD and access patterns in hand, it's time to start modeling!

18.3. Data modeling walkthrough

Let's start working through our modeling patterns. I always like to start by asking a few questions:

1. Should I use a simple or composite primary key?

2. What *interesting* requirements do I have?

3. Which entity should I start modeling first?

In all but the most simple data models, you'll use a composite primary key. A composite primary key will allow you to group items into an item collection and retrieve multiple items in a single request with the Query operation. That said, our application is pretty simple here, so we should keep the simple primary key as an option.

In terms of unique requirements, there are a few here. First, we have a strong need for our session tokens to be unique across our entire table. We can use a UUID generator to create our session tokens to reduce the likelihood of duplicates, but we need to strongly enforce that uniqueness when writing the token to our

table. It would be a disaster if we gave the same token to multiple users, as each one could impersonate the other.

Additionally, the two deletion patterns are pretty interesting. The time-based deletion pattern could be tricky because it's an asychronous action that won't be triggered by any particular web request from a user. We'll need to think how to handle that. Further, the 'Delete Sessions for User' is interesting because we'll need to group by a user to find all the user's tokens.

Let's decide which entity to start modeling. I always like to start with a 'core' entity and build around that.

In our entity chart and ERD, we have two entities listed: Users and Sessions. However, note that User is not really present in our access patterns. We have a need to delete all tokens for a user, but we're not storing and retrieving anything about the user itself. Rather, the user is just a grouping mechanism for the tokens.

Accordingly, we probably only need to model the Session entity in our application. Because we only have one entity, and because our core access patterns around creating and retrieving require uniquely identifying a session by its session token, we'll actually go with a simple primary key here.

With that in mind, let's start working through our interesting access patterns.

18.3.1. Uniqueness requirement on session token

A session token should uniquely identify the user to which it has been given. Assigning the same token to multiple users would be a huge security problem—one user would be able to take actions on behalf of a different user! To handle this, we want to ensure when creating a token that there is not an existing token with the same

value.

If you want to guarantee uniqueness on a value in DynamoDB, you have to build that value into your primary key. We'll do that here by making the `SessionToken` our primary key.

Below are a few example items in our table:

Primary key	Attributes		
Partition key: SessionToken			
0bc6bdf8-6dac-4212-b11a-81f784297c78	Username	CreatedAt	ExpiresAt
	alexdebrie	2020-02-01 15:55:34	2020-02-08 15:55:34
70617849-37d3-45c4-8bb8-ab0072d30bb9	Username	CreatedAt	ExpiresAt
	justinbieber	2020-02-03 22:05:17	2020-02-10 22:05:17
396fa43d-eaf8-4819-b2c5-4efe82f978ba	Username	CreatedAt	ExpiresAt
	lucyricardo	2020-02-05 09:08:11	2020-02-12 09:08:11
cf9eb31f-56c3-4f17-b4ad-84dfe9a490e1	Username	CreatedAt	ExpiresAt
	alexdebrie	2020-02-06 17:09:41	2020-02-13 17:09:41

One thing to note is that we don't have a generic name like `PK` for our partition key. Because we're only storing a single type of entity in this table, we can use a meaningful name like `SessionToken` for the partition key. Even when we get around to our User-based access pattern, we won't be storing information about the User—that will be in a different part of your application. This data model is solely for authentication.

When inserting items into the table, you'll want to use a condition expression that ensures there is not an existing item with the same token.

```
created_at = datetime.datetime.now()
expires_at = created_at + datetime.timedelta(days=7)

result = dynamodb.put_item(
    TableName='SessionStore',
    Item={
        "SessionToken": { "S": str(uuid.uuidv4()) },
        "Username": { "S": "bountyhunter1" },
        "CreatedAt": { "S": created_at.isoformat() },
        "ExpiresAt": { "S": expires_at.isoformat() }
    },
    ConditionExpression: "attribute_not_exists(SessionToken)"
)
```

Notice the `ConditionExpression` argument that is asserting that there is no item with the same token. If there is a session with that same token, an exception will be thrown and this item won't be written to the table.

18.3.2. Time-based deletion of sessions

Now that we have our basic item structure set out, let's move on to the second interesting challenge.

As part of our application, we want to expire tokens that have been in existence too long. This will aid in security and limit the blast radius if tokens are leaked. For our needs, we'll expire a token one week after it was created.

There are three different ways we could handle expiration of sessions:

1. **Pro-active expiration.** We maintain an index of tokens that expire according to their expiration time. We periodically scan that index and delete tokens that have expired.

   This adds cost and complexity as we're both adding another index and taking on responsibility for a background job to remove items from our table. It's an option to consider but one

294

we'd prefer not to use.

2. **Lazy expiration.** Rather than actively looking for tokens to expire, we could defer until later. Whenever a user passes a token to us, we could fetch it from our database and check its expiration time. If the token is expired, we will delete it from the database and force the user to re-authenticate.

   I prefer this approach to the pro-active expiration approach as it's less work for us to maintain. However, you will end up paying for items that are effectively dead in the case that a user doesn't return after the token is expired. Over time, this will cost you money and clog up your table with useless items.

3. **DynamoDB Time-To-Live (TTL).** The last option is to use the TTL feature from DynamoDB. With TTL, DynamoDB will automatically remove items from your table as they expire. To do this, you specify an attribute on the table that is used as the TTL field. If you include this attribute on an item and the attribute value is an epoch timestamp, DynamoDB will remove that item when the current time has passed the time in the field.

I strongly prefer the TTL option as it's the lowest maintenance option for me. To do this, we'll also need to make sure our Session items have the TTL attribute on them.

Note that we already have an `ExpiresAt` attribute that has the expiration time. However, that attribute is using ISO8601 format rather than an epoch timestamp. I generally prefer using the ISO8601 format for timestamps as they are still sortable but also human-readable, which makes it easier for debugging. Instead of changing `ExpiresAt` to an epoch timestamp format, I'll just duplicate the expiration time by adding a `TTL` attribute that is an epoch timestamp.

Now our table looks as follows:

Primary key	Attributes			
Partition key: SessionToken				
0bc6bdf8-6dac-4212-b11a-81f784297c78	Username	CreatedAt	ExpiresAt	TTL
	alexdebrie	2020-02-01 15:55:34	2020-02-08 15:55:34	1581198934
70617849-37d3-45c4-8bb8-ab0072d30bb9	Username	CreatedAt	ExpiresAt	TTL
	justinbieber	2020-02-03 22:05:17	2020-02-10 22:05:17	1581393917
396fa43d-eaf8-4819-b2c5-4efe82f978ba	Username	CreatedAt	ExpiresAt	TTL
	lucyricardo	2020-02-05 09:08:11	2020-02-12 09:08:11	1581520091
cf9eb31f-56c3-4f17-b4ad-84dfe9a490e1	Username	CreatedAt	ExpiresAt	TTL
	alexdebrie	2020-02-06 17:09:41	2020-02-13 17:09:41	1581635381

There is one more note around TTLs. DynamoDB states that items with TTLs are generally removed within 48 hours of their given expiration time. While I've found TTL deletion to be much faster than that in practice, it does raise the possibility that you will allow an expired token to work after it has expired.

To handle this, we can use a filter expression when retrieving an item from the table. This filter expression should remove any items that have already expired. Some sample code is as follows:

```python
epoch_seconds = int(time.time())
result = dynamodb.query(
    TableName='SessionStore',
    KeyConditionExpression="#token = :token",
    FilterExpression="#ttl <= :epoch",
    ExpressionAttributeNames={
        "#token": "SessionToken",
        "#ttl": "TTL"
    },
    ExpressionAttributeValues={
        ":token": { "S": "0bc6bdf8-6dac-4212-b11a-81f784297c78" },
        ":epoch": { "N": str(epoch_seconds) }
    }
)
```

In this `Query` operation, we use the `FilterExpression` argument to remove any items whose TTL is earlier than the current time.

Recall that when learning about filter expressions in Chapter 6, I mentioned that stronger TTL validation is a prime use case for

using filter expressions.

18.3.3. Manual deletion of a user's tokens

The final access pattern we need to support is manual deletion of a user's tokens. A user may want to do this if they're worried they've been hacked, as they can void all tokens that currently exist for their account.

We have a few issues to consider here. The first is that it's possible there are multiple tokens for a single user, such as if they've logged in on both a cell phone and a laptop. However, there's no "DELETE WHERE" syntax for DynamoDB. When deleting items, you need to provide the entire primary key of the items you want to delete.

The second problem is that we can't identify items to delete based on the primary key of the table. The username isn't built into the primary key, and our application has no way to generate the session tokens for a given user.

To handle this, we'll need to provide a way to lookup session tokens by a username. Once we have that, we can make follow-up requests to delete tokens for the user.

First, we'll add a global secondary index on our table. In our secondary index, we'll use a simple key schema with just `Username` as the partition key. The only other attribute we'll need is the `SessionToken`, so we'll use a projection of `KEYS_ONLY` which will only copy the primary key from the main table into our index.

Our secondary index would look as follows:

Primary key	Attributes
Partition key: Username	
alexdebrie	SessionToken
	0bc6bdf8-6dac-4212-b11a-81f784297c78
	SessionToken
	cf9eb31f-56c3-4f17-b4ad-84dfe9a490e1
justinbieber	SessionToken
	70617849-37d3-45c4-8bb8-ab0072d30bb9
lucyricardo	SessionToken
	396fa43d-eaf8-4819-b2c5-4efe82f978ba

Notice that the two tokens for the user `alexdebrie` are in the same partition, and the only attribute that was copied over was the `SessionToken`.

Now if a user wants to delete their tokens, we can use code similar to the following:

```
results = dynamodb.query(
    TableName='SessionStore',
    Index='UserIndex',
    KeyConditionExpression="#username = :username",
    ExpressionAttributeNames={
        "#username": "Username"
    },
    ExpressionAttributeValues={
        ":username": { "S": "alexdebrie" }
    }
)

for result in results['Items']:
    dynamodb.delete_item(
        TableName='SessionStore'
        Key={
            'SessionToken': result['SessionToken']
        }
    )
```

This script runs a Query operation against our secondary index. For

298

each session it finds for the requested user, it runs a DeleteItem operation to remove it from the table. This handles deletion of all tokens for a single user.

18.4. Conclusion

This was a pretty straight-forward example just to get us started. We had just one entity to model, but we did use some interesting features of DynamoDB such as secondary indexes, filter expressions, and TTLs.

Let's review our final solution.

Table Structure

Our table uses a simple primary key with a partition key of `SessionToken`. It has a single global secondary index called `UserIndex` whose key schema is a simple primary key with a partition key of `Username`.

The table also has a TTL property set on the attribute named `TTL`.

Normally I show an entity chart in the summary, but that's not necessary here since we only have one item type.

For our primary keys, we were able to use meaningful names like `SessionToken` and `Username` rather than generic ones like `PK` and `SK`.

Access Patterns

We have the following four access patterns that we're solving:

Access Pattern	Index	Parameters	Notes
Create Session	Main table	• SessionToken	Add condition expression to ensure no duplicates.
Get Session	Main table	• SessionToken	Add filter expression to ensure no expired tokens.
Delete Session (time-based)	N/A	N/A	Expiration handled by DynamoDB TTL
Delete Session (manual)	UserIndex	• Username	1. Find tokens for user.
	Main Table	• SessionToken	2. Delete each token returned from step 1.

Table 15. Session store access patterns

Chapter 19. Building an e-commerce application

With our first example out of the way, let's work on something a little more complex. In this example, we're going to model the ordering system for an e-commerce application. We'll still have the training wheels on, but we'll start to look at important DynamoDB concepts like primary key overloading and handling relationships between entities.

Let's get started.

19.1. Introduction

In this example, you're working on part of an e-commerce store. For this part of our application, we're mostly focused on two core areas: customer management and order management. A different service in our application will handle things like inventory, pricing, and cart management.

I'm going to use screens from ThriftBooks.com to act as our example application. I love using real-life examples as it means I don't have to do any UI design. Fortunately, I have some children who are voracious readers and a wife that is trying to keep up with their needs. This means we have some great example data. Lucky you!

We want to handle the following screens. First, the Account Overview page:

Notice that this page includes information about the user, such as the user's email address, as well as a paginated list of the most recent orders from the user. The information about a particular order is limited on this screen. It includes the order date, order number, order status, number of items, and total cost, but it doesn't include information about the individual items themselves. Also notice that there's something mutable about orders—the order status—meaning that we'll need to change things about the order over time.

To get more information about an order, you need to click on an order to get to the Order Detail page, shown below.

The Order Detail page shows all information about the order, including the summary information we already saw but also additional information about each of the items in the order and the payment and shipping information for the order.

Finally, the Account section also allows customers to save addresses. The Addresses page looks as follows:

A customer can save multiple addresses for use later on. Each address has a name ("Home", "Parents' House") to identify it.

Let's add a final requirement straight from our Product Manager. While a customer is identified by their username, a customer is also required to give an email address when signing up for an account. We want *both* of these to be unique. There cannot be two customers with the same username, and you cannot sign up for two accounts with the same email address.

With these needs in mind, let's build our ERD and list out our access patterns.

19.2. ERD and Access Patterns

As always, the first step to data modeling is to create our entity-relationship diagram. The ERD for this use case is below:
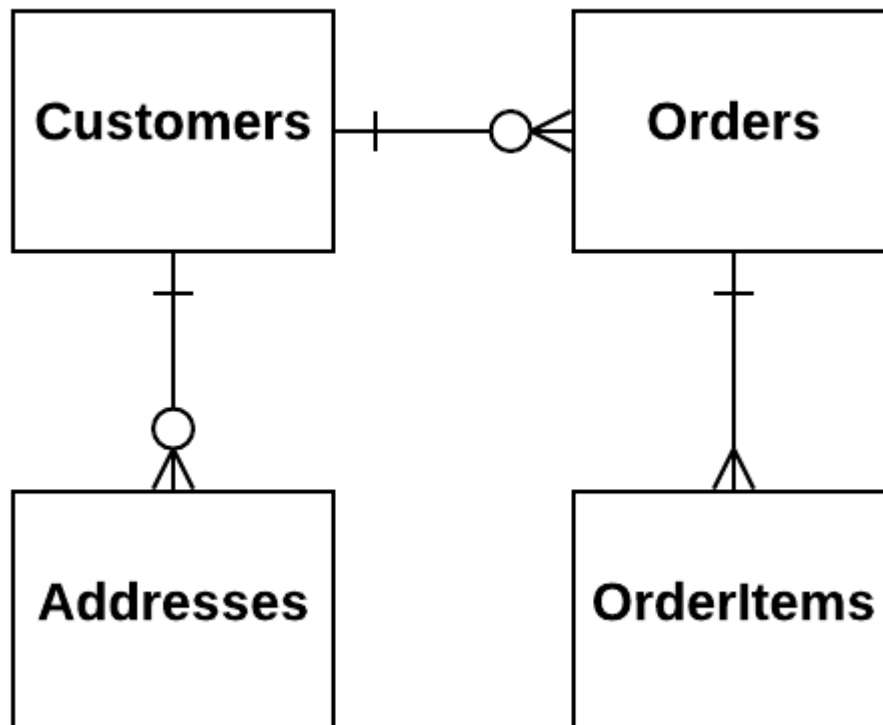
Our application has four entities with three relationships. First,

there are Customers, as identified in the top lefthand corner. A Customer may have multiple Addresses, so there is a one-to-many relationship between Customers and Addresses as shown on the lefthand side.

Moving across the top, a Customer may place multiple Orders over time (indeed, our financial success depends on it!), so there is a one-to-many relationship between Customers and Orders. Finally, an Order can contain multiple OrderItems, as a customer may purchase multiple books in a single order. Thus, there is a one-to-many relationship between Orders and OrderItems.

Now that we have our ERD, let's create our entity chart and list our access patterns.

The entity chart looks like this:

Entity	PK	SK
Customers		
Addresses		
Orders		
OrderItems		

Table 16. E-commerce entity chart

And our acccess patterns are the following:

- Create Customer (unique on both username and email address)
- Create / Update / Delete Mailing Address for Customer
- Place Order
- Update Order
- View Customer & Most Recent Orders for Customer
- View Order & Order Items

With these access patterns in mind, let's start our data modeling.

19.3. Data modeling walkthrough

As I start working on a data modeling, I always think about the same three questions:

1. Should I use a simple or composite primary key?
2. What *interesting* requirements do I have?
3. Which entity should I start modeling first?

In this example, we're beyond a simple model that just has one or two entities. This points toward using a composite primary key. Further, we have a few 'fetch many' access patterns, which strongly points toward a composite primary key. We'll go with that to start.

In terms of interesting requirements, there are two that I noticed:

1. The Customer item needs to be unique on two dimensions: username *and* email address.
2. We have a few patterns of "Fetch parent and all related items" (e.g. Fetch Customer and Orders for Customer). This indicates we'll need to "pre-join" our data by locating the parent item in the same item collection as the related items.

In choosing which entity to start with, I always like to start with a 'core' entity in the application and then work outward as we model it out. In this application, we have two entities that are pretty central: Customers and Orders.

I'm going to start with Customers for two reasons:

1. Customers have a few uniqueness requirements, which generally require modeling in the primary key.
2. Customers are the parent entity for Orders. I usually prefer to start with parent entities in the primary key.

With that in mind, let's model out our Customer items.

19.3.1. Modeling the Customer entity

Starting with the Customer item, let's think about our needs around the Customer.

First, we know that there are two one-to-many relationships with Customers: Addresses and Orders. Given this, it's likely we'll be making an item collection in the primary key that handles at least one of those relationships.

Second, we have two uniqueness requirements for Customers: username and email address. The username is used for actual customer lookups, whereas the email address is solely a requirement around uniqueness.

We're going to focus on handling the uniquness requirements first because that must be built into the primary key of the main table. You can't handle this via a secondary index.

We discussed uniqueness on two attributes in Chapter 16. You can't build uniqueness on multiple attributes into a single item, as that would only ensure the *combination* of the attributes is unique. Rather, we'll need to make multiple items.

Let's create two types of items—Customers and CustomerEmails— with the following primary key patterns:

Customer:

- **PK:** `CUSTOMER#<Username>`
- **SK:** `CUSTOMER#<Username>`

CustomerEmail:

- **PK:** `CUSTOMEREMAIL#<Email>`
- **SK:** `CUSTOMEREMAIL#<Email>`

We can load our table with some items that look like the following:

Primary key		Attributes		
Partition key: PK	**Sort key: SK**			
CUSTOMER#alexdebrie	CUSTOMER#alexdebrie	Username	Email address	Name
		alexdebrie	alexdebrie1@gmail.com	Alex DeBrie
CUSTOMER#the_don	CUSTOMER#the_don	Username	Email address	Name
		the_don	vito@corleone.com	Vito Corleone
CUSTOMEREMAIL#alexdebrie1@gmail.com	CUSTOMEREMAIL#alexdebrie1@gmail.com	Username	Email address	
		alexdebrie	alexdebrie1@gmail.com	
CUSTOMEREMAIL#vito@corleone.com	CUSTOMEREMAIL#vito@corleone.com	Username	Email address	
		the_don	vito@corleone.com	

So far, our service has two customers: Alex DeBrie and Vito Corleone. For each customer, there are two items in DynamoDB. One item tracks the customer by username and includes all information about the customer. The other item tracks the customer by email address and includes just a few attributes to identify to whom the email belongs.

While this table shows the CustomerEmail items, I will hide them when showing the table in subsequent views. They're not critical to the rest of the table design, so hiding them will de-clutter the table.

We can update our entity chart to add the CustomerEmails item type and to fill out the primary key patterns for our first two items:

Entity	PK	SK
Customers	`CUSTOMER#<Username>`	`CUSTOMER#<Username>`
CustomerEmails	`CUSTOMEREMAIL#<Email>`	`CUSTOMEREMAIL#<Email>`
Addresses		
Orders		
OrderItems		

Table 17. E-commerce entity chart

Finally, when creating a new customer, we'll want to only create the customer if there is not an existing customer with the same username and if this email address has not been used for another customer. We can handle that using a DynamoDB Transaction.

The code below shows the code to create a customer with proper validation:

```
response = client.transact_write_items(
    TransactItems=[
        {
            'Put': {
                'TableName': 'EcommerceTable',
                'Item': {
                    'PK': { 'S': 'CUSTOMER#alexdebrie' },
                    'SK': { 'S': 'CUSTOMER#alexdebrie' },
                    'Username': { 'S': 'alexdebrie' },
                    'Name': { 'S': 'Alex DeBrie' },
                    ... other attributes ...
                },
                'ConditionExpression': 'attribute_not_exists(PK)
            }
        },
        {
            'Put': {
                'TableName': 'EcommerceTable',
                'Item': {
                    'PK': { 'S': 'CUSTOMEREMAIL#alexdebrie1@gmail.com' },
                    'SK': { 'S': 'CUSTOMEREMAIL#alexdebrie1@gmail.com' },
                },
                'ConditionExpression': 'attribute_not_exists(PK)
            }
        }
    ]
)
```

Our TransactWriteItems API has two write requests: one to write the Customer item and one to write the CustomerEmail item. Notice that both have condition expressions to confirm that there is not an existing item with the same PK. If one of the conditions is violated, it means that either the username or email address is already in use and thus the entire transaction will be cancelled.

Now that we've handled our Customer item, let's move on to one of

our relationships. I'll go with Addresses next.

19.3.2. Modeling the Addresses entity

There is a one-to-many relationship between Customers and Addresses. We can use strategies from Chapter 11 to see how we can handle the relationship.

The first thing we should ask is whether we can denormalize the relationship. When denormalizing by using a complex attribute, we need to ask two things:

1. Do we have any access patterns that fetch related entity directly by values of the related entity, outside the context of the parent?
2. Is the amount of data in the complex attribute unbounded?

In this case, the answer to the first question is 'No'. We will show customers their saved addresses, but it's always in the context of the customer's account, whether on the Addresses page or the Order Checkout page. We don't have an access pattern like "Fetch Customer by Address".

The answer to the second question is (or can be) 'No' as well. While we may not have considered this limitation upfront, it won't be a burden on our customers to limit them to only 20 addresses. Notice that data modeling can be a bit of a dance. You may not have thought to limit the number of saved addresses during the initial requirements design, but it's easy to add on to make the data modeling easier.

Because both answers were 'No', we can use the denormalization strategy and use a complex attribute. Let's store each customer's addresses on the Customer item.

Our updated table looks as follows:

Notice that our Customer items from before have an `Addresses` attribute outlined in red. The `Addresses` attribute is of the map type and includes one or more named addresses for the customer.

We can update our entity chart as follows:

Entity	PK	SK
Customers	`CUSTOMER#<Username>`	`CUSTOMER#<Username>`
CustomerEmails	`CUSTOMEREMAIL#<Email>`	`CUSTOMEREMAIL#<Email>`
Addresses	N/A	N/A
Orders		
OrderItems		

Table 18. E-commerce entity chart

There is no separate Address item type, so we don't have a PK or SK pattern for that entity.

19.3.3. Modeling Orders

Now let's move on to the Order item. This is our second one-to-many relationship with the Customer item.

Let's walk through the same analysis as we did with Addresses—can we handle this relationship through denormalization?

Unfortunately, it doesn't appear to be a good idea here. Because DynamoDB item sizes are limited to 400KB, you can only

denormalize and store as a complex attribute if there is a limit to the number of related items. However, we don't want to limit the number of orders that a customer can make with us—we would be leaving money on the table! Because of that, we'll have to find a different strategy.

Notice that we have a join-like access pattern where we need to fetch both the Customer and the Orders in a single request. The next strategy, and the most common one for one-to-many relationships, is to use the primary key plus the Query API to 'pre-join' our data.

The Query API can only fetch items with the same partition key, so we need to make sure our Order items have the same partition key as the Customer items. Further, we want to retrieve our Orders by the time they were placed, starting with the most recent.

Let's use the following pattern for our Order items:

- **PK:** `CUSTOMER#<Username>`
- **SK:** `#ORDER#<OrderId>`

For the OrderId, we'll used a KSUID. KSUIDs are unique identifiers that include a timestamp in the beginning. This allows for chronological ordering as well as uniqueness. You can read more about KSUIDs in Chapter 14.

We can add a few Order items to our table to get the following:

Primary key		Attributes				
Partition key: PK	Sort key: SK					
CUSTOMER#alexdebrie	#ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY	OrderId	CreatedAt	Status	Amount	NumberItems
		1VrgXBQ0VCshuQUnh1HrDIHQNwY	2020-01-03 01:57:44	SHIPPED	67.43	7
	#ORDER#1VwVAvJk1GvBFfpTAjm0KG7Cg9d	OrderId	CreatedAt	Status	Amount	NumberItems
		1VwVAvJk1GvBFfpTAjm0KG7Cg9d	2020-01-04 18:53:24	CANCELLED	12.43	2
	CUSTOMER#alexdebrie	Username	Email address	Name		
		alexdebrie	alexdebrie1@gmail.com	Alex DeBrie		
CUSTOMER#the_don	#ORDER#1W1hwN4ywvTR6xzwhU8EHbXULBa	OrderId	CreatedAt	Status	Amount	NumberItems
		1W1hwN4ywvTR6xzwhU8EHbXULBa	2020-01-06 15:07:25	DELIVERED	98.54	4
	CUSTOMER#the_don	Username	Email address	Name		
		the_don	vito@corleone.com	Vito Corleone		

We've added three Order items to our table, two for Alex DeBrie and one for Vito Corleone. Notice that the Orders have the same partition key and are thus in the same item collection as the Customer. This means we can fetch both the Customer and the most recent Orders in a single request.

An additional note—see that we added a prefix of # to our Order items. Because we want the most recent Orders, we will be fetching our Orders in *descending* order. This means our Customer item needs to be *after* all the Order items so that we can fetch the Customer item plus the end of the Order items. If we didn't have the # prefix for Order items, then Orders would show up after the Customer and would mess up our ordering.

To handle our pattern to retrieve the Customer and the most recent Orders, we can write the following Query:

```python
resp = client.query(
    TableName='EcommerceTable',
    KeyConditionExpression='#pk = :pk',
    ExpressionAttributeNames={
        '#pk': 'PK'
    },
    ExpressionAttributeValues={
        ':pk': { 'S': 'CUSTOMER#alexdebrie' }
    },
    ScanIndexForward=False,
    Limit=11
)
```

We use a key expression that uses the proper PK to find the item collection we want. Then we set `ScanIndexForward=False` so that it will start at the *end* of our item collection and go in descending order, which will return the Customer and the most recent Orders. Finally, we set a limit of 11 so that we get the Customer item plus the ten most recent orders.

We can update our entity chart as follows:

Entity	PK	SK
Customers	`CUSTOMER#<Username>`	`CUSTOMER#<Username>`
CustomerEmails	`CUSTOMEREMAIL#<Email>`	`CUSTOMEREMAIL#<Email>`
Addresses	N/A	N/A
Orders	`CUSTOMER#<Username>`	`#ORDER#<OrderId>`
OrderItems		

Table 19. E-commerce entity chart

19.3.4. Modeling the Order Items

The final entity we need to handle is the OrderItem. An OrderItem refers to one of the items that was in an order, such as a specific book or t-shirt.

There is a one-to-many relationship between Orders and OrderItems, and we have an access pattern where we want join-like functionality as we want to fetch both the Order and all its OrderItems for the OrderDetails page.

Like in the last pattern, we can't denormalize these onto the Order item as the number of items in an order is unbounded. We don't want to limit the number of items a customer can include in an order.

Further, we can't use the same strategy of a primary key plus Query API to handle this one-to-many relationship. If we did that, our

OrderItems would be placed between Orders in the base table's item collections. This would significantly reduce the efficiency of the "Fetch Customer and Most Recent Orders" access pattern we handled in the last section as we would now be pulling back a ton of extraneous OrderItems with our request.

That said, the principles we used in the last section are still valid. We'll just handle it in a secondary index.

First, let's create the OrderItem entity in our base table. We'll use the following pattern for OrderItems:

- **PK**: `ORDER#<OrderId>#ITEM#<ItemId>`
- **SK**: `ORDER#<OrderId>#ITEM#<ItemId>`

Our table will look like this:

Primary key		Attributes					
Partition key: PK	Sort key: SK						
CUSTOMER#alexdebrie	#ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY	OrderId	CreatedAt	Status	Amount	NumberItems	
		1VrgXBQ0VCshuQUnh1HrDIHQNwY	2020-01-03 01:57:44	SHIPPED	67.43	7	
	#ORDER#1VwVAvJk1GvBFfpTAjm0KG7Cg9d	OrderId	CreatedAt	Status	Amount	NumberItems	
		1VwVAvJk1GvBFfpTAjm0KG7Cg9d	2020-01-04 18:53:24	CANCELLED	12.43	2	
	CUSTOMER#alexdebrie	Username	Email address	Name			
		alexdebrie	alexdebrie1@gmail.com	Alex DeBrie			
CUSTOMER#the_don	#ORDER#1W1hwN4ywvTR6xzwhU8EHbXULBa	OrderId	CreatedAt	Status	Amount	NumberItems	
		1W1hwN4ywvTR6xzwhU8EHbXULBa	2020-01-06 15:07:25	DELIVERED	98.54	4	
	CUSTOMER#the_don	Username	Email address	Name			
		the_don	vito@corleone.com	Vito Corleone			
ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY#ITEM#48d7	ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY#ITEM#48d7	OrderId	ItemId	Description	Price		
		88da49e72b80	48d7	Go, Dog, Go!	9.72		
ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY#ITEM#be43	ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY#ITEM#be43	OrderId	ItemId	Description	Price		
		88da49e72b80	be43	Les Miserables	14.64		

We have added two OrderItems into our table. They are outlined in red at the bottom. Notice that the OrderItems have the same `OrderId` as an Order in our table but that the Order and OrderItems are in different item collections.

To get them in the same item collection, we'll add some additional properties to both Order and OrderItems.

The GSI1 structure for Orders will be as follows:

- **GSI1PK:** `ORDER#<OrderId>`
- **GSI1SK:** `ORDER#<OrderId>`

The GSI1 structure for OrderItems will be as follows:

- **GSI1PK:** `ORDER#<OrderId>`
- **GSI1SK:** `ITEM#<ItemId>`

Now our base table looks as follows:

Notice that our Order and OrderItems items have been decorated with the `GSI1PK` and `GSI1SK` attributes.

We can then look at our `GSI1` secondary index:

Primary key		Attributes						
Partition key: GSI1PK	Sort key: GSI1SK	PK	SK	OrderId	ItemId	Description	Price	
ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY	ITEM#48d7	ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY#ITEM#48d7	ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY#ITEM#48d7	88da49e72b80	48d7	Go, Dog, Go!	9.72	
		PK	SK	OrderId	ItemId	Description	Price	
	ITEM#be43	ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY#ITEM#be43	ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY#ITEM#be43	88da49e72b80	be43	Les Miserables	14.64	
		PK	SK	OrderId	CreatedAt	Status	Amount	NumberItems
	ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY	CUSTOMER#alexdebrie	#ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY	1VrgXBQ0VCshuQUnh1HrDIHQNwY	2020-01-03 01:57:44	SHIPPED	67.43	7
ORDER#1VwVAvJk1GvBFfpTAjm0KG7Cg9d	ORDER#1VwVAvJk1GvBFfpTAjm0KG7Cg9d	PK	SK	OrderId	CreatedAt	Status	Amount	NumberItems
		CUSTOMER#alexdebrie	#ORDER#1VwVAvJk1GvBFfpTAjm0KG7Cg9d	1VwVAvJk1GvBFfpTAjm0KG7Cg9d	2020-01-04 18:53:24	CANCELLED	12.43	2
ORDER#1W1hwN4ywvTR6xzwhU8EHbXULBa	ORDER#1W1hwN4ywvTR6xzwhU8EHbXULBa	PK	SK	OrderId	CreatedAt	Status	Amount	NumberItems
		CUSTOMER#the_don	#ORDER#1W1hwN4ywvTR6xzwhU8EHbXULBa	1W1hwN4ywvTR6xzwhU8EHbXULBa	2020-01-06 15:07:25	DELIVERED	98.54	4

Now our Orders and OrderItems have been re-arranged so they are in the same item collection. As such, we can fetch an Order and all of its OrderItems by using the Query API against our secondary index.

The code to fetch an Order and all of its OrderItems is as follows:

```
resp = client.query(
    TableName='EcommerceTable',
    IndexName='GSI1',
    KeyConditionExpression='#gsi1pk = :gsi1pk',
    ExpressionAttributeNames={
        '#gsi1pk': 'GSI1PK'
    },
    ExpressionAttributeValues={
        ':gsi1pk': 'ORDER#1VrgXBQ0VCshuQUnh1HrDIHQNwY'
    }
)
```

We can also update our entity chart as follows:

Entity	PK	SK
Customers	CUSTOMER#<Username>	CUSTOMER#<Username>
CustomerEmails	CUSTOMEREMAIL#<Email>	CUSTOMEREMAIL#<Email>

Entity	PK	SK
Addresses	N/A	N/A
Orders	`CUSTOMER#<Username>`	`#ORDER#<OrderId>`
OrderItems	`ORDER#<OrderId>#ITEM#` `<ItemId>`	`ORDER#<OrderId>#ITEM#` `<ItemId>`

Table 20. E-commerce entity chart

Further, let's make a corresponding entity chart for `GSI1` so we can track items in that index:

Entity	GSI1PK	GSI1SK
Customers		
CustomerEmails		
Addresses		
Orders	`ORDER#<OrderId>`	`ORDER#<OrderId>`
OrderItems	`ORDER#<OrderId>`	`ITEM#<ItemId>`

Table 21. E-commerce GSI1 entity chart

19.4. Conclusion

We're starting to get warmed up with our DynamoDB table design. In this chapter, we looked at some advanced patterns including using primary key overloading to create item collections with heterogeneous items.

Let's review our final solution.

Table Structure

Our table uses a composite primary key with generic names of `PK` and `SK` for the partition key and sort key, respectively. We also have a global secondary index named `GSI1` with similarly generic names of `GSI1PK` and `GSI1SK` for the partition and sort keys.

Our final entity chart for the main table is as follows:

Entity	PK	SK
Customers	`CUSTOMER#<Username>`	`CUSTOMER#<Username>`
CustomerEmails	`CUSTOMEREMAIL#<Email>`	`CUSTOMEREMAIL#<Email>`
Addresses	N/A	N/A
Orders	`CUSTOMER#<Username>`	`#ORDER#<OrderId>`
OrderItems	`ORDER#<OrderId>#ITEM#` `<ItemId>`	`ORDER#<OrderId>#ITEM#` `<ItemId>`

Table 22. E-commerce entity chart

And the final entity chart for the GSI1 index is as follows:

Entity	GSI1PK	GSI1SK
Customers		
CustomerEmails		
Addresses		
Orders	`ORDER#<OrderId>`	`ORDER#<OrderId>`
OrderItems	`ORDER#<OrderId>`	`ITEM#<ItemId>`

Table 23. E-commerce GSI1 entity chart

Notice a few divergences from our ERD to our entity chart. First, we needed to add a special item type, 'CustomerEmails', that are used solely for tracking the uniqueness of email addresses provided by customers. Second, we don't have a separate item for Addresses as we denormalized it onto the Customer item.

After you make these entity charts, you should include them in the documentation for your repository to assist others in knowing how the table is configured. You don't want to make them dig through your data access layer to figure this stuff out.

Access Patterns

We have the following six access patterns that we're solving:

Access Pattern	Index	Parameters	Notes
Create Customer	N/A	N/A	Use TransactWriteItems to create Customer and CustomerEmail item with conditions to ensure uniqueness on each
Create / Update Address	N/A	N/A	Use UpdateItem to update the `Addresses` attribute on the Customer item
View Customer & Most Recent Orders	Main table	• Username	Use `ScanIndexForward=False` to fetch in descending order.
Save Order	N/A	N/A	Use TransactWriteItems to create Order and OrderItems in one request
Update Order	N/A	N/A	Use UpdateItem to update the status of an Order
View Order & Order Items	GSI1	• OrderId	

Table 24. E-commerce access patterns

Just like your entity charts, this chart with your access pattern should be included in the documentation for your repository so that it's easier to understand what's happening in your application.

In the next chapter, we're going to dial it up to 11 by modeling a complex application with a large number of entities and relationships.

Chapter 20. Building Big Time Deals

The first two examples were warmups with a small number of access patterns. We got our feet wet, but they're likely to be more simple than most real-world applications.

It's time to take off the training wheels.

In this example, we're going to create Big Time Deals, a Deals app that is similar to slickdeals.net or a variety of other similar sites. This application will have a number of relations between entities, and we'll get to see how to model out some complex patterns.

One thing I really like about this example is that it's built for two kinds of users. First, there are the external users—people like you and me that want to visit the site to find the latest deals.

But we also have to consider the needs of the editors of the site. They are interacting with the site via an internal content management system (CMS) to do things like add deals to the site, set featured deals, and send notifications to users. It's fun to think of this problem from both sides to make a pattern that works for both parties.

With that in mind, let's get started!

20.1. Introduction

To begin, let's walk through our site and get a feel for the entities and access patterns we need to support.

First, Big Time Deals is solely accessible via a mobile app. We want to provide a fast, native way for users to view deals.

The front page of Big Time Deals looks as follows:
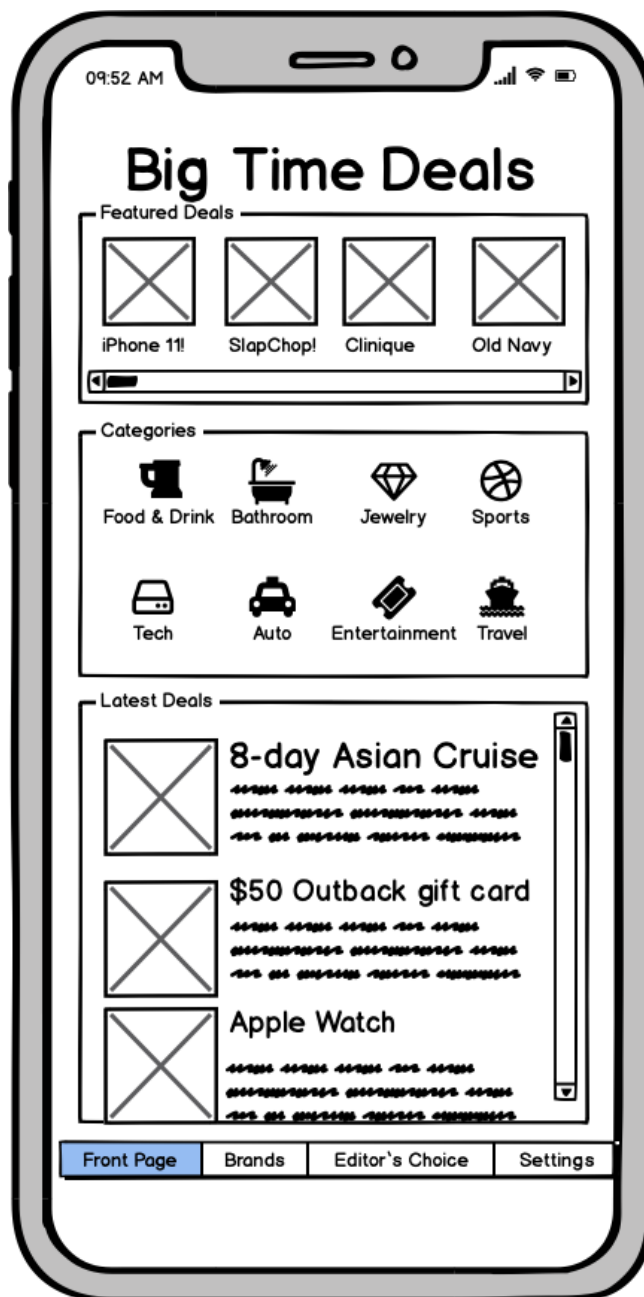
There are three main things to point out here. First, on the top row, we have Featured Deals. These are 5-10 curated deals that have been selected by the Big Time Deals editors to be interesting to our users.

Second, there are eight categories of deals. Each deal belongs to

exactly one category, and you can view all deals in a particular category by clicking the category icon.

Finally, the latest deals are shown in an infinitely scrollable area in the bottom half of the screen. This shows a deal image, the deal title, and some text about the deal.

The bottom menu for the application has four different pages. We've discussed the Front Page, so let's go to the Brands page.

The Brands page looks as follows:
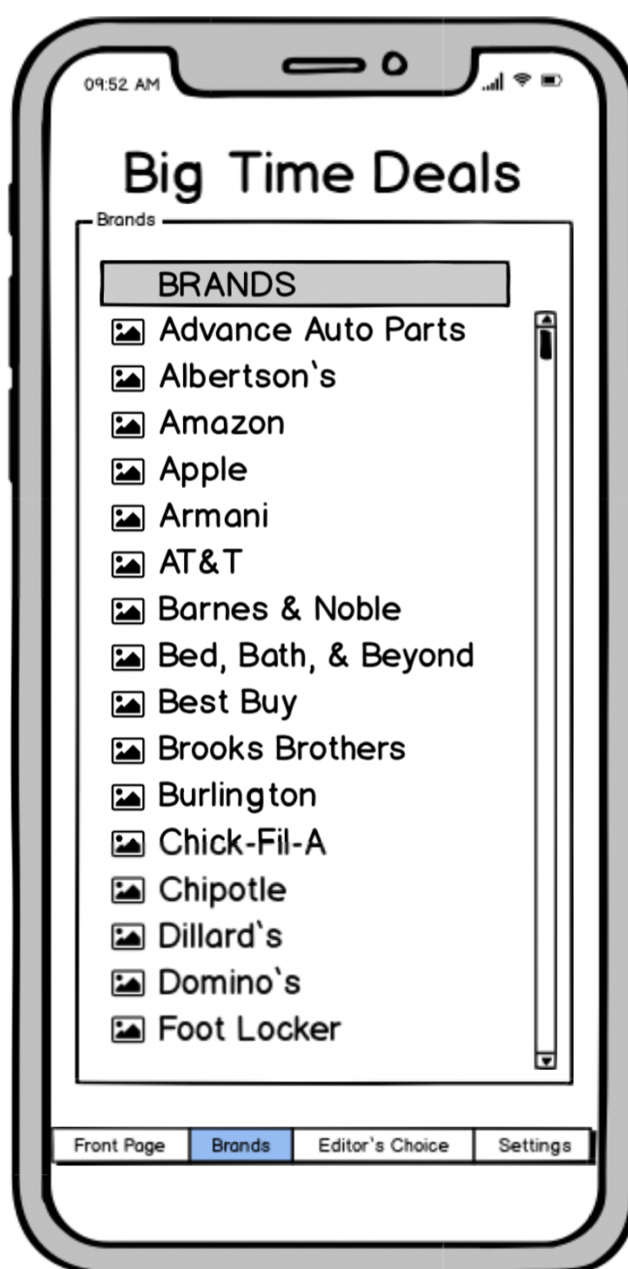
The Brands page is very simple. It includes an alphabetical list of all brands and an icon for each brand. There aren't that many brands in our system—only about 200 currently—and we expect to add maybe 1-2 brands per month.

If you click on a brand from the Brands page, you will be directed to the page for that brand. The Brand page looks like the following:
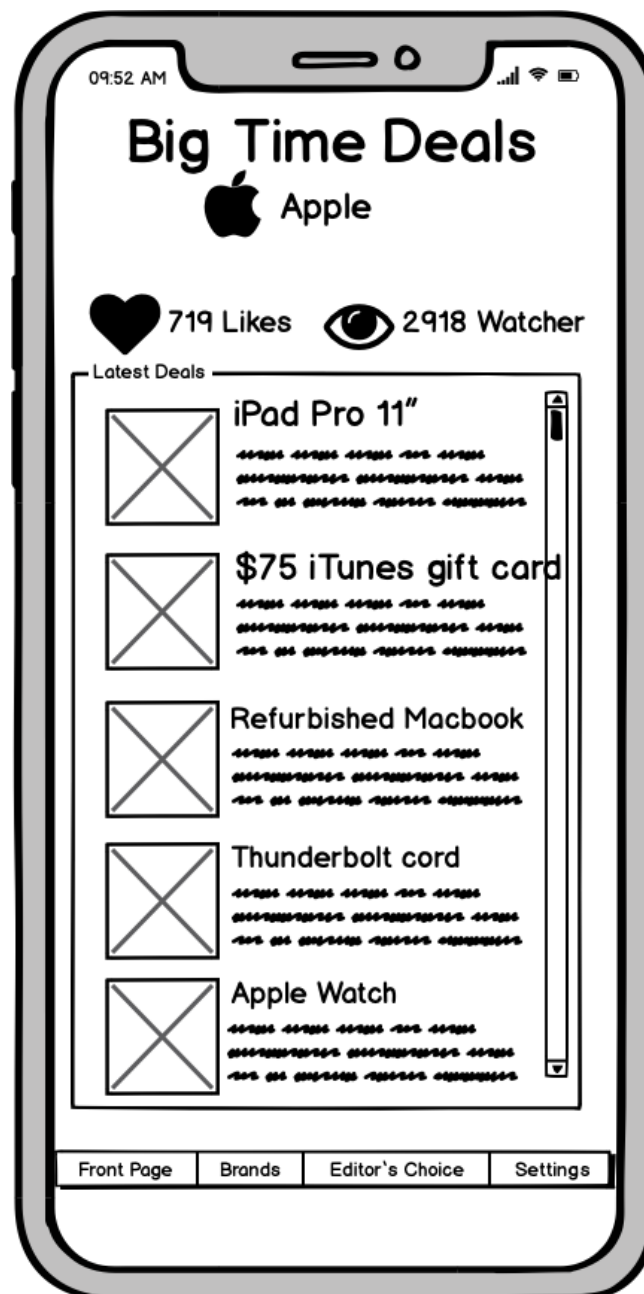
The Brand page includes the name of the brand and the icon. The Brand page also shows the number of likes and watchers for the given brand. A user may 'like' the brand to show support for it, and

a user may 'watch' the brand to receive notifications whenever there is a new deal for the brand.

Finally, the Brand page shows the latest deals for the given brand.

In addition to the Brand page, there is also a Category page for each of the eight categories. The Category page for the Tech category looks as follows:
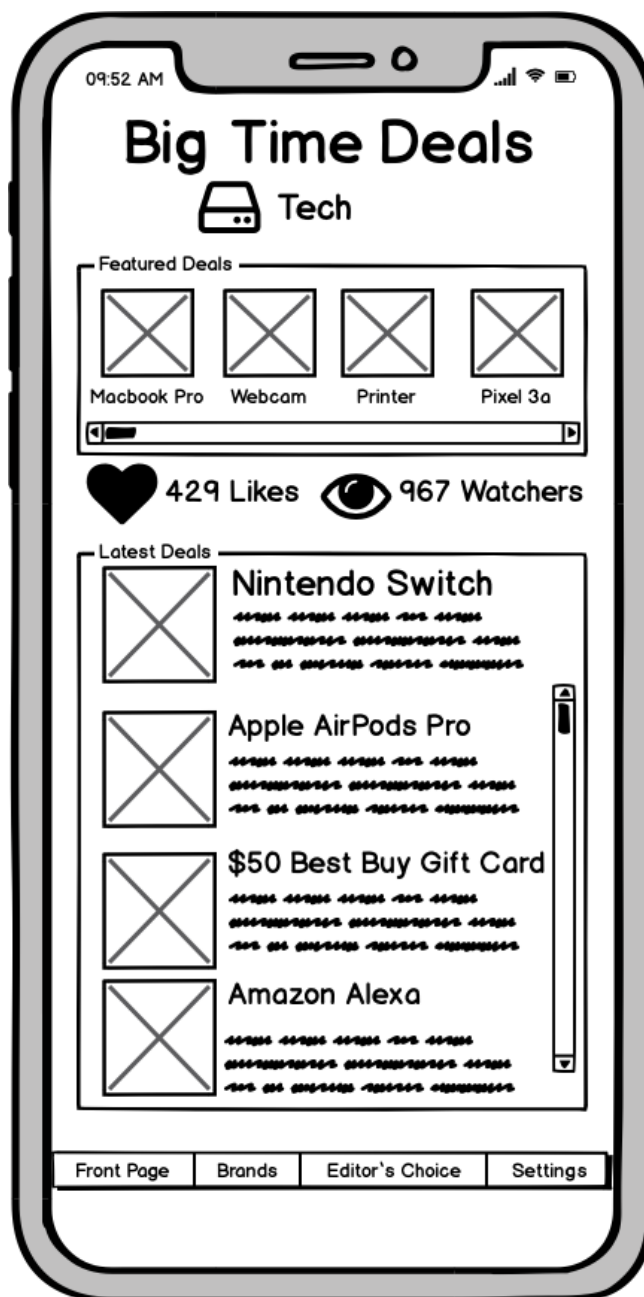
The Category page is very similar to the Brand page with likes, watchers, and latest deals.

The Category page also includes featured deals, which are deals within the category that are curated by our editors. We didn't include featured deals on each Brand page because it's too much work for our editors to handle featured deal curation across 200+ brands, but it's not as big a task to handle it for the eight categories.

If you click on a specific deal, you'll get routed to the following Deal page:

The deal page is pretty simple. It includes a picture of the deal and details about the deal. It also includes the price of the deal. There is

a button to get this deal (which includes an affiliate link to the external retailer) as well as buttons to see additional deals from the same brand or in the same category.

The next page is the Editor's Choice page, which looks as follows:
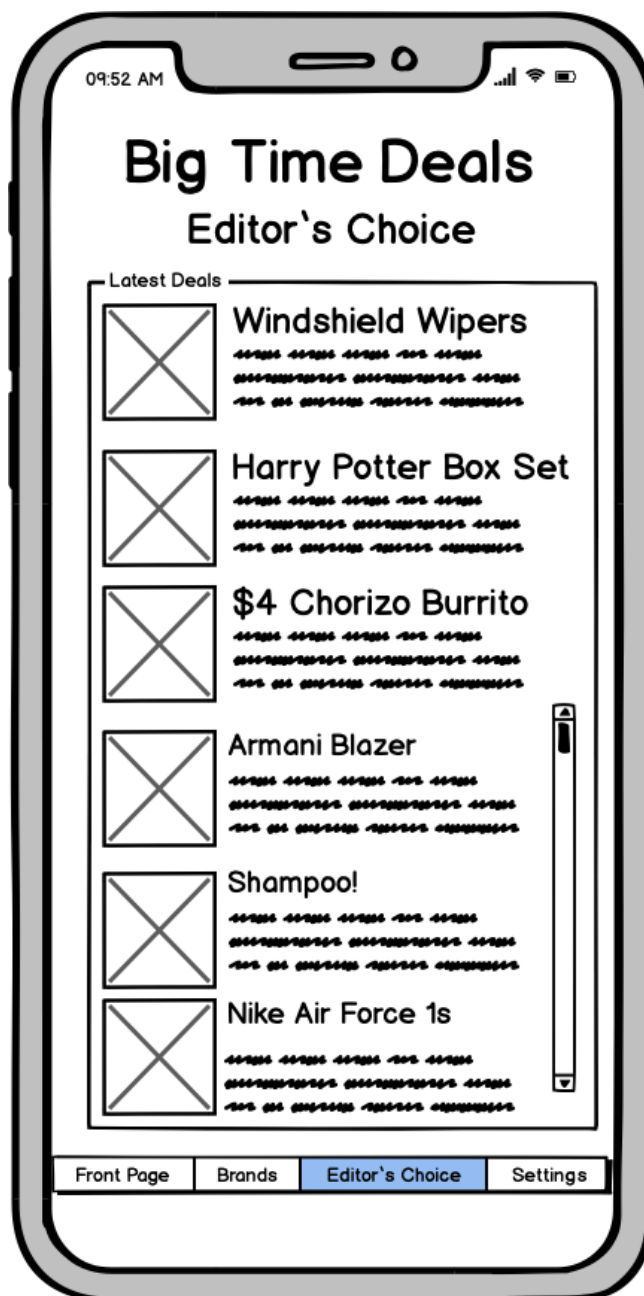
The Editor's Choice page is another curated portion of the site. This includes more unique or eclectic deals that might be missed in other areas of the site. Like the Featured sections, the Editor's Choice page is entirely managed by our editors.

Finally, there is a Settings page:
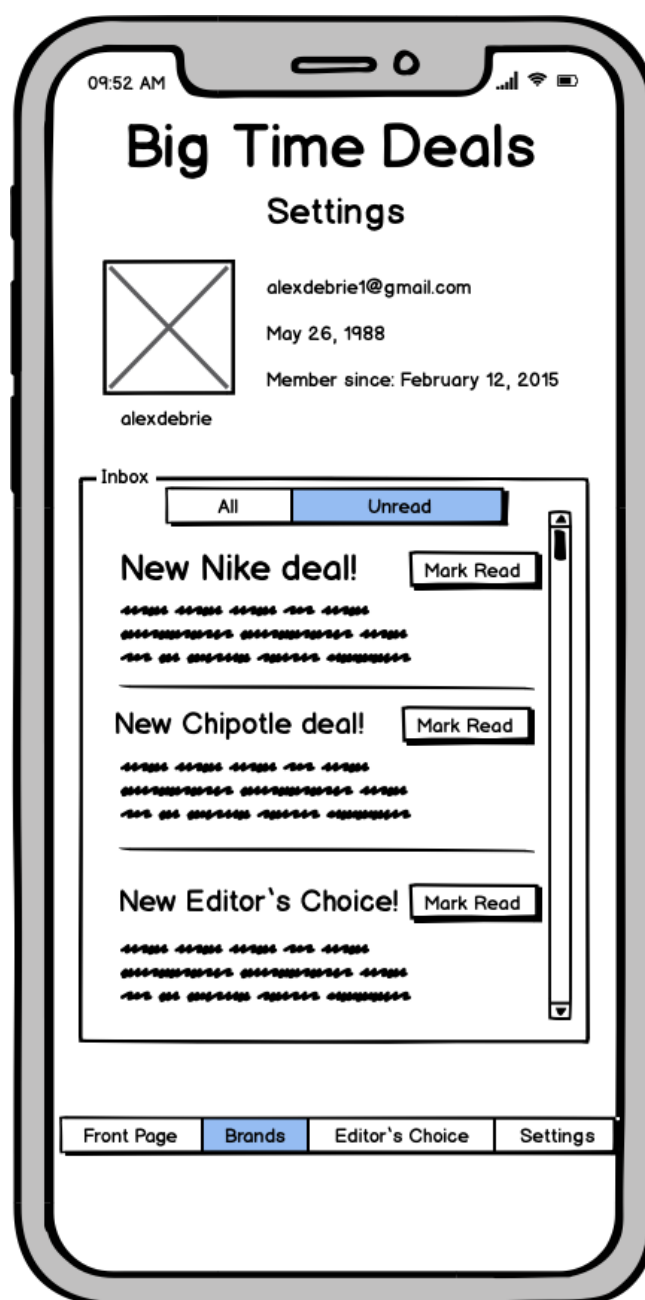
The Settings page has two purposes. First, it includes information about the user in question—a profile picture, an email address, a birthday, and how long they've been a member.

It also includes an Inbox. Users will receive messages in their Inbox in one of two scenarios:

1. A new deal has launched for a brand or category that they're watching

2. The Big Time Deals team has blasted all users with news of a particular deal

Inbox messages are shown in chronological order with the most recent messages first. There are two view modes: show all messages, or show only the unread messages.

Now that we know the entities that we will be modeling, take the time to create your ERD and draw up the access patterns in your application.

20.2. ERD & Access Patterns

As always, we'll start with the ERD. Here is the ERD I created for our Deals application:
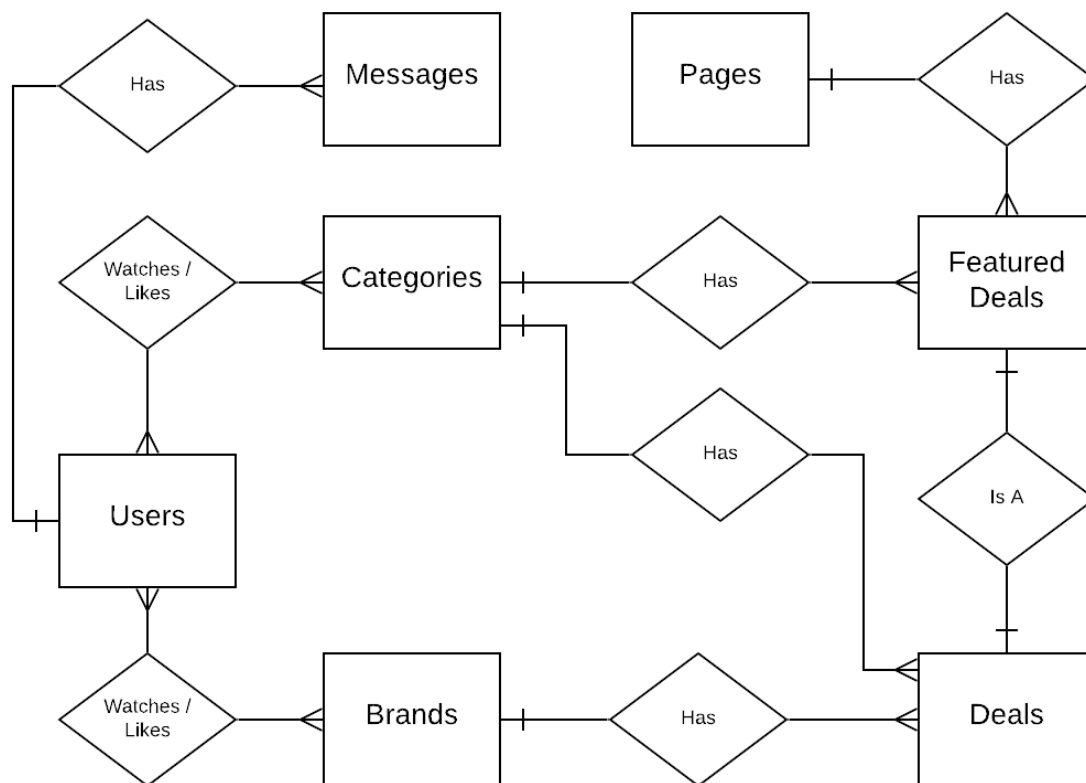
Let's start on the left-hand side. There are Users who will create

accounts in our application. These Users will have multiple Messages in their inbox, but a specific Message will belong to a particular User. Thus, there is a one-to-many relationship between Users and Messages.

Users also have a many-to-many relationship with both Categories and Brands. For both entities, there is both a Likes relationship and a Watches relationship. While these relationships are different, I've collapsed them into a single box for simplicity.

Look now to the bottom right-hand corner of the ERD. There we have Deals, which represent a single deal in our application. Each Deal belongs to a single Brand and Category, so there is a one-to-many relationship between Brands and Deals and a one-to-many relationship between Categories and Deals.

Additionally, some Deals are *Featured Deals*. This means they have been selected by editors to be featured somewhere on the site. This is an odd relationship to model, but I've shown it as a one-to-one relationship between Deals and Featured Deals.

These Featured Deals are shown in different places. First, each Category has a number of Featured Deals, so there is a one-to-many relationship between Categories and Featured Deals.

Additionally, we have two Pages—Front Page and Editor's Choice—that show Featured Deals. There is a one-to-many relationship between a Page and Featured Deals.

Now that we have our ERD, let's create our entity chart and list our access patterns.

The entity chart looks like this:

Entity	PK	SK
Deal		

Entity	PK	SK
Brand		
Category		
FeaturedDeal		
Page		
User		
Message		

Table 25. Big Time Deals entity chart

When we have a large amount of patterns, I like to group them into conceptual categories.

Here are the categories and access patterns:

Create core deal entities:

- Create Deal
- Create Brand
- Create Category

Set Featured Deals:

- for Front Page
- for Category
- for Editor's Choice Page

Fetch page:

- Fetch Featured Deals & Latest Deals for Front Page
- Fetch Featured Deals & Latest Deals for Category
- Fetch Featured Deals for Editor's Choice Page
- Fetch Latest Deals for Brand
- Fetch all Brands

- Fetch Deal

Users and interactions:

- Create User
- Like Brand for User
- Watch Brand for User
- Like Category for User
- Watch Category for User

Messages:

- View Messages for User
- View Unread Messages for User
- Mark Message as Read
- Send Hot New Deal Message to all Users
- Send new Brand Deal Message to Brand Watchers
- Send new Category Deal Message to Category Watchers

This has *twenty-three* access patterns that we're going to model out. Let's do this!

20.3. Data modeling walkthrough

As usual, we're going to start out data modeling by asking our three questions:

1. Should I use a simple or composite primary key?
2. What *interesting* requirements do I have?
3. Which entity should I start modeling first?

If you can model twenty-three access patterns with a simple primary key, my hat's off to you. We're going to go with a composite primary key in this example.

In terms of interesting requirements, we have a few. First, there are a number of *time-based* access patterns. We want to fetch the latest deals for a category, for a brand, and even for the Big Time Deals site as a whole. Time-based access patterns can be tricky when you're doing it in large partitions, as we will be here. We'll need to balance query efficiency with proper partitioning to ensure our partitions don't get too large.

A second interesting requirement is how the Deal entity is viewed in a number of different contexts. It's not just on the Deal page for the particular deal; a Deal is also shown on the Brand page, the Category page, and the Front Page. Further, it could be in the latest deals section or the featured deals section on some of these pages. We'll need to think how to duplicate or project that Deal item into the proper locations in our table.

A third interesting note is one I mentioned at the beginning: we have two sets of users in our application. Some access patterns will be handled from the perspective of external deal-hunters while others will be internal editors for our application.

Finally, we have two access patterns where we need to "Fetch all entities of Type X". We want to retrieve all Brand items to show on the Brands page, and we want to retrieve all Users when we blast everyone with a Hot New Deal message. These access patterns are usually pretty tricky, but we'll find ways to handle them.

In choosing which entity to start with, I prefer to start with a 'core' entity. Usually that means an entity that is the parent entity in a few different one-to-many relationships. We don't really have that here.

We do have the Deals entity which is the *related* entity in a number of one-to-many relationships, but its only relationship where it's the parent is in the one-to-one relationship with Featured Deals. However, because it's pretty important in our application and because it needs to be seen in a number of different contexts, let's start there.

20.3.1. Modeling the Deal entity

Starting with the Deal entity, there are a few groupings of access patterns. First are the simple read & write operations for the Deal item itself. When our editors are adding a new deal or when a user navigates to view a deal, we'll need to be able to find that specific entity, likely by a unique ID.

Second, we have a number of 'fetch latest deals for X' access patterns, where X might be a particular brand or a particular category. We even have a "fetch latest deals overall" access pattern. These "fetch latest" access patterns can sometimes be difficult as they can result in large partitions or very hot partitions.

Let's handle the basic write & read operations first. We'll use the following primary key pattern for our Deal entity:

- **PK:** `DEAL#<DealId>`
- **SK:** `DEAL#<DealId>`

The `DealId` will be a unique identifier generated for each Deal item. For generating our `DealId` attribute, we'll use a KSUID. A KSUID is a K-Sortable Unique IDentifier. It is generated by an algorithm that combines a current timestamp and some randomness. A KSUID is a 27-character string that provides the uniqueness properties of a UUIDv4 while also retaining chronological ordering when using lexicographical sorting.

For more on KSUIDs, check out Chapter 14.

With this pattern in mind, our table with a few Deal items looks as follows:

Primary key		Attributes						
Partition key: PK	Sort key: SK							
DEAL#1WK0JFwUfFNUw9hddaTjEDpizTV	DEAL#1WK0JFwUfFNUw9hddaTjEDpizTV	Title	DealId	Link	Price	Category	Brand	CreatedAt
		8-day Asian Cruise	1WK0JFwUfFNUw9hddaTjEDpizTV	https://cruises.com/8-day	$2,479	Travel	Princess	2020-01-13T02:35:04
DEAL#1X7KZHsuFogN0gLOx4Vvrf7azl8	DEAL#1X7KZHsuFogN0gLOx4Vvrf7azl8	Title	DealId	Link	Price	Category	Brand	CreatedAt
		$50 Outback Gift Card	1X7KZHsuFogN0gLOx4Vvrf7azl8	https://giftcards.com/outback	$45	Food & Drink	Outback	2020-01-30T13:42:26
DEAL#1X7AGV4dycQ31BUOAifKRyz5Q6d	DEAL#1X7AGV4dycQ31BUOAifKRyz5Q6d	Title	DealId	Link	Price	Category	Brand	CreatedAt
		Apple Watch	1X7AGV4dycQ31BUOAifKRyz5Q6d	https://macoutlet.com/watch	$350	Tech	Apple	2020-01-30T12:17:43

This pattern allows fast, individual operations on specific Deal items. Now let's move on to fetching our latest deals.

Fetching the most recent deal items

We have three different patterns of fetching the most recent deals:

1. Fetch most recent deals overall for the Front Page
2. Fetch most recent deals by Brand for the Brand Page
3. Fetch most recent deals by Category for the Category Page

The pattern we'll use for all three will be similar, but we'll focus on the first one to start since we haven't modeled out the Brand and Category items yet.

There are two considerations that I'm thinking about when modeling these. One is about data size, and one is about data speed.

Deals partition data size

Let's talk about data size first. When modeling out a "fetch most recent" pattern, the common approach is to put the relevant items in a single item collection by sharing the partition key, then using the sort key to sort the items by a timestamp or a KSUID that is sortable by timestamp. This is something we saw in the previous chapter with Users & Orders.

However, I'm a little nervous about that pattern in this instance. In the last chapter, we had a good way to segment Orders—by User— that ensured a more even distribution across our table. In this one, we would be placing *all Deal items* in the entire application into a single partition. Over time, this partition will get quite large.

Instead, we'll do some sharding of our partition key to help spread this out. Sharding is when you split your data across multiple partitions to even out a heavy load. There are a number of different read-sharding patterns, including using random partitions with a scatter-gather approach at read time. I don't love that approach here as then we would need to hit all N random partitions and re-assemble the latest items to show users the latest deals.

Instead, we'll do some sharding by the timestamp. Essentially we'll group items into item collections based on a similar time range. This will enable fast reads of deals in order while also splitting deals into different partitions.

To handle this, we'll add two attributes to our Deal items. Those attributes will be as follows:

- **GSI1PK:** `DEALS#<TruncatedTimestamp>`
- **GSI1SK:** `DEAL#<DealId>`

For the `<TruncatedTimestamp>` value, we'll truncate the timestamp down to a time period in which it is. You can truncate down to any granularity you like—hour, day, week, month, year—and your choice will depend on how frequently you're writing items.

For example, if your timestamp was `2020-02-14 12:52:14` and you truncated to the day, your truncated timestamp would be `2020-02-14 00:00:00`. If you truncated to the month, your truncated timestamp would be `2020-02-01 00:00:00`.

In our example, imagine that we'll have roughly 100 new deals per day. We might choose to truncate down to the day as a partition with 100 items should be manageable. Given that, our Deals items will now look as follows:

Partition key: PK	Sort key: SK	Attributes								
DEAL#1WK0JFwUfFNUw9hddaTjEDpizTV	DEAL#1WK0JFwUfFNUw9hddaTjEDpizTV	Title	DealId	Link	Price	Category	Brand	CreatedAt	GSI1PK	GSI1SK
		8-day Asian Cruise	1WK0JFwUfFNUw9hddaTjEDpizTV	https://cruises.com/8-day	$2,479	Travel	Princess	2020-01-13T02:35:04	DEALS#2020-01-13T00:00:00	DEAL#1WK0JFwUfFNUw9hddaTjEDpizTV
DEAL#1X7KZHsuFogN0gLOx4Vvrf7azl8	DEAL#1X7KZHsuFogN0gLOx4Vvrf7azl8	Title	DealId	Link	Price	Category	Brand	CreatedAt	GSI1PK	GSI1SK
		$50 Outback Gift Card	1X7KZHsuFogN0gLOx4Vvrf7azl8	https://giftcards.com/outback	$45	Food & Drink	Outback	2020-01-30T13:42:26	DEALS#2020-01-30T00:00:00	DEAL#1X7KZHsuFogN0gLOx4Vvrf7azl8
DEAL#1X7AGV4dycQ31BUOAifKRyz5Q6d	DEAL#1X7AGV4dycQ31BUOAifKRyz5Q6d	Title	DealId	Link	Price	Category	Brand	CreatedAt	GSI1PK	GSI1SK
		Apple Watch	1X7AGV4dycQ31BUOAifKRyz5Q6d	https://macoutlet.com/watch	$350	Tech	Apple	2020-01-30T12:17:43	DEALS#2020-01-30T00:00:00	DEAL#1X7AGV4dycQ31BUOAifKRyz5Q6d

Notice the two attributes added that are outlined in red. These will be used for our global secondary index.

Let's take a look at that index:

Partition key: GSI1PK	Sort key: GSI1SK	Attributes								
DEALS#2020-01-13T00:00:00	DEAL#1WK0JFwUfFNUw9hddaTjEDpizTV	PK	SK	Title	DealId	Link	Price	Category	Brand	CreatedAt
		DEAL#1WK0JFwUfFNUw9hddaTjEDpizTV	DEAL#1WK0JFwUfFNUw9hddaTjEDpizTV	8-day Asian Cruise	1WK0JFwUfFNUw9hddaTjEDpizTV	https://cruises.com/8-day	$2,479	Travel	Princess	2020-01-13T02:35:04
DEALS#2020-01-30T00:00:00	DEAL#1X7AGV4dycQ31BUOAifKRyz5Q6d	PK	SK	Title	DealId	Link	Price	Category	Brand	CreatedAt
		DEAL#1X7AGV4dycQ31BUOAifKRyz5Q6d	DEAL#1X7AGV4dycQ31BUOAifKRyz5Q6d	Apple Watch	1X7AGV4dycQ31BUOAifKRyz5Q6d	https://macoutlet.com/watch	$350	Tech	Apple	2020-01-30T12:17:43
	DEAL#1X7KZHsuFogN0gLOx4Vvrf7azl8	PK	SK	Title	DealId	Link	Price	Category	Brand	CreatedAt
		DEAL#1X7KZHsuFogN0gLOx4Vvrf7azl8	DEAL#1X7KZHsuFogN0gLOx4Vvrf7azl8	$50 Outback Gift Card	1X7KZHsuFogN0gLOx4Vvrf7azl8	https://giftcards.com/outback	$45	Food & Drink	Outback	2020-01-30T13:42:26

Notice that deals are now grouped together by the day they were created.

Note that this could run into a pagination problem. Imagine our Fetch Latest Deals endpoint promises to return 25 deals in each request. If the user is making this request at the beginning of a day where there aren't many deals or if they've paginated near the end of a day's deals, there might not be 25 items within the given partition.

We can handle that in our application logic. First, you would have a `fetch_items_for_date` function that looks as follows:

```python
def fetch_items_for_date(date, last_seen='$', limit=25):
    resp = client.query(
        TableName='BigTimeDeals',
        Index='GSI1',
        KeyConditionExpression='#pk = :pk AND #sk < :sk',
        ExpressionAttributeNames={
            '#pk': 'PK',
            '#sk': 'SK'
        },
        ExpressionAttributeValues={
            ':pk': { 'S': f"DEALS#{date.strftime('%Y-%m-%dT00:00:00')}" },
            'sk': { 'S': f'DEAL#{last_seen}' }
        },
        ScanIndexForward=False,
        Limit=limit
    )
```

Then, when wanting to fetch some items, you could use the following logic:

```python
def get_deals(date, last_deal_seen='$', count=0):
    deals = []

    while len(deals) < 25 and count < 5:
        items = fetch_items_for_date(date, last_deal_seen, count)
        for item in items:
            deals.append(
                Deal(
                    title=item['Title']['S'],
                    deal_id=item['DealId']['S'],
                    link=item['Link']['S'],
                    ...
                )
            )
        date = date - datetime.timedelta(days=1)
        count += 1

    return deals[:24]
```

This function would be called by your HTTP endpoint handler with date and last deal seen (if any) by the user. It would call our `fetch_items_for_date` function to retrieve some deals until it reached 25 total items.

Notice that we've also included a `count` property to ensure we only look for a maximum of 5 days. If there were fewer than 25 deals in our table, we could keep querying forever, attempting to find deals for earlier and earlier dates. At some point we simply give up and return the number of results we have.

In most cases, we could handle this with a single request, and it would rarely take more than two requests. Plus, we get nicer separation of our deals by date so we don't overload a partition.

Deals partition data speed

Now that we've covered the data size issue for Deals, let's talk about data speed. This is the issue I'm actually more worried about. Because the most recent deals will be viewed by everyone that hits our front page, we'll be reading from that partition quite a bit. We will very likely have a hot key issue where one partition in our database is read significantly more frequently than others.

The good thing is that this access pattern is knowable in advance and thus easily cachable. Whenever an editor in our internal CMS adds a new deal, we can run an operation to fetch the most recent two days worth of deals and cache them.

There are two ways we could do this. If we wanted to use some sort of external cache like Redis, we could store our latest deals there. That architecture might look as follows:
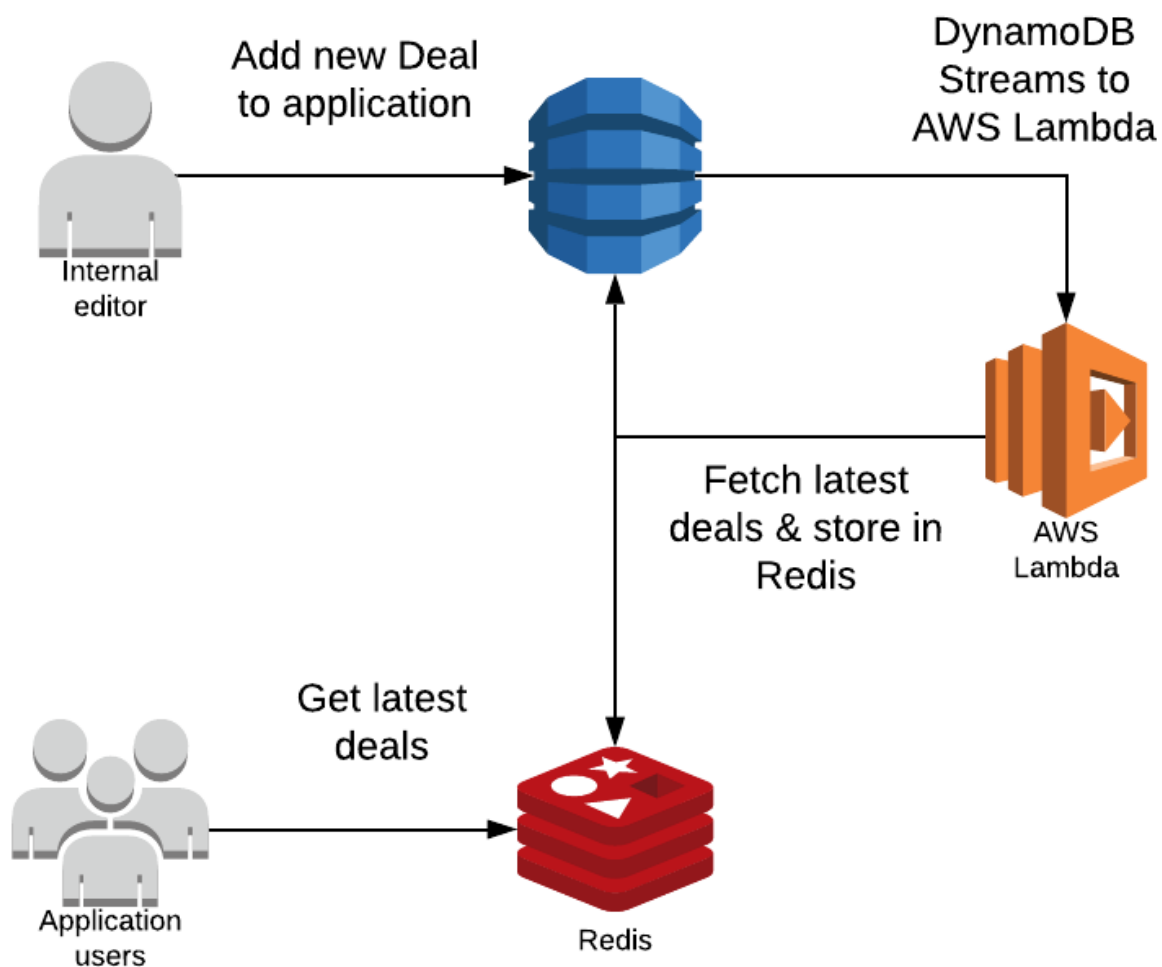
In the top left of the diagram, an internal editor adds a new Deal to our DynamoDB table. This triggers an AWS Lambda function that is processing the DynamoDB Stream for our table. Upon seeing a new Deal has been added, it queries our table to find all Deals for the past two days. Then it stores those Deals in Redis. Later, when

application users ask to view deals, the most recent two days will be read from Redis.

If you didn't want to add an additional system like Redis to the mix, you could even cache this in DynamoDB. After retrieving the last two days worth of deals, you could copy all that data across multiple items that are serving as little caches.

The architecture for this might look as follows:
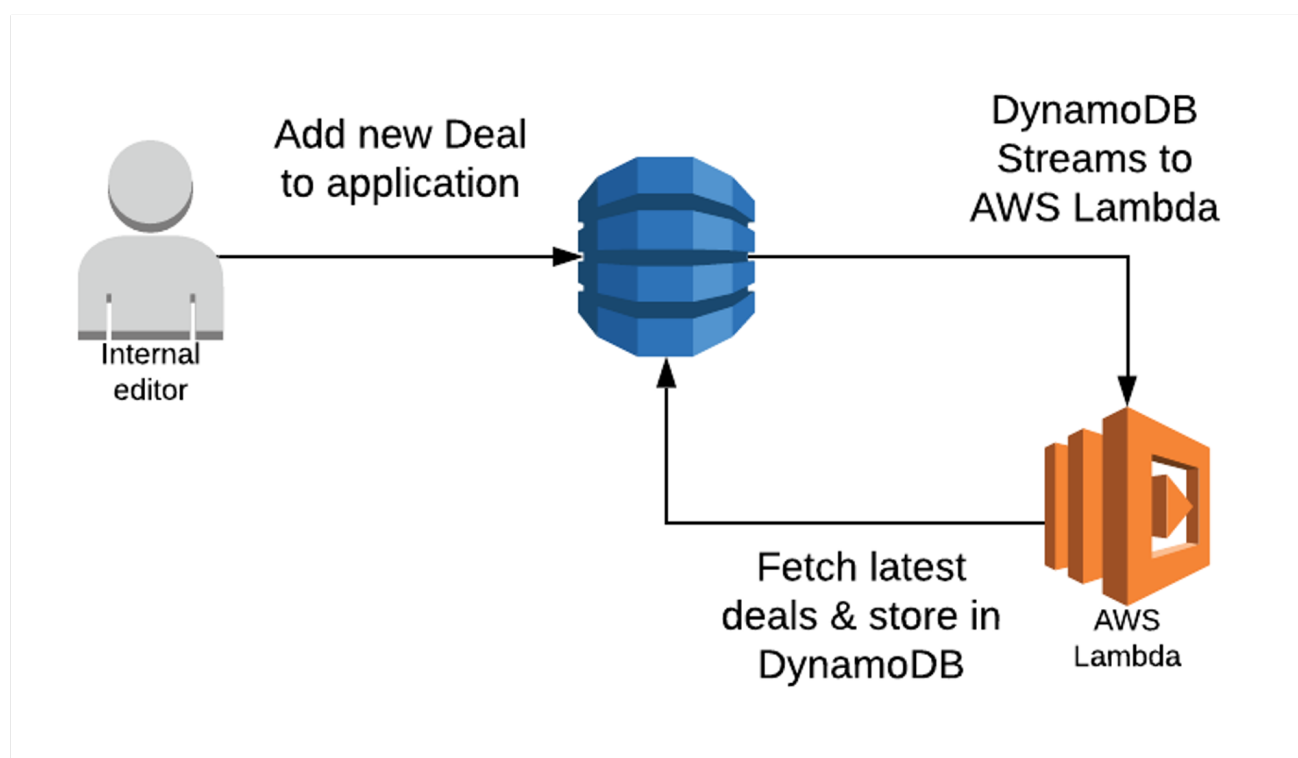
To create these little caches, we'll add some items to our tables that look as follows:

Primary key		Attributes
Partition key: PK	**Sort key: SK**	
DEALSCACHE#1	DEALSCACHE#1	Deals
		[{"DealId":"1WK0JFwUfFNUw9hddaTjEDpizTV","Link":"..."}]
DEALSCACHE#2	DEALSCACHE#2	Deals
		[{"DealId":"1WK0JFwUfFNUw9hddaTjEDpizTV","Link":"..."}]
DEALSCACHE#3	DEALSCACHE#3	Deals
		[{"DealId":"1WK0JFwUfFNUw9hddaTjEDpizTV","Link":"..."}]
DEALSCACHE#9	DEALSCACHE#9	Deals
		[{"DealId":"1WK0JFwUfFNUw9hddaTjEDpizTV","Link":"..."}]

After retrieving all deals for the last two days, we store them in the `Deals` attribute on our cache items. Then we make N items in DynamoDB, where N is the number of copies of this data that we want. The PK and SK for this item will be `DEALSCACHE#<CacheNumber>`.

Again, each of these cached items are exactly the same. It just helps us to spread the reads across multiple partitions rather than hitting the same item with a ton of traffic.

When fetching those deals, your code could look like this:

```python
import random

shard = random.randint(1, 10)

resp = client.get_item(
    TableName='BigTimeDeals',
    Key={
        'PK': { 'S': f"DEALSCACHE#{shard}" },
        'SK': { 'S': f"DEALSCACHE#{shard}" },
    }
)

return resp['Attributes']['Deals']['L']
```

In this code, we're getting a random integer between 1 and N, where N is the number of items we've created. Then we go fetch the `DEALSCACHE#<Shard>` item for the integer we generated.

Now when we get a bunch of users asking for the latest deals, our

flow looks like this:
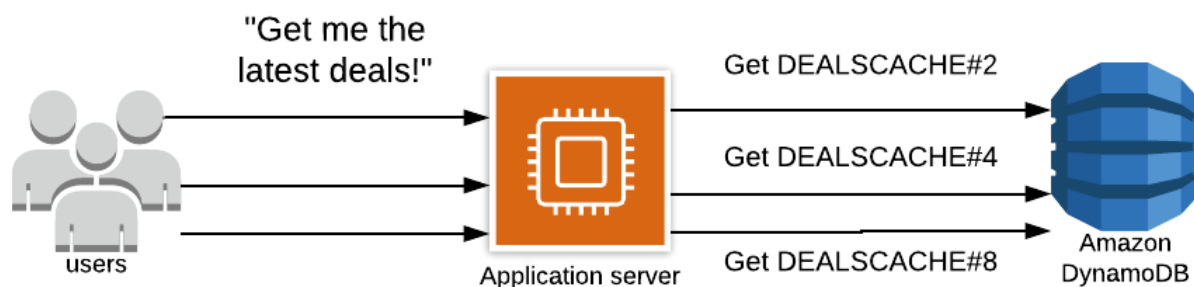
These requests are fanned out to different items in our DynamoDB table to avoid hitting that 3000 RCU limit on a particular partition.

The great thing about this pattern is that it's easy to add on later if we need it, and we can also increase the number of shards as needed. To increase the shards, just update our write code to write more items, then update the random integer generator in our read path.

Let's update our entity chart with our Deal item. Note that I'm not going to include the DealsCache item as this is an optional addition you could do.

Entity	PK	SK
Deal	`DEAL#<DealId>`	`DEAL#<DealId>`
Brand		
Category		
FeaturedDeal		
Page		
User		
Message		

Table 26. Big Time Deals entity chart

We also have some attributes in our GSI1 secondary index:

Entity	GSI1PK	GSI1SK
Deal	`DEALS#<TruncatedTimestamp>`	`DEAL#<DealId>`
Brand		
Category		
FeaturedDeal		
Page		
User		
Message		

Table 27. Big Time Deals GSI1 entity chart

Fetching most recent deals by Brand

Now that we've handled fetching the most recent deals overall, let's work on the similar pattern of fetching the most recent deals for a particular brand.

Notice that our access pattern is "Fetch Brand and Latest Deals for Brand". This implies a 'pre-join' access pattern where we place the parent Brand item in the same item collection as the related Deal items.

However, that doesn't fit well with our partitioning strategy on latest deals. Remember that we're splitting our Deal items across partitions based on a truncated timestamp. This means that our Deal items will be in multiple different item collections. Our Brand item can only be in *one* of those item collections. Thus, we need to make the choice about abandoning our partitioning scheme for deals in a brand, or we need to make multiple requests when fetching the Brand item and the latest Deal items for that brand.

I'm going to go with the latter for two reasons. First, I'm still concerned about that partition getting too big, so I'd prefer to break it up. Second, we'll be able to make these two calls (one to fetch the

Brand item and one to fetch the latest Deals) in parallel, so we don't have sequential, waterfall requests that really slow down our access patterns.

Given that, let's add the following attributes to our Deal items:

- **GSI2PK:** `BRAND#<Brand>#<TruncatedTimestamp>`
- **GSI2SK:** `DEAL#<DealId>`

With this pattern, we can use the `GSI2` index to handle the "Fetch latest deals for brand" access pattern using similar code as the "Fetch latest deals overall" that we saw above.

Fetching most recent deals by Category

Finally, we have a third "Fetch most recent deals" access pattern. This one fetches most recent deals by Category.

The principles here are exactly the same as the previous one of fetching the latest deals by Brand. We can handle it by adding the following attributes to our Deal items:

- **GSI3PK:** `CATEGORY#<Category>#<TruncatedTimestamp>`
- **GSI3SK:** `DEAL#<DealId>`

Then we would use the `GSI3` index to fetch the most recent deals for a category.

20.3.2. Modeling the Brand item

Now that we've got our "Fetch latest deals" access patterns finished, let's move on to a different entity type. We already talked a bit about the Brand-based access patterns, so let's go there.

In addition to the basic Create & Read Brand access patterns, we also have the following access patterns:

- Fetch all Brands
- Watch Brand for User
- Like Brand for User

We also had the "Fetch Brand and latest Deals for Brand", but we partially covered that before. Given that, it's now turned into a simple "Fetch Brand" access pattern with a second request to get the latest deals for the brand.

Let's model the Brand entity and handle the "Fetch all Brands" access patterns first.

Because the Brand entity is not the subject of any one-to-many relationships, we'll go with a simple PK and SK design for our Brand item:

- **PK:** `BRAND#<Brand>`
- **SK:** `BRAND#<Brand>`

This will handle our basic Create & Read Brand access patterns.

To handle the "Fetch all Brands" access patterns, we will need to put all of our brands in the same item collection. In general, I'm very leery of putting all of one entity type into a single item collection. These partitions can get fat and then you lose the partitioning strategy that DynamoDB wants you to use.

However, let's think more about our access pattern here. When fetching all Brands, all we need is the Brand name. All Brands are shown in a list view. Users can then see more about a particular Brand by clicking on the Brand to go to its page.

Because we have full control over when a Brand is created, we'll

create a *singleton item*. This singleton item is a container for a bunch of data, and it doesn't have any parameters in the primary key like most of our entities do. You can read more about singleton items in Chapter 16.

This singleton item will use `BRANDS` as the value for both the PK and SK attributes. It will have a single attribute, `Brands`, which is an attribute of type set containing all Brands in our application. Now when fetching the list of Brands, we just need to fetch that item and display the names to users.

The table with our Brand items and the Brands singleton item looks as follows:

Primary key		Attributes	
Partition key: PK	Sort key: SK		
BRAND#APPLE	BRAND#APPLE	BrandName	BrandLogoUrl
		Apple	https://...
BRAND#AMAZON	BRAND#AMAZON	BrandName	BrandLogoUrl
		Amazon	https://...
BRAND#GUCCI	BRAND#GUCCI	BrandName	BrandLogoUrl
		Gucci	https://...
BRANDS	BRANDS	Brands	
		["Amazon", "Apple", "Gucci"]	

Notice that we have a few individual Brand items for Amazon, Apple, and Gucci. Additionally, we have the Brands singleton item that contains a list of all the brands in our application.

One last note: if you're worried about hot key issues on the Brands singleton item, you could copy the data across multiple items like we did with our Deals cache items.

Handling Brand Likes

While we're on Brands, let's handle the access pattern for a user to "Like" a Brand. There are two factors to consider here:

1. We want to prevent users from liking a Brand more than once, and

2. We want to display how many users have liked a Brand.

There are two ways for us to handle the first issue: by using a set attribute on the Brand item or by creating a separate item to indicate the 'Like'.

The first option of using a set attribute only works if the total number of 'likers' would be small. Remember that we have a 400KB limit on a DynamoDB item. In this case, the number of users that like a particular Brand could be unbounded, so using a set attribute is infeasible.

Given that, we'll keep track of each Like in a separate item. The primary key pattern of this separate item will be as follows:

- **PK:** `BRANDLIKE#<Brand>#<Username>`
- **SK:** `BRANDLIKE#<Brand>#<Username>`

Because the Brand Like item includes both the Brand name and the User name, we can prevent multiple likes by including a condition expression to assert that the Brand Like doesn't currently exist.

We also mentioned we want to display how many users have liked a Brand. It would be inefficient to query and count *all* the Brand Like items each time we wanted to display the Brand. Rather, we'll maintain a `likes` attribute on the Brand item. Whenever a user tries to like a Brand, we'll increment this counter.

Notice that these two operations need to happen together. We want to increment the counter when a new like happens, but we only want to increment and record the like if the user hasn't already liked it. This is a great use case for a DynamoDB transaction.

The code to handle this transaction would be as follows:

```
result = dynamodb.transact_write_items(
    TransactItems=[
        {
            "Put": {
                "Item": {
                    "PK": { "S": "BRANDLIKE#APPLE#alexdebrie" },
                    "SK": { "S": "BRANDLIKE#APPLE#alexdebrie" },
                    ...rest of attributes ...
                },
                "TableName": "BigTimeDeals",
                "ConditionExpression": "attribute_not_exists(PK)"
            }
        },
        {
            "Update": {
                "Key": {
                    "PK": { "S": "BRAND#APPLE" },
                    "SK": { "S": "BRAND#APPLE" },
                },
                "TableName": "BigTimeDeals",
                "ConditionExpression": "attribute_exists(PK)"
                "UpdateExpression": "SET #likes = #likes + :incr",
                "ExpressionAttributeNames": {
                    "#likes": "LikesCount"
                },
                "ExpressionAttributeValues": {
                    ":incr": { "N": "1" }
                }
            }
        }
    ]
)
```

This transaction has two operations. The first tries to create a Brand Like item for the User "alexdebrie" for the Brand "Apple". Notice that it includes a condition expression to ensure the Brand Like item doesn't already exist, which would indicate the user already liked this Brand.

The second operation increments the `LikesCount` on the Apple

Brand item by 1. It also includes a condition expression to ensure the Brand exists.

Remember that each operation will succeed only if the other operation also succeeds. If the Brand Like already exists, the LikesCount won't be incremented. If the Brand doesn't exist, the Brand Like won't be created.

Handling Brand Watches

In addition to liking a Brand, users may also *watch* a Brand. When a user is watching a Brand, they will also be notified whenever a new Deal is added for a Brand.

The patterns here are pretty similar to the Brand Like discussion, with one caveat: our internal system needs to be able to find all watchers for a Brand so that we can notify them.

Because of that, we'll put all Brand Watch items in the same item collection so that we can run a Query operation on it.

We'll use the following primary key pattern:

- **PK:** `BRANDWATCH#<Brand>`
- **SK:** `USER#<Username>`

Notice that Username is not a part of the partition key in this one, so all Brand Watch items will be in the same item collection.

We'll still use the same DynamoDB Transaction workflow when adding a watcher to increase the WatchCount while ensuring that the user has not previously watched the Brand.

Sending New Brand Deal Messages to Brand Watchers

One of our access patterns is to send a New Brand Deal Message to all Users that are watching a Brand. Let's discuss how to handle that here.

Remember that we can use DynamoDB Streams to react to changes in our DynamoDB table. To handle this access pattern, we will subscribe to the DynamoDB stream. Whenever we receive an event on our stream, we will do the following:

1. Check to see if the event is indicating that a Deal item was inserted into our table;

2. If yes, use the Query operation to find all BrandWatch items for the Brand of the new Deal;

3. For each BrandWatch item found, create a new Message item for the user that alerts them of the new Brand Deal.

For steps 2 & 3, the code will be similar to the following:

```python
resp = dynamodb.query(
    TableName='BigTimeDeals',
    KeyConditionExpression="#pk = :pk",
    ExpressionAttributeNames={
        '#pk': "PK"
    },
    ExpressionAttributeValues={
        ':pk': { 'S': "BRANDWATCH#APPLE" }
    }
)

for item in resp['Items']:
    username = item['Username']['S']
    send_message_to_brand_watcher(username)
```

First, we run a Query operation to fetch all the watchers for the given Brand. Then we send a message to each watcher to alert them of the new deal.

Patterns like this make it much easier to handle reactive functionality without slowing down the hot paths of your application.

20.3.3. Modeling the Category item

Now that we've handled the Brand item, let's handle the Category item as well. Categories are very similar to Brands, so we won't cover it in quite as much detail.

There are two main differences between the Categories and Brands:

1. Categories do not have a "Fetch all Categories" access pattern, as there are only eight categories that do not change.
2. The Categories page has a list of 5-10 "Featured Deals" within the Category.

For the first difference, that means we won't have a singleton "CATEGORIES" item like we had with Brands.

For the second difference, we need to figure out how to indicate a Deal is 'Featured' within a particular Category. One option would be to add some additional attributes and use a sparse index pattern to group a Category with its Featured Deals. However, that seems a bit overweight. We know that we only have a limited number of Featured Deals within a particular Category.

Instead, let's combine our two denormalization strategies from the one-to-many relationships chapter. We'll store information about Featured Deals in a complex attribute on the Category itself, and this information will be duplicated from the underlying Deal item.

Our Category items would look as follows:

Primary key		Attributes	
Partition key: PK	**Sort key: SK**		
CATEGORY#TECH	CATEGORY#TECH	CategoryName	FeaturedDeals
		Tech	[{"DealTitle":"Refurbished 15'' MacBook Pro"}]
CATEGORY#TRAVEL	CATEGORY#TRAVEL	CategoryName	FeaturedDeals
		Travel	[{"DealTitle":"8-day Caribbean Cruise"}]

Each Category item includes information about the Featured Deals on the item directly. Remember that setting Featured Deals is an internal use case, so we can program our internal CMS such that it includes all information about all Featured Deals whenever an editor is setting the Featured Deals for a Category.

Thus, for modeling out the Category items and related entities, we create the following item types:

Category

- **PK:** `CATEGORY#<Category>`
- **SK:** `CATEGORY#<Category>`

CategoryLike

- **PK:** `CATEGORYLIKE#<Category>#<Username>`
- **SK:** `CATEGORYLIKE#<Category>#<Username>`

CategoryWatch

- **PK:** `CATEGORYWATCH#<Category>`
- **SK:** `USER#<Username>`

20.3.4. Featured Deals and Editors' Choice

We're almost done with the Deals-related portion of this data

model. The last thing we need to handle is around the Featured Deals on the front page of the application and the Editor's Choice page.

For me, this problem is similar to the "Featured Deals for Category" problem that we just addressed. We could add some attributes to a Deal to indicate that it's featured and shuttle them off into a secondary index. In contrast, we could go a much simpler route by just duplicating some of that data elsewhere.

Let's use a combination of the 'singleton item' strategy we discussed earlier with this duplication strategy. We'll create two new singleton items: one for the Front Page and one for the Editor's Choice page.

The table might look as follows:

Notice the singleton items for the Front Page and the Editor's Choice page. Additionally, like we did with the Deals Cache items, we could copy those across a number of partitions if needed. This is a simple, effective way to handle these groupings of featured deals.

Let's take a breath here and take a look at our updated entity chart with all Deal-related items finished.

Entity	PK	SK
Deal	`DEAL#<DealId>`	`DEAL#<DealId>`
Brand	`BRAND#<Brand>`	`BRAND#<Brand>`
Brands	`BRANDS`	`BRANDS`
BrandLike	`BRANDLIKE#<Brand>#<Username>`	`BRANDLIKE#<Brand>#<Username>`
BrandWatch	`BRANDWATCH#<Brand>`	`USER#<Username>`
Category	`CATEGORY#<Category>`	`CATEGORY#<Category>`
CategoryLike	`CATEGORYLIKE#<Category>#<Username>`	`CATEGORYLIKE#<Category>#<Username>`
CategoryWatch	`CATEGORYWATCH#<Category>`	`USER#<Username>`
FrontPage	`FRONTPAGE`	`FRONTPAGE`
Editor's Choice	`EDITORSCHOICE`	`EDITORSCHOICE`
User		
Message		

Table 28. Big Time Deals entity chart

We also have some attributes in our secondary indexes. To save space, I'll only show items that have attributes in those indexes.

First, our GSI1 secondary index:

Entity	GSI1PK	GSI1SK
Deal	`DEALS#<TruncatedTimestamp>`	`DEAL#<DealId>`

Table 29. Big Time Deals GSI1 entity chart

Then, our GSI2 secondary index:

Entity	GSI2PK	GSI2SK
Deal	`BRAND#<Brand>#<TruncatedTimestamp>`	`DEAL#<DealId>`

Table 30. Big Time Deals GSI2 entity chart

And finally, the GSI3 secondary index:

355

Entity	GSI3PK	GSI3SK
Deal	`CATEGORY#<Category>#<Truncated Timestamp>`	`DEAL#<DealId>`

Table 31. Big Time Deals GSI3 entity chart

20.3.5. Modeling the User item

Now that we have most of the entities around Deals modeled, let's move on to Users and Messages.

In addition to the simple Create / Read / Update User access patterns, we have the following access patterns that are based on Users:

- Fetch all Messages for User

- Fetch all Unread Messages for User

- Mark Message as Read

- Send new Hot Deal Message to all Users

Remember that we handled the "Send new Brand Deal Message to all Brand Watchers" and "Send new Category Deal Message to all Category Watchers" in the sections on Brands and Categories, respectively.

I'm going to start with the last access pattern—send new Hot Deal Message to All Users—then focus on the access patterns around fetching Messages.

The User item and Finding all Users

When we think about the "Send new Hot Deal Message to all Users", it's really a two-step operation:

1. Find all Users in our application

2. For each User, send a Message

For the 'find all Users' portion of it, we might think to mimic what we did for Brands: use a singleton item to hold all usernames in an attribute. However, the number of Users we'll have is unbounded. If we want our application to be successful, we want to have as many Users as possible, which means we *want* to exceed 400KB of data.

A second approach we could do is to put all Users into a single partition. For example, we could have a secondary index where each User item had a static partition key like `USERS` so they were all grouped together. However, this could lead to hot key issues. Each change to a User item would result in a write to the same partition in the secondary index, which would result in a huge number of writes.

Instead, let's use one of our sparse indexing strategies from Chapter 13. Here, we want to use the second type of sparse index, which projects only a single type of entity into a table.

To do this, we'll create a User entity with the following attributes:

- **PK**: `USER#<Username>`

- **SK**: `USER#<Username>`

- **UserIndex**: `USER#<Username>`

Our table with some User items will look as follows:

Primary key		Attributes		
Partition key: PK	**Sort key: SK**			
DEAL#1WK0JFwUfFNUw9hddaTjEDpizTV	DEAL#1WK0JFwUfFNUw9hddaTjEDpizTV	CreatedAt	Title	DealId
		2020-01-13T02:35:04	8-day Asian Cruise	1WK0JFwUfFNUw9hddaTjEDpizTV
USER#alexdebrie	USER#alexdebrie	Username	CreatedAt	UserIndex
		alexdebrie	2020-02-29 11:21:03	USER#alexdebrie
USER#jeffbezos	USER#jeffbezos	Username	CreatedAt	UserIndex
		jeffbezos	2020-01-09 20:20:51	USER#jeffbezos
USER#warrenbuffett	USER#warrenbuffett	Username	CreatedAt	UserIndex
		warrenbuffett	2020-01-12 16:20:41	USER#warrenbuffett

Notice that we have three User items in our table. I've also placed a Deal item to help demonstrate how our sparse index works.

Each of our User items has a `UserIndex` attribute. We will create a secondary index that uses the `UserIndex` attribute as the partition key. That index looks as follows:

Primary key	Attributes			
Partition key: UserIndex				
USER#alexdebrie	PK	SK	Username	CreatedAt
	USER#alexdebrie	USER#alexdebrie	alexdebrie	2020-02-29 11:21:03
USER#jeffbezos	PK	SK	Username	CreatedAt
	USER#jeffbezos	USER#jeffbezos	jeffbezos	2020-01-09 20:20:51
USER#warrenbuffett	PK	SK	Username	CreatedAt
	USER#warrenbuffett	USER#warrenbuffett	warrenbuffett	2020-01-12 16:20:41

Notice that only our User items have been copied into this table. Because other items won't have that attribute, this is a *sparse* index containing just Users.

Now if we want to message every User in our application, we can use the following code:

```
resp = dynamodb.scan(
    TableName='BigTimeDeals',
    IndexName='UserIndex'
)

for item in resp['Items']:
    username = item['Username']['S']
    send_message_to_user(username)
```

This is a similar pattern to what we did when sending messages to Brand or Category Watchers. However, rather than using the Query operation on a partition key, we're using the Scan operation on an index. We will scan our index, then message each User that we find in our index.

One final note on sparse indexes: an astute observer might note that *all* of our secondary indexes are sparse indexes. After all, only the Deal item has been projected into GSI1, GSI2, and GSI3. What's the difference here?

With the UserIndex, we're intentionally using the sparseness of the index to aid in our filtering strategy. As we move on, we may include other items in our GSI1 index. However, we can't add other items into our UserIndex as that would defeat the purpose of that index.

Handling our User Messages

The final access patterns we need to handle are with the Messages for a particular User. There are three access patterns here:

- Find all Messages for User

- Find all Unread Messages for User

- Mark Message as Read

Notice the first two are the same pattern with an additional filter condition. Let's think through our different filtering strategies from Chapter 13 on how we handle the two different patterns.

We probably won't want to distinguish between these using a partition key (e.g. putting read and unread messages for a User in different partitions). This would require twice as many reads on the

'Find all Messages' access pattern, and it will be difficult to find the exact number of Messages we want without overfetching.

Likewise, we can't use the sort key to filter here, whether we use it directly or as part of a composite sort key. The composite sort key works best when you *always* want to filter on a particular value. Here, we sometimes want to filter and sometimes don't.

That leaves us to two types of strategies: using a sparse index, or overfetching and then filtering apart from the core access of DynamoDB, either with a filter expression or with client-side filtering. I like to avoid the overfetching strategies unless there are a wide variety of filtering patterns we need to support. Since we only need to provide one here, let's go with a sparse index.

When modeling Users, we used the sparse index to project only a particular type of entity into an index. Here, we're going to use the other sparse index strategy where we filter within a particular entity type.

Let's create our Message item with the following pattern:

- **PK:** `MESSAGES#<Username>`
- **SK:** `MESSAGE#<MessageId>`
- **GSI1PK:** `MESSAGES#<Username>`
- **GSI1SK:** `MESSAGE#<MessageId>`

For the MessageId, we'll stick with the KSUID that we used for Deals and discussed in Chapter 14.

Note that the PK & SK patterns are the exact same as the GSI1PK and GSI1SK patterns. The distinction is that the GSI1 attributes *will only be added for unread Messages*. Thus, GSI1 will be a sparse index for unread Messages.

Our table with some Message items looks as follows:

Primary key		Attributes					
Partition key: PK	Sort key: SK						
MESSAGE# alexdebrie	MESSAGE#2020-02-25 15:57:45	Subject	Body	Unread			
		New Nike Deal!	Ready to take flight? Check out this new deal	False			
	MESSAGE#2020-03-01 05:24:50	Subject	Body	Unread	GSI1PK	GSI1SK	
		Relax in the Bahamas!	Get your tan on in the Bahamas	True	MESSAGE# alexdebrie	MESSAGE# 2020-03-01 05:24:50	
	MESSAGE#2020-03-11 18:27:14	Subject	Body	Unread	GSI1PK	GSI1SK	
		Le Creuset Sale!	It's Dutch Oven time ...	True	MESSAGE# alexdebrie	MESSAGE# 2020-03-11 18:27:14	
MESSAGE#j effbezos	MESSAGE#2020-02-29 11:21:03	Subject	Body	Unread	GSI1PK	GSI1SK	
		Amazon Prime Discount!	Hey Jeff, do you like Amazon Prime? ...	True	MESSAGE#j effbezos	MESSAGE# 2020-02-29 11:21:03	

We have four Messages in our table. They're grouped according to Username, which makes it easy to retrieve all Messages for a User. Also notice that three of the Messages are unread. For those three Messages, they have `GSI1PK` and `GSI1SK` values.

When we look at our GSI1 secondary index, we'll see only unread Messages for a User:

Primary key		Attributes				
Partition key: GSI1PK	Sort key: GSI1SK					
MESSAGE# alexdebrie	MESSAGE#2020-03-01 05:24:50	PK	SK	Subject	Body	Unread
		MESSAGE# alexdebrie	MESSAGE#2020-03-01 05:24:50	Relax in the Bahamas!	Get your tan on in the Bahamas	True
	MESSAGE#2020-03-11 18:27:14	PK	SK	Subject	Body	Unread
		MESSAGE# alexdebrie	MESSAGE#2020-03-11 18:27:14	Le Creuset Sale!	It's Dutch Oven time ...	True
MESSAGE#j effbezos	MESSAGE#2020-02-29 11:21:03	PK	SK	Subject	Body	Unread
		MESSAGE#j effbezos	MESSAGE#2020-02-29 11:21:03	Amazon Prime Discount!	Hey Jeff, do you like Amazon Prime? ...	True

This lets us quickly retrieve unread Messages for a User.

The modeling part is important, but I don't want to leave out how we implement this in code either. Let's walk through a few code snippets.

First, when creating a new Message, it will be marked as Unread. Our `create_message` function will handle adding the GSI1 attributes:

```python
def create_message(message):
    resp = client.put_item(
        TableName='BigTimeDeals',
        Item={
            'PK': { 'S': f"MESSAGE#{message.username}" },
            'SK': { 'S': f"MESSAGE#{message.created_at}" },
            'Subject': { 'S': f"MESSAGE#{message.subject}" },
            'Unread': { 'S': "True" },
            'GSI1PK': { 'S': f"MESSAGE#{message.username}" },
            'GSI1SK': { 'S': f"MESSAGE#{message.created_at}" },
        }
    )

    return message
```

Notice that the caller of our function doesn't need to add the GSI1 values or even think about whether the message is unread. Because it's unread by virtue of being new, we can set that property and both of the GSI1 properties in the data access layer.

Second, let's see how we would update a Message to mark it read:

```python
def mark_message_read(message):
    resp = client.update_item(
        TableName='BigTimeDeals',
        Key={
            'PK': { 'S': f"MESSAGE#{message.username}" },
            'SK': { 'S': f"MESSAGE#{message.created_at}" },
        },
        UpdateExpression="SET #unread = :false, REMOVE #gsi1pk, gsi1sk",
        ExpressionAttributeNames={
            '#unread': 'Unread',
            '#gsi1pk': 'GSI1PK',
            '#gsi1sk': 'GSI1SK'
        },
        ExpressionAttributeValues={
            ':false': { 'S': 'False' }
        }
    )

    return message
```

In this method, we would run an UpdateItem operation to do two things:

1. Change the `Unread` property to "False", and
2. Remove the GSI1PK and GSI1SK attributes so that it will be removed from the sparse index.

Again, the calling portion of our application doesn't need to worry about modifying indexing attributes on our item. That is all left in the data access portion.

Finally, let's see our code to retrieve all messages:

```python
def get_messages_for_user(username, unread_only=False):
    args = {
        'TableName': 'BigTimeDeals',
        'KeyConditionExpression': '#pk = :pk',
        'ExpressionAttributeNames': {
            '#pk': 'PK'
        },
        'ExpressionAttributeValues': {
            ':pk': { 'S': f"MESSAGE#{username}" }
        },
        'ScanIndexForward': False
    }

    if unread_only:
        args['IndexName'] = 'GSI1'
    resp = client.query(**args)
```

We can use the same method for fetching all Messages and fetching unread Messages. With our `unread_only` argument, a caller can specify whether they only want unread Messages. If that's true, we'll add the IndexName property to our Query operation. Otherwise, we'll hit our base table.

With this sparse index pattern, we're able to efficiently handle both access patterns around Messages.

20.4. Conclusion

That concludes our Big Time Deals example. We saw a lot of interesting patterns here, including:

- Arbitrary partitioning of Deals data to prevent hot partitions
- Singleton items
- Transactional operations to maintain reference counts
- DynamoDB Streams to react to incoming changes
- Sparse indexes to project a single entity type
- Sparse indexes to filter within an entity type

The final entity charts and access patterns are below, but I want to close this out first.

This is a complex pattern, but it becomes formulaic as you work through it. Once you see this in action, you start to see that you're not limited in DynamoDB. We could add more patterns as needed here without much complication. DynamoDB can handle almost anything you throw at it.

In the next chapter, we'll do another equally-complex access pattern with a more familiar setup. Check it out!

Final entity charts and access patterns

Let's look as our final entity charts. First, the entity chart for our base table:

Entity	PK	SK
Deal	`DEAL#<DealId>`	`DEAL#<DealId>`
Brand	`BRAND#<Brand>`	`BRAND#<Brand>`
Brands	`BRANDS`	`BRANDS`
BrandLike	`BRANDLIKE#<Brand>#<Username>`	`BRANDLIKE#<Brand>#<Username>`
BrandWatch	`BRANDWATCH#<Brand>`	`USER#<Username>`
Category	`CATEGORY#<Category>`	`CATEGORY#<Category>`
CategoryLike	`CATEGORYLIKE#<Category>#<Username>`	`CATEGORYLIKE#<Category>#<Username>`
CategoryWatch	`CATEGORYWATCH#<Category>`	`USER#<Username>`
FrontPage	`FRONTPAGE`	`FRONTPAGE`
Editor's Choice	`EDITORSCHOICE`	`EDITORSCHOICE`
User	`USER#<Username>`	`USER#<Username>`
Message	`MESSAGES#<Username>`	`MESSAGE#<MessageId>`

Table 32. Big Time Deals entity chart

Then, our GSI1 secondary index:

Entity	GSI1PK	GSI1SK
Deal	`DEALS#<TruncatedTimestamp>`	`DEAL#<DealId>`
UnreadMessages	`MESSAGES#<Username>`	`MESSAGE#<MessageId>`

Table 33. Big Time Deals GSI1 entity chart

Then, our GSI2 secondary index:

Entity	GSI2PK	GSI2SK
Deal	`BRAND#<Brand>#<TruncatedTimestamp>`	`DEAL#<DealId>`

Table 34. Big Time Deals GSI2 entity chart

And finally, the GSI3 secondary index:

Entity	GSI3PK	GSI3SK
Deal	`CATEGORY#<Category>#<TruncatedTimestamp>`	`DEAL#<DealId>`

Table 35. Big Time Deals GSI3 entity chart

And here's our access pattern list (warning, this is long!):

Access Pattern	Index	Parameters	Notes
Create Deal	N/A	N/A	Will happen in internal CMS
Create Brand	N/A	N/A	Add to BRANDS container object
Create Category	N/A	N/A	Fixed number of categories (8)
Set Featured Deals for Front Page	N/A	N/A	Will happen in internal CMS. Send up all featured deals.
Set Featured Deals for Category	N/A	N/A	Will happen in internal CMS. Send up all featured deals.
Set Featured Deals for Editor's Choice Page	N/A	N/A	Will happen in internal CMS. Send up all featured deals.

Access Pattern	Index	Parameters	Notes
Fetch Front Page & Latest Deals	Main table	N/A	Fetch Front Page Item
	GSI1	• LastDealIdSeen	Query timestamp partitions for up to 25 deals
Fetch Category & Latest Deals	Main table	• CategoryName	Fetch Category Item
	GSI3	• CategoryName • LastDealIdSeen	Query timestamp partitions for up to 25 deals
Fetch Editor's Choice Page	Main table	N/A	Fetch Editor's Choice item
Fetch Latest Deals for Brand	GSI2	*BrandName	Query timestamp partitions for up to 25 deals
Fetch all Brands	Main table	N/A	Fetch BRANDS container item
Fetch Deal	Main table	• Brand	GetItem on Deal Id
Create User	Main table	N/A	Condition expression to ensure uniqueness on username
Like Brand For User	Main table	N/A	Transaction to increment Brand LikeCount and ensure User hasn't liked
WatchBrand For User	Main table	N/A	Transaction to increment Brand WatchCount and ensure User hasn't watched
Like Category For User	Main table	N/A	Transaction to increment Category LikeCount and ensure User hasn't liked

Access Pattern	Index	Parameters	Notes
WatchCategory For User	Main table	N/A	Transaction to increment Category WatchCount and ensure User hasn't watched
View Messages for User	Main table	• Username	Query to find all Messages
View Unread Messages for User	GSI1	• Username	Query to find all Messages
Mark Message as Read	Main table	N/A	Update Status and remove GSI1 attributes
Send Hot New Deal Message to all Users	User index	N/A	1. Scan UserIndex to find all Users
	Main table	N/A	2. Create Message for each User in step 1
Send new Brand Deal Message to all Brand Watchers	Main table	• BrandName	1.Query BrandWatchers partition to find watchers for Brand
	Main table	N/A	2. Create Message for each User in step 1
Send new Category Deal Message to all Category Watchers	Main table	• CategoryName	1.Query CategoryWatchers partition to find watchers for Category
	Main table	N/A	2. Create Message for each User in step 1

Chapter 21. Recreating GitHub's Backend

The last example was a pretty big one, but we're about to go even bigger. In this example, we're going to re-create the GitHub data model around core metadata. We'll model things like Repos, Users, Organizations, and more. This model contains a number of closely-related objects with a large number of access patterns. By the end of this example, you should feel confident that DynamoDB can handle highly-relational access patterns.

Let's get started!

21.1. Introduction

As usual, we'll start with some introductory work to set up the problem. First, I want to mention which elements of GitHub we'll be using as well as the elements we'll be skipping. Second, I want to cover some basics about GitHub to aid those who aren't familiar. Even if you are familiar with GitHub, I'd recommend reviewing the second part just to understand the business constraints that will affect our data modeling.

21.1.1. Coverage of the GitHub API in this example

Let's start with the elements we'll be covering. In this chapter, we will model the following concepts from GitHub:

- Repositories

- Users

- Organizations

- Payment plans

- Forks

- Issues

- Pull Requests

- Comments

- Reactions

- Stars

This is pretty broad coverage of the main concepts in GitHub. That said, there are two large categories that we left out. They are:

- **Smaller, ancillary data.** There are a number of other entities in GitHub—including Gists, Checks, Apps, Actions, Projects—that I chose to exclude. The main reason is that they provide little additional benefit. Once you've modeled ten different entities in a data model, there's little benefit to adding an eleventh. I chose the most relevant items and removed the rest.

- **Git repository contents.** A GitHub repository is a display of a set of files tracked using `git`, the open-source version-control system. Each repository contains files, branches, commits, and more.

  The deeper you get into git, the harder it is to keep track of the data modeling principles we're trying to learn. Further, much of the API around files and branches is more search-related, which would likely be done in an external system rather than in DynamoDB. Given that, we will focus on the metadata elements of GitHub rather than on the repository contents.

21.1.2. Walkthrough of the basics of GitHub

Now that we know the entities we'll be covering, let's walk through what these entities are and how they interact with each other.

Repositories

The core entity in GitHub is a repository, or 'repo' for short. A repo displays a collection of code, documentation, or other file contents that multiple users can use to collaborate.

Below is a screenshot of the GitHub repo for the AWS SDK for JavaScript. We'll walk through some of the key points below.
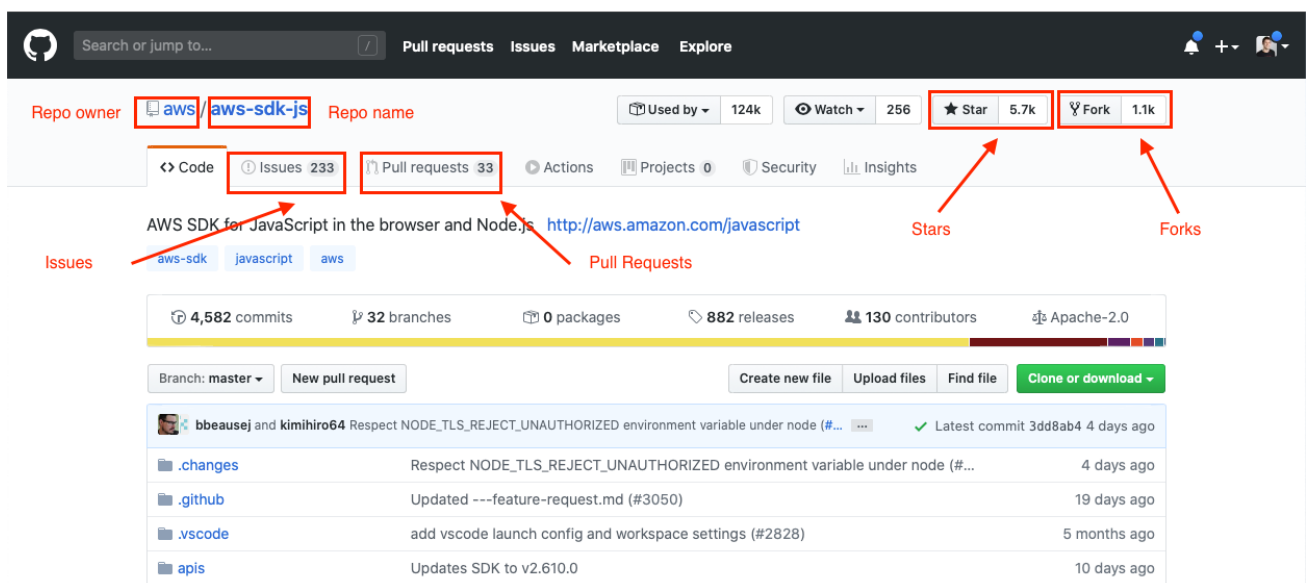
- *Repo Name.* Each repository has a name. For this example, you can see the repository name in the top left corner. This repo's name is `aws-sdk-js`.

- *Repo Owner.* Each repository has a single owner. The owner can be either a User or an Organization. For many purposes, there is no difference as to whether a repo is owned by a User or an Organization. The repository owner is shown in the top left before the repo name. This repo's owner is `aws`.

The combination of repo name and repo owner must be unique across all of GitHub. You will often see a repository referred to by the combination of its repo name and owner, such as `aws/aws-sdk-js`. Note that there is a one-to-many relationship between owners and repos. An owner may have many repos, but a repo only has one owner.

- *Issues & Pull Requests.* Below the repo owner and name, notice the tabs for "Issues" and "Pull Requests". These indicate the number of *open* issues and pull requests, respectively, for the current repo. Issues and pull requests are discussed in additional detail below.

- *Stars.* On the top right, there is a box that indicates this repo has 5.7k stars. An individual User can choose to 'star' a repository, indicating the user likes something about the repo. It's comparable to a 'like' in various social media applications. Each user can star a repo only one time.

- *Forks.* On the top right next to Stars, you can see that this repository has 1.1k forks. A user or organization can 'fork' a repository to copy the repository's existing code into a repository owned by the forking user or organization. This can be done if a user wants to make changes to submit back to the original repo, or it can be done so that the user or org has a copy of the code which they control entirely.

Users

A user represents—you guessed it—a user in GitHub. It generally corresponds to an actual human, but there's no guarantee it will. I've seen people create GitHub accounts that are used solely for automation purposes, and the same person may have one GitHub user account for their professional work and another account for their personal projects.

A user is uniquely identified by a username. Users can own repos, as discussed above. Users can also belong to one or more organizations, as discussed below.

Organizations

An organization is a shared account for a number of users. An organization will often be created by a company or a team at a company so that repos and other entities aren't tied to a specific user at the company.

Like users, an organization is uniquely identified by an organization name. Importantly, *names are unique across organizations and users*. There cannot be an organization with the same name as a user and vice versa. This is because a repository is uniquely identifable by the combination of owner and repo name.

Payment Plans

Both users and organizations can sign up for paid plans on GitHub. These plans determine which features a user or organization is allowed to use as well as the account-wide limits in place.

Forks

As mentioned above, a user or organization can fork a repository in order to make a copy of the existing contents that are owned by the forking user or organization.

Forked repositories point back to their original repo, and you can

browse the forked repositories of a given repository in the GitHub UI, as shown below.
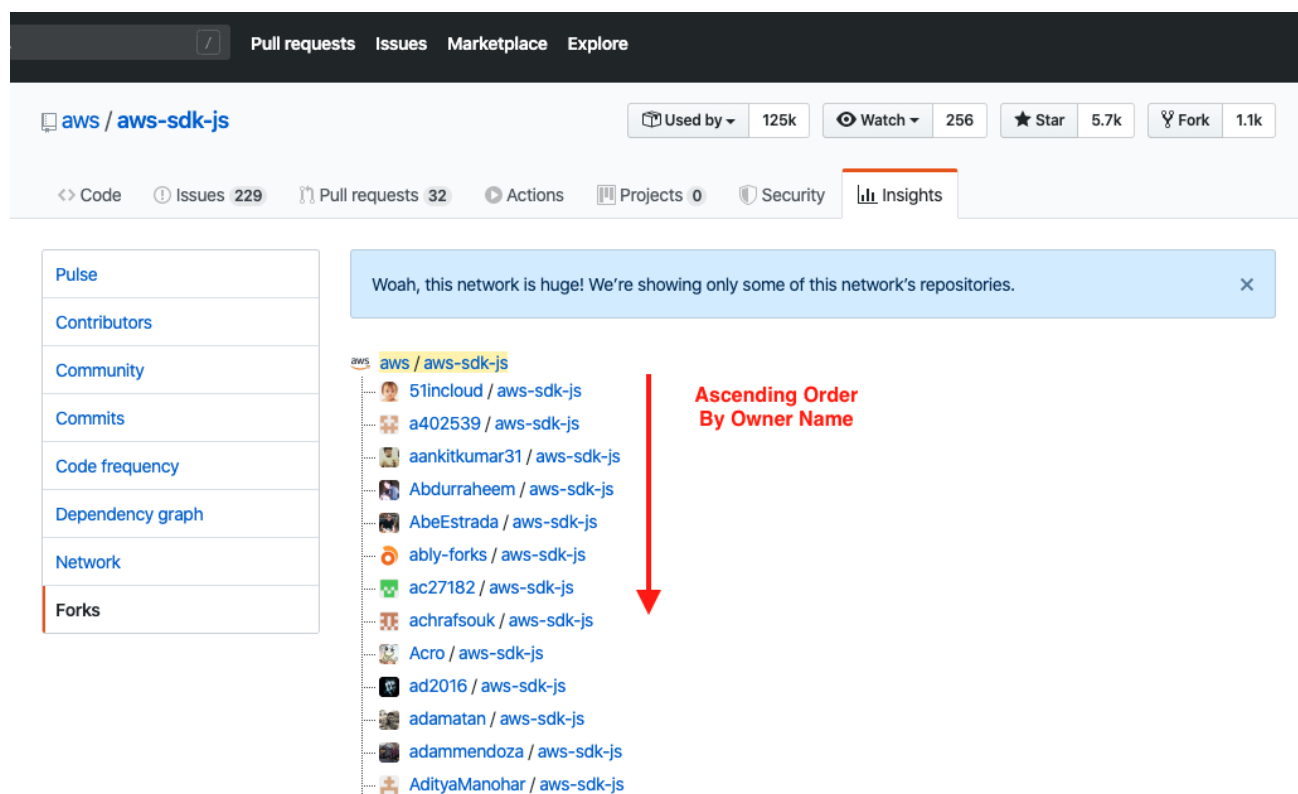
As you look at forks, notice that forks are returned in *ascending* order according to the owner (User or Organization) that forked the repo.

Issues

Users can file issues in a repository to provide feature requests, bug reports, or other notes. Issues don't have code in them; they're more like discussion topics in a particular repository.

Each issue within a repository is given an ID to identify it. This ID is an integer and is incremented within a single repository. Interestingly, issues and pull requests in a single repository are incremented *together*. Thus if you had a repository with two issues and three pull requests, the next issue that was opened would

receive an ID of `6`.

Below is a screenshot of the issues overview page in GitHub:
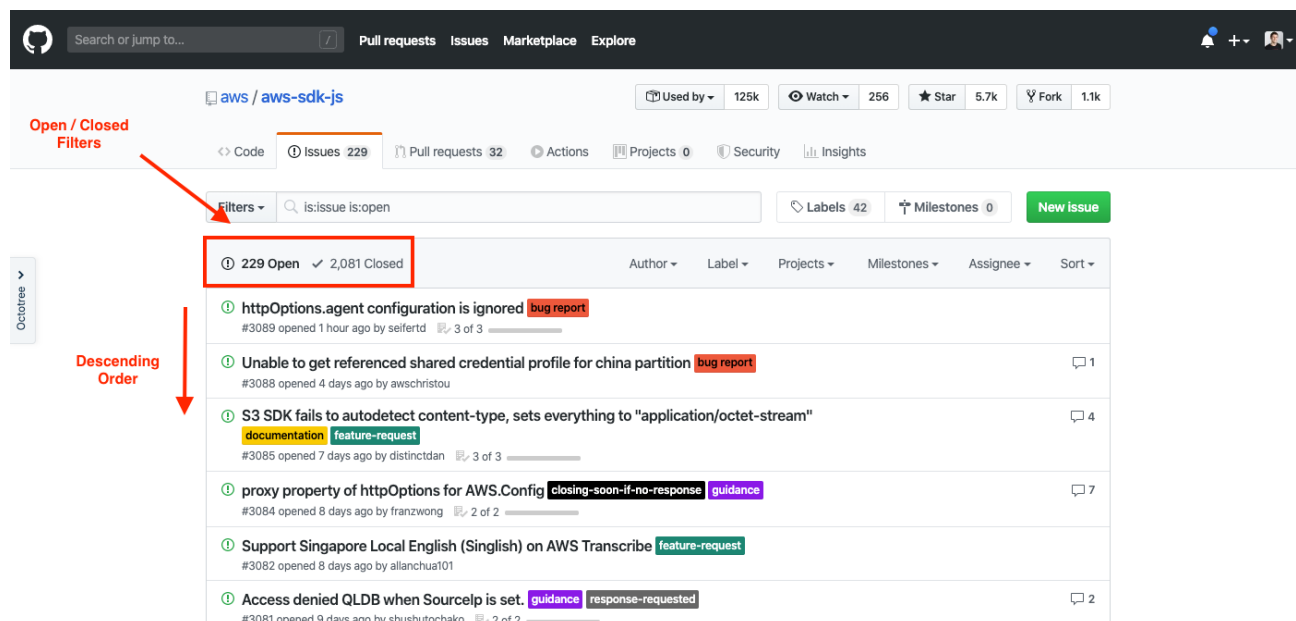
Two important things to note:

1. Issues are sorted in *descending* order by issue number. This is the key access pattern we'll handle when fetching issues.

2. Issues have a status of `Open` or `Closed`. When viewing issues in the GitHub UI, you are viewing open *or* closed issues.

Pull Requests

A pull request is a collection of git commits that include changes to the file contents of a repo. The pull request is opened as a way to suggest that the collection of commits in the pull request should be saved in the repository (this is a gross simplification of git but will work for our purposes).

As mentioned above, a pull request will receive an ID that identifies it, and new pull requests are given IDs based on the next increment

of combined issues and pull requests in the given repository.

Pull Requests are requested similar to Issues—in descending order, with default filters by status (`Open` vs. `Closed`).

Comments

Both issues and pull requests can have comments. This allows for discussion between multiple users on a particular issue or pull request. All comments are at the same level, with no threading, and there is no limit to the number of comments in an issue or PR.

Reactions

Users may react to an issue, pull request, or comment. There are eight different kinds of reactions: thumbs up, thumbs down, smiley face, celebration, disappointed face, heart, rocket, and eyes. For each issue, pull request, and comment, the number of each reaction type is stored.

The image below shows comments, reactions, and the ID for an issue. Pull requests are very similar.

Now that we know the entities that we will be modeling, take the time to create your ERD and draw up the access patterns in your application.

21.2. ERD & Access Patterns

Here is the ERD I created for the GitHub application:
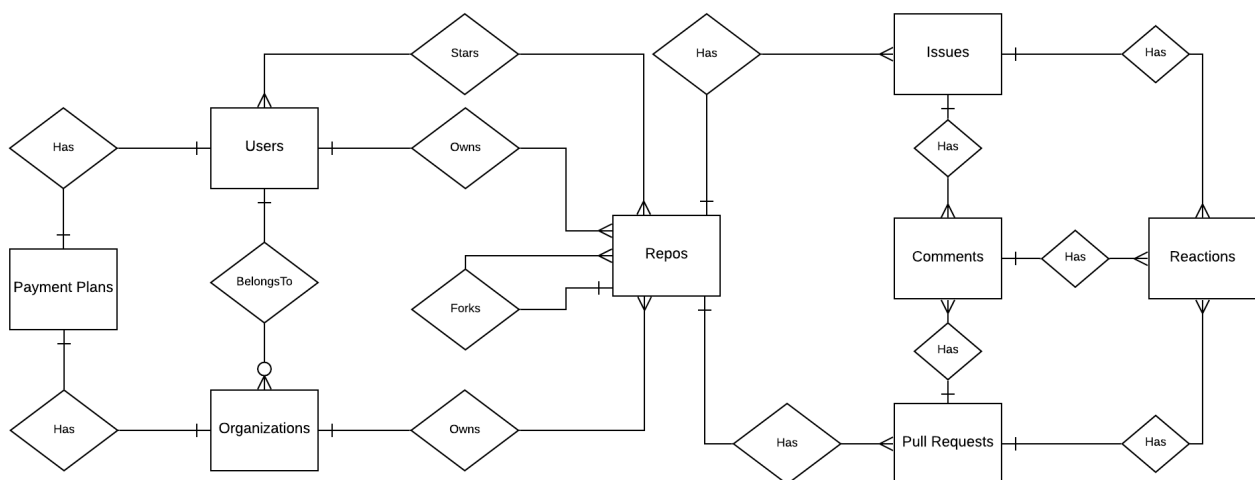
Let's walk through what's going on here.

On the left side of the ERD, we have Users and Organizations. A User may belong to multiple Organizations but need not belong to any. Likewise, an Organization may have multiple Users that belong to it. Thus, there is a many-to-many relationship between Users and Organizations.

Further, both Users and Organizations can have Payment Plans with GitHub to unlock additional features. There is a one-to-one relationship between both Users and Organizations and Payment Plans.

Moving to the center of the ERD, we see the Repo entity. Both Users and Organizations can own Repos. There is a one-to-many relationship between Users or Organizations and Repos as a User or Organization may own multiple Repos but a Repo only has one owner.

Further, a User can also star a Repo to indicate interest in the Repo's contents. There is a many-to-many relationship between Users and Repos via this Star relationship. Organizations may not add stars to repositories.

There are three one-to-many relationships where the Repo is the parent entity. First, on the left-center of the ERD, a Repo has a one-to-many relationship with other Repos via forking. When a User or Organization forks a Repo, the forked Repo maintains a relationship with its upstream.

Notice that a forked Repo is itself still a Repo as the forked version of the Repo is owned by a User or Organization and will have Issues and Pull Requests separate from the parent from which it was forked.

The two other one-to-many relationships with Repos are Issues and

Pull Requests. A single Repo will have multiple of both Issues and Pull Requests.

Issues and Pull Requests can each have Comments left by Users. While technically Issues, Pull Requests, and Comments are all left by Users, there are no access patterns related to fetching any of these entities by the User that left them. Accordingly, we didn't map the relationship here.

Finally, Issues, Pull Requests, and Comments each have a one-to-many relationship with Reactions.

Because this is a pretty advanced data model, we will be using a composite primary key in our table. I've made the following entity chart to get us started with modeling:

Entity	PK	SK
Repo		
Issue		
Pull Request		
Comment		
Reaction		
Fork		
User		
Organization		
Payment Plan		

Table 36. GitHub model entity chart

Finally, let's list the access patterns for our application. Because we have a bunch, I'll group them into categories.

Repo basics:

- Get / Create Repo
- Get / Create / List Issue(s) for Repo

- Get / Create / List Pull Request(s) for Repo
- Fork Repo
- Get Forks for Repo

Interactions:

- Add Comment to Issue
- Add Comment to Pull Request
- Add Reaction to Issue / Pull Request / Comment
- Star Repo
- Get Stargazers for Repo

User management:

- Create User
- Create Organization
- Add User to Organization
- Get Users for Organization
- Get Organizations for User

Accounts & Repos:

- Get Repos for User
- Get Repos for Organization

That's a lot of access patterns—24 in all!

21.3. Data modeling walkthrough

Now that we have our ERD and our access patterns, let's get started with the modeling.

As usual, we'll start with the same three questions:

1. Should I use a simple or composite primary key?
2. What *interesting* requirements do I have?
3. Which entity should I start modeling first?

In this example, we have complex access patterns and multiple relationships, so it's highly likely we'll want to use a composite primary key. We can start there.

In terms of interesting requirements, I think we have four. First, there are two "shared" resources. Within a Repo, both Issues and Pull Requests are given a number as an identifier. Interestingly, that ID is shared across Issues and Pull Requests in a Repo. You won't have an Issue and a Pull Request with the same ID in a Repo.

Similarly, account names are unique across all of GitHub. This applies to User and Organizations. Thus, you can't create an Organization with a name that has the same name as an existing User and vice versa.

The second interesting requirement is around Forks. There is a one-to-many relationship between Repos and Forks. However, a Fork for one person is a Repo for another! That's pretty unique, and we'll see how to handle that.

A third interesting requirement is that we have a number of places where we'll track reference counts. Repos can be starred and forked by a large number of users, and we'll want to keep a count on the Repo itself to display the total rather than query all the Fork items each time we fetch a Repo.

Finally, the last interesting requirement is that a Reaction can be applied to multiple targets. Users can react to an Issue, a Pull Request, an Issue Comment, or a Pull Request Comment. Further, there are eight different types of reactions to track! We'll need to

decide whether to store this on a single item or across eight different items.

With our interesting requirements noted, it's time to pick our first entity. I like to start with a 'core' entity in a data model to get started. For this model, the best entity to start with looks like the Repo entity. Not only does it have the most access patterns, but also it is the parent of a number of relationships. There are three one-to-many relationships—Issues, Pull Requests, and Repos (via a Fork)--as well as a many-to-many relationship with Users via Stars.

21.3.1. Modeling the Repo and related entities

To design our primary key pattern for the Repo entity, let's start with the three entities that have a one-to-many relationship with our Repo. The first thing to note is that all of these related entities— Issues, Pull Requests, and Forks—are unbounded, so we can't use the denormalization strategy for any of them.

Given that, we'll look at using the primary key + Query strategy for our one-to-many relationships. At this point, I'm actually going to remove Forks from consideration because Forks are themselves Repos. This likely means we'll need to do some grouping in a secondary index rather than in the base table.

Recall from Chapter 14 on one-to-many relationship strategies that we can usually model *two* one-to-many relationships in a single item collection by locating the parent object in the middle of the collection. The trick is that one of the access patterns needs to be accessed in ascending order, while the other needs to be accessed in descending order.

Let's think about how our two entities will be accessed:

- **Issues.** Descending order by issue number, as we'll want to fetch

the most recent issues first.

- **Pull Requests.** Descending order by pull request number, as we'll want to fetch the most recent pull requests first.

Given this, it looks like we'll only be able to model one of the one-to-many relationships in the Repo's item collection in the main table. There's really no difference when choosing between Issues & Pull Requests, so I'll just go with Issues as the one-to-many relationship to model in our primary key.

Repos and Issues

Both the Repo and Issue entities will be in the same item collection in the main table, meaning they need to have the same partition key. Let's use the following parties for these entities:

Repos:

- **PK:** `REPO#<Owner>#<RepoName>`
- **SK:** `REPO#<Owner>#<RepoName>`

Issues:

- **PK:** `REPO#<Owner>#<RepoName>`
- **SK:** `ISSUE#<ZeroPaddedIssueNumber>`

One thing to note here: for the Issue item, the Issue Number in the sort key is a *zero-padded* number. The reason for this is discussed in further detail in Chapter 14 on sorting. At a high level, if you are ordering numbers that are stored in a string attribute (as here, because we have the `ISSUE#` prefix), then you need to make sure your numbers are all the same length. String sorting is done from left to right, one character at a time, so `ISSUE#10` would come

before `ISSUE#9` because the seventh character (`1` vs `9`) is lower for the former.

Accordingly, we're going to increase our `IssueNumber` length to seven digits.

Let's take a look at how our table will look with some sample data for Repos and Issues.

Primary key		Attributes					
Partition key: PK	**Sort key: SK**						
REPO#alexdebrie#dynamodb-book	ISSUE#0000001	RepoName	RepoOwner	CreatedAt		IssueNumber	Status
		dynamodb-book	alexdebrie	2019-10-20 15:32:32		1	Open
	ISSUE#0000002	RepoName	RepoOwner	CreatedAt		IssueNumber	Status
		dynamodb-book	alexdebrie	2019-10-22 19:35:37		2	Closed
	ISSUE#0000012	RepoName	RepoOwner	CreatedAt		IssueNumber	Status
		dynamodb-book	alexdebrie	2019-11-07 03:43:42		12	Open
	REPO#alexdebrie#dynamodb-book	RepoName	RepoOwner	CreatedAt			
		dynamodb-book	alexdebrie	2019-10-07 23:38:31			
REPO#alexdebrie#graphql-demo	REPO#alexdebrie#graphql-demo	RepoName	RepoOwner	CreatedAt			
		graphql-demo	alexdebrie	2019-12-20 19:47:15			

Notice that there are two different item collections in the table so far. One is for the `alexdebrie/dynamodb-book` repository and contains three Issue items and one Repo item. The other is for the `alexdebrie/graphql-demo` repository and contains just the Repo item.

Also notice that in the item collection for the `alexdebrie/dynamodb-book` item, the Repo item is located *after* the Issue items when ordered by the sort key. This allows us to fetch the Repo item and the most recent Issue items by reversing the sort order in a Query operation.

If I want to fetch the Repo item and the most recent Issue items, I would use the following Query API action:

```
result = dynamodb.query(
    TableName='GitHubTable',
    KeyConditionExpression="#pk = :pk",
    ExpressionAttributeNames={
        "#pk": "PK",
    },
    ExpressionAttributeValues={
        ":pk": { "S": "REPO#alexdebrie#dynamodb-book" },
    },
    ScanIndexForward=False
)
```

A few notes here:

1. We do an exact match on the partition key (PK) as is required by all Query operations.

2. We want to move *up* our item collection in descending order. This will give us the Repo item and the Issue items in descending order. To do this, we set ScanIndexForward=False.

With this primary key design, we now have the basics around creating Repos and Issues. We can also handle the "Fetch Repo & Issues for Repo" access pattern.

We can update our entity chart to look like this:

Entity	PK	SK
Repo	REPO#<Owner>#<RepoName>	REPO#<Owner>#<RepoName>
Issue	REPO#<Owner>#<RepoName>	ISSUE#<ZeroPaddedIssueNumber>
Pull Request		
Comment		
Reaction		
Fork		
User		
Organization		
Payment Plan		

Table 37. GitHub model entity chart

Later on, we'll cover two other tricky things about Issues:

1. How to handle auto-incrementing numbers for Issues & Pull Requests
2. How to filter on Open vs. Closed Issues

Before we do that, let's model Pull Requests since both of those needs are present there as well.

Pull Requests

Now that we've handled the Repo and Issue items, let's move on to the Pull Requests item. Pull requests are very similar to issues. We need to be able to access them individually, when a user is browsing to a specific pull request. Further, we need to be able to fetch a group of Pull Request items for a given repository, along with the Repo item that has information about the parent repository. Finally, pull requests are often retrieved in descending order according to their pull request number.

To handle the Pull Request access patterns, let's first add our Pull Request items to the table. We already learned we can't have them in the same item collection as the Repo item in our main table, so we'll split them into a different collection.

We'll use the following pattern for the PK and SK of Pull Request items:

- **PK**: `PR#<Owner>#<RepoName>#<ZeroPaddedPullRequestNumber>`
- **SK**: `PR#<Owner>#<RepoName>#<ZeroPaddedPullRequestNumber>`

These key values are long, but we need to assure uniqueness for each pull request across all repositories.

Next, we'll use a global secondary index to put Repos and Pull Requests in the same item collection. To do this, we'll add the following attributes to Repos and Pull Requests:

Repos:

- **GSI1PK:** `REPO#<Owner>#<RepoName>`
- **GSI1SK:** `REPO#<Owner>#<RepoName>`

Pull Requests:

- **GSI1PK:** `REPO#<Owner>#<RepoName>`
- **GSI1SK:** `PR#<ZeroPaddedPullRequestNumber>`

Our main table will look as follows:

Primary key		Attributes					
Partition key: PK	Sort key: SK						
REPO#alexdebrie#dynamodb-book	ISSUE#0000001	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	
		dynamodb-book	alexdebrie	2019-10-20 15:32:32	1	Open	
	ISSUE#0000002	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	
		dynamodb-book	alexdebrie	2019-10-22 19:35:37	2	Closed	
	ISSUE#0000012	RepoName	RepoOwner	CreatedAt	IssueNumber	Status	
		dynamodb-book	alexdebrie	2019-11-07 03:43:42	12	Open	
	REPO#alexdebrie#dynamodb-book	RepoName	RepoOwner	CreatedAt	GSI1PK	GSI1SK	
		dynamodb-book	alexdebrie	2019-10-07 23:38:31	REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book	
REPO#alexdebrie#graphql-demo	REPO#alexdebrie#graphql-demo	RepoName	RepoOwner	CreatedAt	GSI1PK	GSI1SK	
		graphql-demo	alexdebrie	2019-12-20 19:47:15	REPO#alexdebrie#graphql-demo	REPO#alexdebrie#graphql-demo	
PR#alexdebrie#dynamodb-book#1	PR#alexdebrie#dynamodb-book#0000003	RepoName	RepoOwner	CreatedAt	PullRequestNumber	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-10-12 12:18:01	3	REPO#alexdebrie#dynamodb-book	PR#0000003
PR#alexdebrie#dynamodb-book#5	PR#alexdebrie#dynamodb-book#0000004	RepoName	RepoOwner	CreatedAt	PullRequestNumber	GSI1PK	GSI1SK
		dynamodb-book	alexdebrie	2019-12-24 14:30:05	4	REPO#alexdebrie#dynamodb-book	PR#0000004

Notice that we have added two Pull Request items at the bottom of the table. Further, both Repo Items and Pull Request items have the `GSI1PK` and `GSI1SK` values.

Let's take a look at GSI1 to see our Repo items and Pull Request items together:

Primary key		Attributes					
Partition key: GSI1PK	Sort key: GSI1SK						
REPO#alexdebrie#dynamodb-book	PR#0000003	PK	SK	RepoName	RepoOwner	CreatedAt	PullRequest Number
		PR#alexdebrie#dynamodb-book#1	PR#alexdebrie#dynamodb-book#0000003	dynamodb-book	alexdebrie	2019-10-12 12:18:01	3
	PR#0000004	PK	SK	RepoName	RepoOwner	CreatedAt	PullRequest Number
		PR#alexdebrie#dynamodb-book#5	PR#alexdebrie#dynamodb-book#0000004	dynamodb-book	alexdebrie	2019-12-24 14:30:05	4
	REPO#alexdebrie#dynamodb-book	PK	SK	RepoName	RepoOwner	CreatedAt	
		REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book	dynamodb-book	alexdebrie	2019-10-07 23:38:31	
REPO#alexdebrie#graphql-demo	REPO#alexdebrie#graphql-demo	PK	SK	RepoName	RepoOwner	CreatedAt	
		REPO#alexdebrie#graphql-demo	REPO#alexdebrie#graphql-demo	graphql-demo	alexdebrie	2019-12-20 19:47:15	

This is similar to how we handle issues above. Repos & Pull Requests are in the same item collection in GSI1, so we can fetch a Repo and its most recent Pull Requests by using the Query API on the GSI1.

Now that we have both issues and pull requests modeled, let's discuss our two additional quirks on those items.

Handling auto-incrementing numbers for Issues and Pull Requests

When reviewing the basics of GitHub above, I mentioned that issues and pull requests are given an integer to identify them when they are created. This integer is an incrementing integer that is shared by issues and pull requests within a single repository. There will only be one issue or pull request with a given number in a repository.

In a relational database, auto-incrementing integers are a common

feature. They are often used as primary keys, and the database will handle it for you.

This is not the case with DynamoDB. There's no auto-incrementing integer feature. However, you can emulate it by combining a few DynamoDB primitives.

To handle this use case, we'll make two requests to DynamoDB whenever a new issue or pull request is created:

First, we'll use the UpdateItem API call to increment an attribute called `IssuesAndPullRequestCount` on the Repo item. This attribute is solely used to track the current number of issues and pull requests that have been created in the lifetime of this repository.

Importantly, we'll also set `ReturnValues="UPDATED_NEW"` so that it will return the updated value of the `IssuesAndPullRequestCount` attribute after the operation.

Second, we will make a PutItem API call to create the Issue or Pull Request item. We will use the value of the `IssuesAndPullRequestCount` attribute in the previous call to know which number we should use for our Issue or Pull Request item.

The full code example is as follows:

```
resp = client.update_item(
    TableName='GitHubTable',
    Key={
        'PK': { 'S': 'REPO#alexdebrie#dynamodb-book' },
        'SK': { 'S': 'REPO#alexdebrie#dynamodb-book' }
    }
    UpdateExpression="SET #count = #count + :incr",
    ExpressionAttributeNames={
        "#count": "IssuesAndPullRequestCount",
    },
    ExpressionAttributeValues={
        ":incr": { "N": "1" }
    },
    ReturnValues='UPDATED_NEW'
)

current_count = resp['Attributes']['IssuesAndPullRequestCount']['N']

resp = client.put_item(
    TableName='GitHubTable',
    Item={
        'PK': { 'S': 'REPO#alexdebrie#dynamodb-book' },
        'SK': { 'S': f"ISSUE#{current_count}" },
        'RepoName': { 'S': 'dynamodb-book' }
        ... other attributes ...
    }
)
```

Notice how we make the UpdateItem call, then use the returned value for `IssuesAndPullRequestCount` to populate the primary key for the PutItem request.

This will result in multiple calls to our DynamoDB table, which we prefer to avoid. However, it's the best way to get an auto-incrementing number that you use when creating a new item.

Filtering on open and closed status

In the introduction, I mentioned that GitHub filters issues and pull request by their status. The issues and pull requests you are viewing are all Open or all Closed.

How should we handle this requirement? When deciding how to

handle filtering, there are two approaches we could take:

1. Build the status (`Open` vs. `Closed`) into the primary key for either the base table or a secondary index. Use the Query API to filter to the type you want, or

2. Use a filter expression or client-side filtering to filter out the items you don't want.

I mentioned in Chapter 13 that filter expressions won't save your bad data model, and that's absolutely true. Despite that, I think there's reason to use a filter expression here for a couple reasons.

First, the values on which we're filtering are limited. There are only two options: `Open` or `Closed`. Because of this, we should have a pretty high ratio of read items to returned items.

Second, we're reading small numbers of items at a time. The GitHub API returns a maximum of 25 issues or pull requests. These items aren't huge, so it's unlikely we'd need to go through multiple pages to find 25 items that fulfill our needs.

An updated version of the Query to fetch a Repo and its most recent *open* Issues would be as follows:

```
result = dynamodb.query(
    TableName='GitHubTable',
    KeyConditionExpression="#pk = :pk",
    FilterExpression="attribute_not_exists(#status) OR #status = :status",
    ExpressionAttributeNames={
        "#pk": "PK",
        "#status": "Status"
    },
    ExpressionAttributeValues={
        ":pk": { "S": "REPO#alexdebrie#dynamodb-book" },
        "status": { "S": "Open" }
    },
    ScanIndexForward=False
)
```

Notice that there is now a `FilterExpression` parameter. It is

asserting that either the `Status` attribute does not exist (indicating it is a Repo item) or that the value of `Status` is `Open`, indicating it's an open issue.

The biggest downside to this pattern is that it's hard to know how to limit the number of items you read from the table. You could find yourself reading the full 1MB from a table even if you only needed the first 25 items. You could do a mixed approach where you first try to find 25 items that match by reading the first 75 items or so. If you need to make a follow up query, then you could do an unbounded one.

Finally, run this access pattern in production to see how it performs. If you're wasting a lot of capacity units with this filter, you could go back and build the status property into your primary key for faster querying.

> In the next chapter, we imagine that this pattern did not work as well as desired once implemented in production, so we move away from filter expressions. Check it out to see how you would model this filter into the primary key directly.

Modeling Forks

We still need to handle forks, the last one-to-many relationship with repos.

Forks is an interestig pattern. Remember that a fork to one person is a repo to another. Thus, it's unlikely that a fork would be a primary entity in our table. Rather, we'll try to do some grouping in a secondary index. We're already using GSI1 for the Repo item, so let's do this in a new index, GSI2.

To do this, we'll add `GSI2PK` and `GSI2SK` attributes on the Repo items.

If a Repo is an original Repo (not a Fork), then the pattern for these attributes will be as follows:

- **GSI2PK:** `REPO#<Owner>#<RepoName>`
- **GSI2SK:** `#REPO#<RepoName>`

Notice the # prefix in the `GSI2SK`, which is needed for sorting.

If a Repo is a Fork, then the pattern is as follows:

- **GSI2PK:** `REPO#<OriginalOwner>#<RepoName>`
- **GSI2SK:** `FORK#<Owner>`

For the Fork item, the `OriginalOwner` refers to the owner of the Repo you are forking. This will group it in the same item collection for the original Repo.

Let's see that in action.

Here are four Repo items in our table. Other items have been removed for clarity.

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book	RepoName	RepoOwner	GSI2PK	GSI2SK
		dynamodb-book	alexdebrie	REPO#alexdebrie#dynamodb-book	#REPO#alexdebrie#dynamodb-book
REPO#danny-developer#dynamodb-book	REPO#danny-developer#dynamodb-book	RepoName	RepoOwner	GSI2PK	GSI2SK
		dynamodb-book	danny-developer	REPO#alexdebrie#dynamodb-book	FORK#danny-developer
REPO#sally-scala#dynamodb-book	REPO#sally-scala#dynamodb-book	RepoName	RepoOwner	GSI2PK	GSI2SK
		dynamodb-book	sally-scala	REPO#alexdebrie#dynamodb-book	FORK#sally-scala
REPO#golang-gertrude#dynamodb-book	REPO#golang-gertrude#dynamodb-book	RepoName	RepoOwner	GSI2PK	GSI2SK
		dynamodb-book	golang-gertrude	REPO#alexdebrie#dynamodb-book	FORK#golang-gertrude

Notice that each Repo item has the standard Repo attributes, like the `RepoName` and `RepoOwner`, as well as Fork attributes, like `GSI2PK` and `GSI2SK`.

If we flip to looking at the GSI2 view, our index is as follows:

Primary key		Attributes				
Partition key: GSI2PK	**Sort key: GSI2SK**					
REPO#alexdebrie#dynamodb-book	#REPO#alexdebrie#dynamodb-book	PK	SK	RepoName	RepoOwner
		REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book	dynamodb-book	alexdebrie
	FORK#danny-developer	PK	SK	RepoName	RepoOwner
		REPO#danny-developer#dynamodb-book	REPO#danny-developer#dynamodb-book	dynamodb-book	danny-developer
	FORK#golang-gertrude	PK	SK	RepoName	RepoOwner
		REPO#golang-gertrude#dynamodb-book	REPO#golang-gertrude#dynamodb-book	dynamodb-book	golang-gertrude
	FORK#sally-scala	PK	SK	RepoName	RepoOwner
		REPO#sally-scala#dynamodb-book	REPO#sally-scala#dynamodb-book	dynamodb-book	sally-scala

Notice how all Forks for a given repository are in the same item collection. The original Repo item is at the top of the item collection and all Forks are after it in ascending order by the forking Owner's name.

Modeling Stars

We've now covered all of the one-to-many relationships with repos. We still have one outstanding relationship—a many-to-many relationship between repos and users via the "star" relationship.

We have an access pattern where we want to fetch a Repo item and all Stargazer items for a Repo. The GitHub API doesn't state any particular order for sorting Stargazers in this response, so we'll be flexible with it.

We're already storing issues items in the same item collection as Repo items in the main table, but we can still store another relation in there as well as discussed in Chapter 14. Let's put Stars in there.

We'll use the following pattern:

Stars:

- **PK:** `REPO#<RepoName>`
- **SK:** `STAR#<UserName>`

If we put a few Stars in our table, our base table looks as follows:

We've added two Star items for the `alexdebrie/dynamodb-book` Repo. Notice how we're handling two different relationships in a single item collection. We place the parent entity—the Repo item, outlined in green—right in the middle of the collection. All the Issues (outlined in red) are ordered *before* the Repo item and all the Star items (outlined in blue) are ordered *after* the Repo item.

If I want to get the Repo and all Stars, I would use the following Query:

```
result = dynamodb.query(
    TableName='GitHubTable',
    KeyConditionExpression="#pk = :pk AND #sk >= :sk",
    ExpressionAttributeNames={
        "#pk": "PK",
        "#sk": "SK"
    },
    ExpressionAttributeValues={
        ":pk": { "S": "REPO#alexdebrie#dynamodb-book" },
        ":sk": { "S": "REPO#alexdebrie#dynamodb-book" }
    }
)
```

Notice that my key condition expression includes values for both the partition key and the sort key. I need to add the sort key condition to make sure I don't get any Issues in my results. I'll handle that by starting right at the Repo item and scanning forward.

This also means I'll need to update my Query above for the Repo & Issues access pattern. It should have a sort key condition as well.

Handling counts for Stars and Forks

There's one last issue we need to cover before moving on from repositories. As shown above, GitHub displays the counts of both forks and stars on the repository page. In the example screenshot, there are over 5,700 stars and 1,100 forks for this repository, and there are repositories that are significantly more popular than that. The Bootstrap CSS library has over 138,000 stars and 68,000 forks!

We saw above that we'll be storing a Star and Fork item for each star and fork. But it would be inefficient to recount all those items each time someone loaded the page. Instead, we'll keep a running count on the Repo item for fast display.

When someone stars or forks a repo, we now need to do two things:

1. Create the Star or Fork/Repo item, ensuring it doesn't already

exist (we don't want to allow a user to star a repo multiple times), and

2. Increment the `StarCount` or `ForkCount` attribute on the Repo item.

Notably, the second part should not happen without the first. If a user tries to star a repo multiple times, we will reject the entire operation.

This is a great use for DynamoDB Transactions. With DynamoDB Transactions, you can perform multiple actions in a single request and ensure that all the actions will succeed or fail together. If one action fails, all other transactions fail.

The code to handle this will look something like the following:

```
result = dynamodb.transact_write_items(
    TransactItems=[
        {
            "Put": {
                "Item": {
                    "PK": { "S": "REPO#alexdebrie#dynamodb-book" },
                    "SK": { "S": "STAR#danny-developer" }
                    ...rest of attributes ...
                },
                "TableName": "GitHubModel",
                "ConditionExpression": "attribute_not_exists(PK)"
            }
        },
        {
            "Update": {
                "Key": {
                    "PK": { "S": "REPO#alexdebrie#dynamodb-book" },
                    "SK": { "S": "#REPO#alexdebrie#dynamodb-book" }
                },
                "TableName": "GitHubModel",
                "ConditionExpression": "attribute_exists(PK)"
                "UpdateExpression": "SET #count = #count + :incr",
                "ExpressionAttributeNames": {
                    "#count": "StarCount"
                },
                "ExpressionAttributeValues": {
                    ":incr": { "N": "1" }
                }
            }
        }
    ]
)
```

There is one write request to create the new Star item, and one write request to increment the `StarCount` attribute on the Repo item. If either part fails, we reject the whole operation and don't get an incorrect count.

Storing reference counts can save a lot of query time in DynamoDB. Using DynamoDB Transactions is a great way to ensure your reference counts stay accurate.

21.3.2. Modeling Comments and Reactions

Now that we've handled the repo entity and its related sub-entities, let's move on to another section of our data modeling. We'll work

on Comments and Reactions next.

As a refresher, users may leave comments in issues and in pull requests on GitHub. There is no threading of comments, and there's no need to fetch all comments by a particular user. All comments are viewed in the context of the issue or pull request to which they belong.

Additionally, comments may have reactions. Further, an issue or a pull request itself may have reactions as well.

Let's model the comments first, then circle back to reactions.

Comments for Issues & Pull Requests

Comments have a one-to-many relationship with their parent Issue or Pull Request. However, our API doesn't have a requirement of 'joining' our data by fetching the Issue and all Comments for an Issue. Rather, we only need to fetch all Comments for an issue.

Given that, we don't need to place Comments in the same item collection as their parent entity. We just need to ensure that all Comments for a particular target end up in the same item collection.

We'll still use the most common one-to-many relationship strategies—the primary key + Query API—but we won't worry about locating the parent item in it.

Let's use the following structure for IssueComments and PRComments:

IssueComments:

- **PK:** `ISSUECOMMENT#<Owner>#<RepoName>#<IssueNumber>`

- **SK:** `ISSUECOMMENT#<CommentId>`

PRComments:

- **PK:** `PRCOMMENT#<Owner>#<RepoName>#<PRNumber>`
- **SK:** `PRCOMMENT#<CommentId>`

Both types of Comments utilize a `CommentId`. We'll use KSUIDs just like we've used for other entities. Check out Chapter 14 for more on KSUIDs.

This strategy will ensure all Comments for an Issue or Pull Request will be in the same item collection, making it easy to fetch all Comments for a particular Issue or Pull Request.

Modeling Reactions

Now that we have Comments modeled, let's move on to Reactions. A GitHub user can react to an issue, a pull request, or a comment. There are eight types of reactions, and a user is allowed to react to each item once *for each reaction type*. Thus, a single user could react to an issue eight different times.

Reactions have similar properties to Stars and Forks. We want to be able to display a quick count of any particular reaction, but we also need to ensure that a user hasn't reacted to a particular object with the given reaction before.

To do this, we'll use a similar strategy as Stars and Forks with a slight twist. With the Stars and Forks, we only needed to keep track of one counter and prevent one duplicate for each entity. We need to keep track of eight here.

We could do this in two ways. First, we could use eight different

attributes on the relevant Issue, Pull Request, or Comment to track the eight different counts. Further, we could use an item with eight different attributes (or even eight different items!) in our table to ensure a user didn't have the same reaction to the same target multiple times.

The second approach is to use a complex attribute type like a map to handle our counts. We'll do that here.

First, on each Issue, Pull Request, and Comment item, we'll have a `ReactionCounts` attribute that stores the counts for the eight different reaction types. Each key in the map will be a reaction type, and the value will be the number of counts.

Then, we'll create a User Reaction Target item that will track a single user's reactions to a particular target (Issue, Pull Request, or Comment). There will be a `Reactions` attribute on the item that is of type string set to track the reactions that have been used.

Our Reaction item will be as follows:

Reaction:

- **PK:**
  `<TargetType>REACTION#<Owner>#<RepoName>#<TargetIdentifier>#<UserName>`
- **SK:**
  `<TargetType>REACTION#<Owner>#<RepoName>#<TargetIdentifier>#<UserName>`

This is a little confusing, so let's walk through it.

First, the `TargetType` refers to the item to which we're reacting. The potential values here are `ISSUE`, `PR`, `ISSUECOMMENT`, or `PRCOMMENT`.

Second, the `TargetIdentifier` property is how you uniquely identify the target. For an Issue or Pull Request, it will be the Issue

or Pull Request Number. For a Comment, it will be the Issue or Pull Request Number *plus* the Comment Id.

To handle reaction counts, we'll use a transaction just like we did with Stars and Forks.

Let's imagine a user adds a heart reaction to an issue in our repo. The transaction code would look as follows:

```
result = dynamodb.transact_write_items(
    TransactItems=[
        {
            "Update": {
                "Key": {
                    "PK": { "S": "ISSUEREACTION#alexdebrie#dynamodb-
book#4#happy-harry" },
                    "SK": { "S": "ISSUEREACTION#alexdebrie#dynamodb-
book#4#happy-harry" },
                },
                "TableName": "GitHubModel",
                "UpdateExpression": "ADD #reactions :reaction",
                "ConditionExpression": "attribute_not_exists(#reactions) OR
NOT contains(#reactions, :reaction)",
                "ExpressionAttributeNames": {
                    "#reactions": "Reactions"
                }
                "ExpressionAttributeValues: {
                    ":reaction": { "SS": [ "Heart" ]}
                }
            }
        },
        {
            "Update": {
                "Key": {
                    "PK": { "S": "REPO#alexdebrie#dynamodb-book" },
                    "SK": { "S": "#ISSUE#4" }
                },
                "TableName": "GitHubModel",
                "UpdateExpression": "SET #reactions.#reaction =
#reactions.#reaction + :incr",
                "ExpressionAttributeNames": {
                    "#reactions": "Reaction",
                    "#reaction": "Heart"
                },
                "ExpressionAttributeValues": {
                    ":incr": { "N": "1" }
                }
            }
        }
    ]
)
```

In the first action, we're adding `Heart` to the `Reactions` string set attribute as long as `Heart` doesn't already exist in the set. In the second action, we're incrementing the count of `Heart` reactions on our Issue item.

The finish line is in sight! So far, we've untangled the mess of relationships around Repos. Now we just need to handle Users and

Organizations.

Before we move on to Users and Organizations, let's update our entity chart:

Entity	PK	SK
Repo	`REPO#<Owner>#<RepoName>`	`REPO#<Owner>#<RepoName>`
Issue	`REPO#<Owner>#<RepoName>`	`ISSUE#<ZeroPaddedIssueNumber>`
Pull Request	`PR#<Owner>#<RepoName>#<ZeroPaddedPRNumber>`	`PR#<Owner>#<RepoName>#<ZeroPaddedPRNumber>`
IssueComment	`ISSUECOMMENT#<Owner>#<RepoName>#<IssueNumber>`	`ISSUECOMMENT#<CommentId>`
PRComment	`PRCOMMENT#<Owner>#<RepoName>#<PullRequestNumber>`	`PRCOMMENT#<CommentId>`
Reaction	`<TargetType>REACTION#<Owner>#<RepoName>#<TargetIdentifier>#<UserName>`	`<TargetType>REACTION#<Owner>#<RepoName>#<TargetIdentifier>#<UserName>`
Fork		
User		
Organization		
Payment Plan		

Table 38. GitHub model entity chart

Additionally, we have the following entities and properties in the GSI1 index:

Entity	GSI1PK	GSI1SK
Repo	`REPO#<Owner>#<RepoName>`	`REPO#<Owner>#<RepoName>`
Pull Request	`PR#<Owner>#<RepoName>`	`PR#<ZeroPaddedPRNumber>`

Table 39. GitHub model GSI1 entity chart

As well as the following entities and properties in our GSI2 index:

Entity	GSI2PK	GSI2SK
Repo	`REPO#<Owner>#<RepoName>`	`REPO#<Owner>#<RepoName>`

Entity	GSI2PK	GSI2SK
Fork	`REPO#<OriginalOwner>#<RepoName>`	`FORK#<Owner>`

Table 40. GitHub model GSI2 entity chart

21.3.3. Modeling Users and Organizations

We've untangled the nest around Repos. Let's finish up with Users and Organizations.

As a review, a User represents an individual user in GitHub. This generally maps to a human user, but it's loose—one human may have multiple GitHub accounts, or someone may make a GitHub account to be used as a bot. It's easiest and still accurate to think of it as one human.

An Organization is a shared account for a number of Users. An Organization will often be created by a company or a team at a company so that repos and other entities aren't tied to a specific User at the company.

There is a many-to-many relationship between users and organizations. A user may belong to many organizations, and an organization may have multiple users as members.

We'll handle Users and Organizations modeling in three parts. First, we'll add the Users and Organizations entities to our table and model the many-to-many relationship between Users and Organizations. Second, we'll add Payment Plans to our Users and Organizations. Finally, we'll update our Repo items so that we can handle the access patterns of "Fetch all Repos for an Account (whether User or Organization)".

User and Organization items

Let's start with the User and Organization entities. With a many-to-many relationship, the tricky part is how to query *both* sides of the relationship with fresh data.

One thing to note is that the relationships between users and organizations are pretty shallow. We don't have use cases where we need to fetch a User and detailed information about all Organizations to which the user belongs. In fact, we usually just need a basic list of the Organizations for a User, as shown in the GitHub UI below:
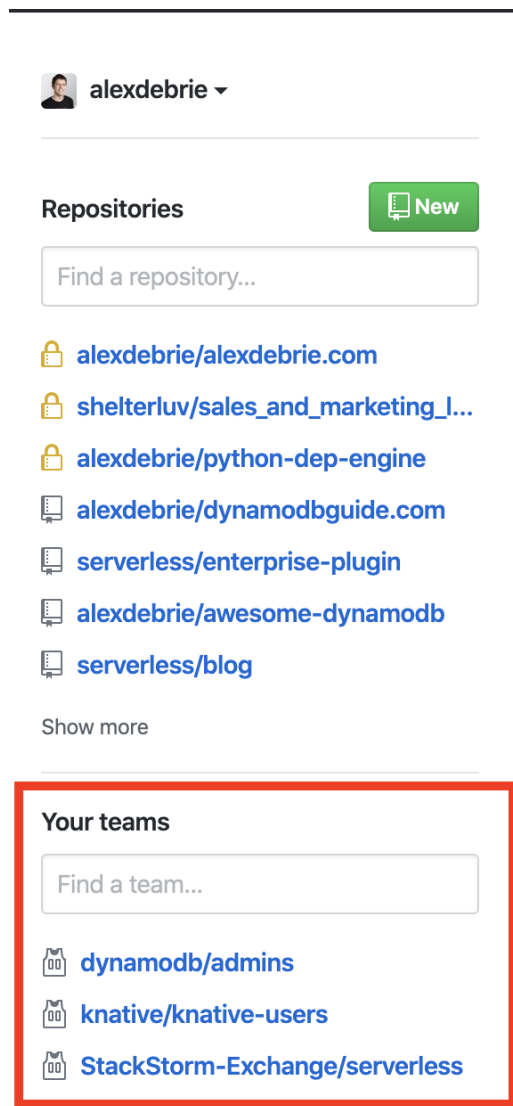
This gives us more flexibility to model the many-to-many relationship. We'll handle this in two similar but slightly different ways.

First, when modeling the organizations to which a user belongs, we'll use a complex attribute type on the User item. This item will be a map where the key is the Organization name and the value is the User's role within the Organization.

This is the "shallow duplication" strategy from Chapter 12. This strategy works because we can reasonably limit the number of Organizations to which a single User can belong. It wouldn't burden many Users to say they could only belong to, say, 40 Organizations.

Now that we've handled one side of the many-to-many relationship, it's much easier to handle the other side. We'll use a Membership tracking item to store information about a User belonging to an Organization.

It's less reasonable to limit the number of Users that belong to an Organization, as you could have an Organization such as a company or a very large open-source project that has hundreds or thousands of users. Because GitHub's pricing model is based on the number of users in an organization, we don't want to arbitrarily limit the number of users in an organization just to make it easier for our data model. As such, we don't want to use the denormalization strategy for Memberships.

Thus, we'll be adding the following three entities to our table:

User:

- **PK:** `ACCOUNT#<AccountName>`
- **SK:** `ACCOUNT#<AccountName>`

Organization:

- **PK:** `ACCOUNT#<AccountName>`
- **SK:** `ACCOUNT#<AccountName>`

Membership:

- **PK:** `ACCOUNT#<OrganizationName>`
- **SK:** `MEMBERSHIP#<UserName>`

Note that both Users and Organizations have the same primary key structure. In fact, you can't tell by the primary key alone whether you're working with a User or an Organization.

This is because Users and Organizations are fairly interchangeable within GitHub. Most importantly, they both compete for names within the same namespace, as there can't be a User and an Organization with the same name. If you want to ensure uniqueness on a property across your table, you need to build that into your primary key.

Further, both Users and Organizations own Repos and will have access patterns around those. The only real difference is that an Organization's item collection can also include Membership items.

To distinguish between Users and Organizations, we'll add an attribute called `Type` on the User and Organization items. This will indicate whether an item is a User or an Organization. Because this attribute is not in the primary key, we won't be able to query on it directly. However, we don't have any access patterns that say "Fetch all Users where X" or "Fetch all Orgs with Y".

Our Users, Organizations, and Membership items would look as follows:

Primary key		Attributes			
Partition key: PK	Sort key: SK				
ACCOUNT#alexdebrie	ACCOUNT#alexdebrie	Type	Username	CreatedAt	Organizations
		User	alexdebrie	2019-10-04 09:54:04	{"megacorp":{"S":"Member"}}
ACCOUNT#danny-developer	ACCOUNT#danny-developer	Type	Username	CreatedAt	
		User	danny-developer	2019-10-06 08:24:30	
ACCOUNT#megacorp	ACCOUNT#megacorp	Type	OrganizationName	CreatedAt	
		Organization	megacorp	2019-10-08 23:14:28	
	MEMBERSHIP#alexdebrie	Username	CreatedAt	Role	
		alexdebrie	2019-10-10 00:14:34	Member	

Notice that there are two Users, each of which have their own item collection. The top User, `alexdebrie`, has an `Organizations` attribute which contains details about the organization to which the User belongs.

There is also an Organization, `megacorp`. In the `megacorp` item collection, there is a Membership item that tracks the membership of `alexdebrie` in `megacorp`.

With this setup, we can handle both the "Get User" and "Get Organization" access patterns using simple `GetItem` lookups. Additionally, we can do the "Get Users in Organization" access pattern by using a Query operation on the item collection for an Organization.

Payment plans for users and organizations

Next, let's model out payment plans for users and organizations. As a reminder, both users and organizations can sign up for payment plans with GitHub that give them access to premium features. There is a one-to-one relationship between users and organizations and a payment plan.

When you have a one-to-one relationship in DynamoDB, you almost always include that information directly on the parent item. The only time you wouldn't is if the related item were quite large

and you wanted to avoid accessing it each time you grab the parent item.

That's not the case here. Accordingly, we'll just include payment plan details as a complex attribute on the User or Organization item itself.

Our items from above would be updated as follows:

Primary key		Attributes				
Partition key: PK	Sort key: SK					
ACCOUNT#alexdebrie	ACCOUNT#alexdebrie	Type	Username	CreatedAt	Organizations	PaymentPlan
		User	alexdebrie	2019-10-04 09:54:04	{"megacorp":{"S":"Member"}}	{"planType": {"S":"Pro"},"planStartDate": {"S":"2019-10-05 11:22:0"}}
ACCOUNT#danny-developer	ACCOUNT#danny-developer	Type	Username	CreatedAt		
		User	danny-developer	2019-10-06 08:24:30		
ACCOUNT#megacorp	ACCOUNT#megacorp	Type	Organization Name	CreatedAt	PaymentPlan	
		Organization	megacorp	2019-10-08 23:14:28	{"planType": {"S":"Enterprise"},"planStartDate": {"S":"2019-10-22 10:15:36"}}	
	MEMBERSHIP#alexdebrie	Username	CreatedAt	Role		
		alexdebrie	2019-10-10 00:14:34	Member		

Note that the `alexdebrie` User and the `megacorp` Organization both have `PaymentPlan` attributes where payment information is listed in a map.

Repos and Accounts

The final access pattern to handle is fetching all Repos for a given account, whether a User or an Organization. This is a one-to-many relationship similar to the ones we've used in other areas.

Because the number of Repos for a User or Organization is unbounded, we can't use the denormalized strategy with a complex

attribute type. We'll need to use the Query-based strategies using either the primary key or a secondary index.

Our Repo item is already pretty busy on the primary key, as it's handling relationships for both Issues and Stars. As such, we'll need to use a secondary index. We have two secondary indexes already, but the Repo item is already using both of them for other relationships. It's using GSI1 to handle Repo + Pull Requests, and it's using GSI2 to handle the hierarchical Fork relationship. Accordingly, we'll need to add a third secondary index.

When fetching repositories for an account, we want to show them in order of most-recently updated. This means we'll need to use the `UpdatedAt` value in our sort key for the Repo items.

Let's use the following pattern for GSI3:

Users & Organizations:

- **GSI3PK:** `ACCOUNT#<AccountName>`
- **GSI3SK:** `ACCOUNT#<AccountName>`

Repos:

- **GSI3PK:** `ACCOUNT#<AccountName>`
- **GSI3SK:** `+#<UpdatedAt>`

Our GSI3 will look as follows:

Primary key		Attributes		
Partition key: GSI3PK	Sort key: GSI3SK			
ACCOUNT#alexdebrie	#2019-10-11 14:11:17	RepoOwner	RepoName	
		alexdebrie	dynamodb-book	
	#2019-10-30 06:38:40	RepoOwner	RepoName	
		alexdebrie	graphql-demo	
	ACCOUNT#alexdebrie	Type	Username	CreatedAt
		User	alexdebrie	2019-10-04 09:54:04
ACCOUNT#megacorp	#2019-10-22 09:06:52	RepoOwner	RepoName	
		megacorp	mega-repo	
	ACCOUNT#megacorp	Type	OrganizationName	CreatedAt
		Organization	megacorp	2019-10-08 23:14:28

In this secondary index, we have two item collections—one for the `alexdebrie` User account and one for the `megacorp` Organization account. Each collection has the Repos that belong to those accounts sorted in order of the time in which they were last updated.

21.4. Conclusion

We did it! This was a beefy data model, but we managed to get it.

In the next chapter, we're going to tackle the dreaded migration. We'll see how to add new patterns and change mistakes in this data model.

Migrations seem scary with DynamoDB but, as you'll see, there's not that much to worry about. There are familiar patterns to follow that make them doable. Check it out!

Final entity charts:

Let's take a look at our final entity chart:

Entity	PK	SK
Repo	REPO#<Owner>#<RepoName>	REPO#<Owner>#<RepoName>

Entity	PK	SK
Issue	`REPO#<Owner>#<RepoName>`	`ISSUE#<ZeroPaddedIssueNumber>`
Pull Request	`PR#<Owner>#<RepoName>#<ZeroPaddedPRNumber>`	`PR#<Owner>#<RepoName>#<ZeroPaddedPRNumber>`
IssueComment	`ISSUECOMMENT#<Owner>#<RepoName>#<IssueNumber>`	`ISSUECOMMENT#<CommentId>`
PRComment	`PRCOMMENT#<Owner>#<RepoName>#<PullRequestNumber>`	`PRCOMMENT#<CommentId>`
Reaction	`<TargetType>REACTION#<Owner>#<RepoName>#<TargetIdentifier>#<UserName>`	`<TargetType>REACTION#<Owner>#<RepoName>#<TargetIdentifier>#<UserName>`
User	`ACCOUNT#<Username>`	`ACCOUNT#<Username>`
Organization	`ACCOUNT#<OrgName>`	`ACCOUNT#<OrgName>`
Payment Plan	N/A	N/A
Membership	`ACCOUNT#<OrgName>`	`MEMBERSHIP#<Username>`

Table 41. GitHub model entity chart

Additionally, we have the following entities and properties in the GSI1 index:

Entity	GSI1PK	GSI1SK
Repo	`REPO#<Owner>#<RepoName>`	`REPO#<Owner>#<RepoName>`
Pull Request	`PR#<Owner>#<RepoName>`	`PR#<ZeroPaddedPRNumber>`

Table 42. GitHub model GSI1 entity chart

As well as the following entities and properties in our GSI2 index:

Entity	GSI2PK	GSI2SK
Repo	`REPO#<Owner>#<RepoName>`	`REPO#<Owner>#<RepoName>`
Fork	`REPO#<OriginalOwner>#<RepoName>`	`FORK#<Owner>`

Table 43. GitHub model GSI2 entity chart

And finally GSI3:

413

Entity	GSI3PK	GSI3SK
Repo	`ACCOUNT#<AccountName>`	`#<UpdatedAt>`
User	`ACCOUNT#<Username>`	`ACCOUNT#<Username>`
Organization	`ACCOUNT#<OrgName>`	`ACCOUNT#<OrgName>`

Table 44. GitHub model GSI3 entity chart

Chapter 22. Handling Migrations in our GitHub example

In our last example, we walked through a complicated data model to re-create the GitHub metadata API. If you're like me, you felt pretty good about how it all turned out.

Now let's imagine it's a year later. We have new access patterns to deal with. We realized some of our initial assumptions were incorrect. How will we modify our table design to handle this?

One of the biggest worries I hear around DynamoDB is the alleged difficulty of migrations as your access patterns change. In this example, we'll see that it's not that bad. We'll modify our GitHub metadata example to accomodate new patterns. This example will implement some of the strategies mentioned in Chapter 15.

Let's get started!

22.1. Introduction

To start off, let's see the new patterns we need to handle. We're going to make four changes in this chapter. Three of these are brand new entity types and one is a change in how we model an existing pattern based on some incorrect assumptions.

The four changes are detailed below.

22.1.1. Adding a Code of Conduct

Over the last few years, it has become more popular to include a Code of Conduct in your GitHub repository. These Codes of Conduct describe how one should act when using or contributing to a particular piece of open source software. The goal is to make a more welcoming and inclusive atmosphere for all developers.

GitHub has provided first-class support for Codes of Conduct by including information about a repository's Code of Conduct, if present, when retrieving the repository via GitHub's API. We need to add support for this in our data model.

22.1.2. Adding GitHub Gists

GitHub Gists are small snippets of code that can be easily shared with people. They are more lightweight than GitHub Repos, though they are functionally similar in a lot of ways. They have full source control and support for multiple files, but they don't have things like issues and pull requests.

An image of the GitHub Gists page for me is shown below:
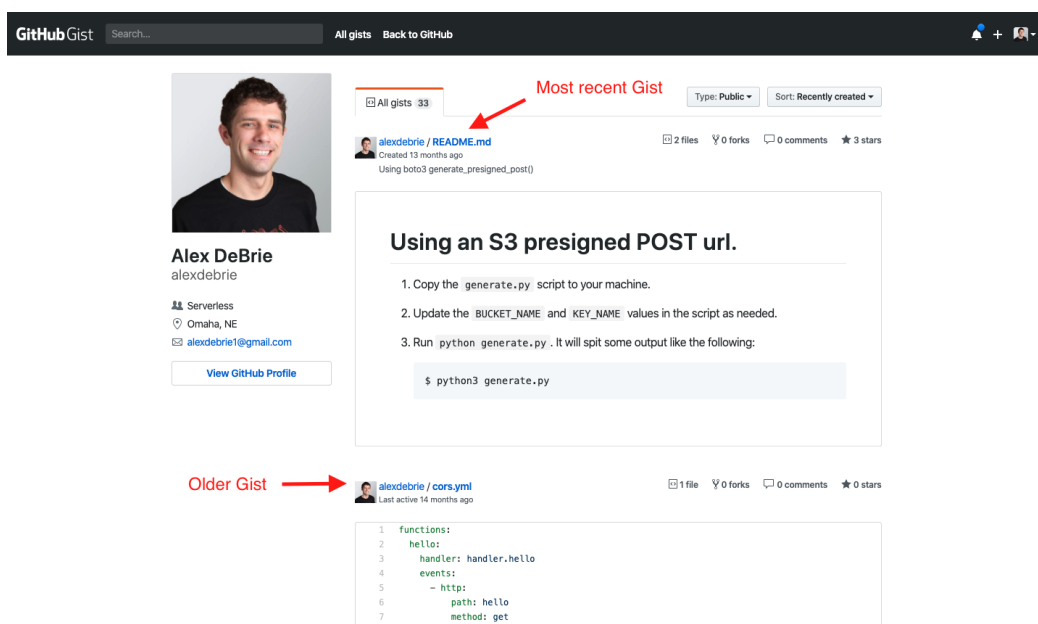
On this screen, notice that it shows my most recent gists according to the date they were created.

22.1.3. Adding GitHub Apps

The third change we're going to make is to add GitHub Apps. GitHub Apps are bits of functionality that a user or organization can create to integrate with your GitHub workflow. Many developer tools companies use GitHub Apps to integrate their tools with their users' GitHub workflows.

For example, I have integrated the Netlify GitHub App with my personal blog and with DynamoDBGuide.com. This integration creates preview branches when I open a pull request and deploys new versions of the site when I merge to master. There are similar integrations with CI/CD systems such as CircleCI or TravisCI or with ticket tracking systems like Jira.

There are two main access patterns for GitHub Apps. First, a user or an organization is the one that will create a GitHub App. In addition to the create & update patterns around that, we will need "Fetch account (User or Organization) and all GitHub Apps for account" to handle the following screen:
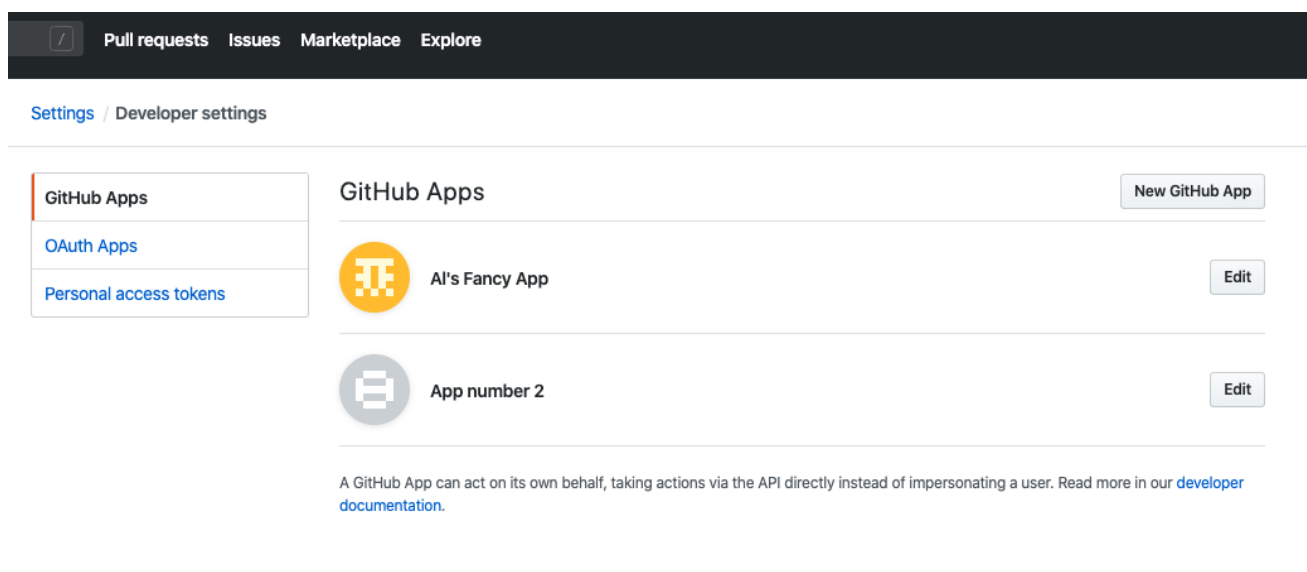
This screen shows all of the GitHub Apps that I have created. There is a similar screen for Organizations.

Second, users can install GitHub Apps into a particular repository. When a GitHub App is added to a repository, this is called an "installation". We will have an access pattern that handles "Fetch Repo and GitHub App Installations for Repo" to handle the following screen:
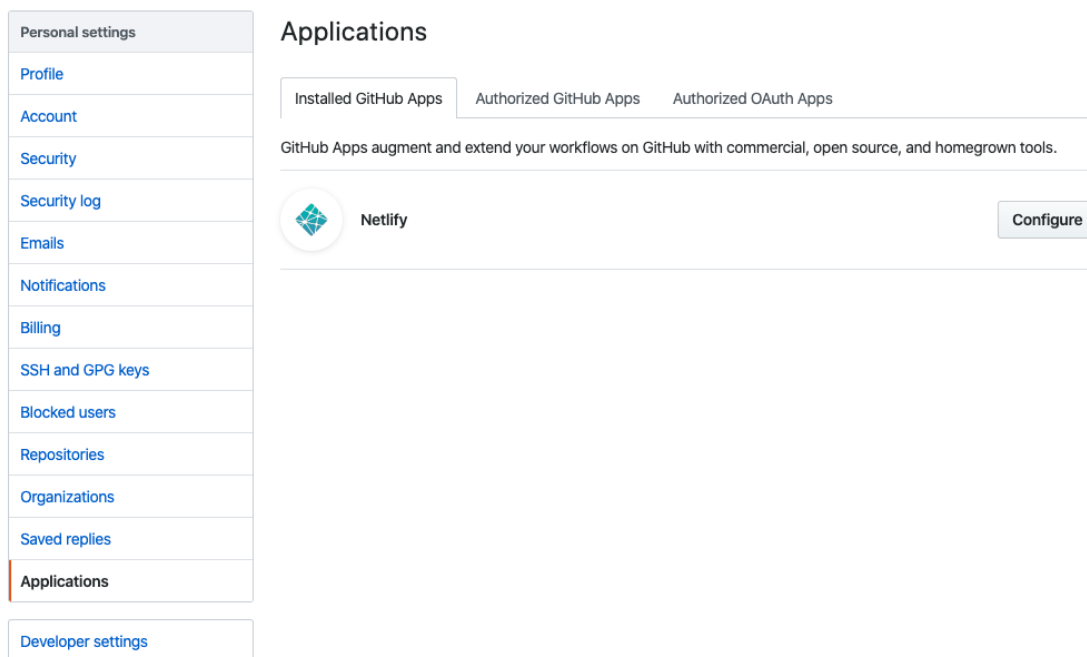
Note that there will usually be a limited (<10) number of installations for a given repo, though this is not an actual limit. Because there is usually a small number of installations, there is no particular ordering requirement for the installations that are returned.

22.1.4. Refactoring Open / Closed access of Issues and Pull Requests

When modeling Issues and Pull Requests in the initial data model, we noted that the access patterns for these both contained a status

filter. When retrieving Issues and Pull Requests, you either retrieve the "Open" ones or the "Closed" ones.

We mentioned that we could handle this using a filter expression because there were only two options. We figured we would have a pretty high hit rate when querying and thus wouldn't have a lot of throwaway data with our filter expression.

As we put this into production, we ran into two problems:

1. Some repositories have poor maintenance, meaning most of the recent Issues or Pull Requests are "Open". Accordingly, we have to query pretty deep into our Repo to find the most recent items with a status of "Closed".

2. We're consistently over-fetching to ensure we return 25 items. Because we don't know how many will be filtered out with the filter expression, we grab 75 Issues or Pull Requests on our first request. Usually, this means we grabbed far too many items to satisfy our requirement. However, in some instances, 75 items was not enough, so we need to make a follow-up query to retrieve additional items. This thrashing leads to less efficient queries.

Given these problems, we're going to switch our Issue & Pull Request access patterns such that we can fetch open or closed items via the primary key, rather than using a filter expression.

22.2. ERD & Access Patterns

Now that we know our four changes, let's follow our usual process. First, we need to create our ERD.

The updated ERD is as follows. All newly-added elements are in

red.
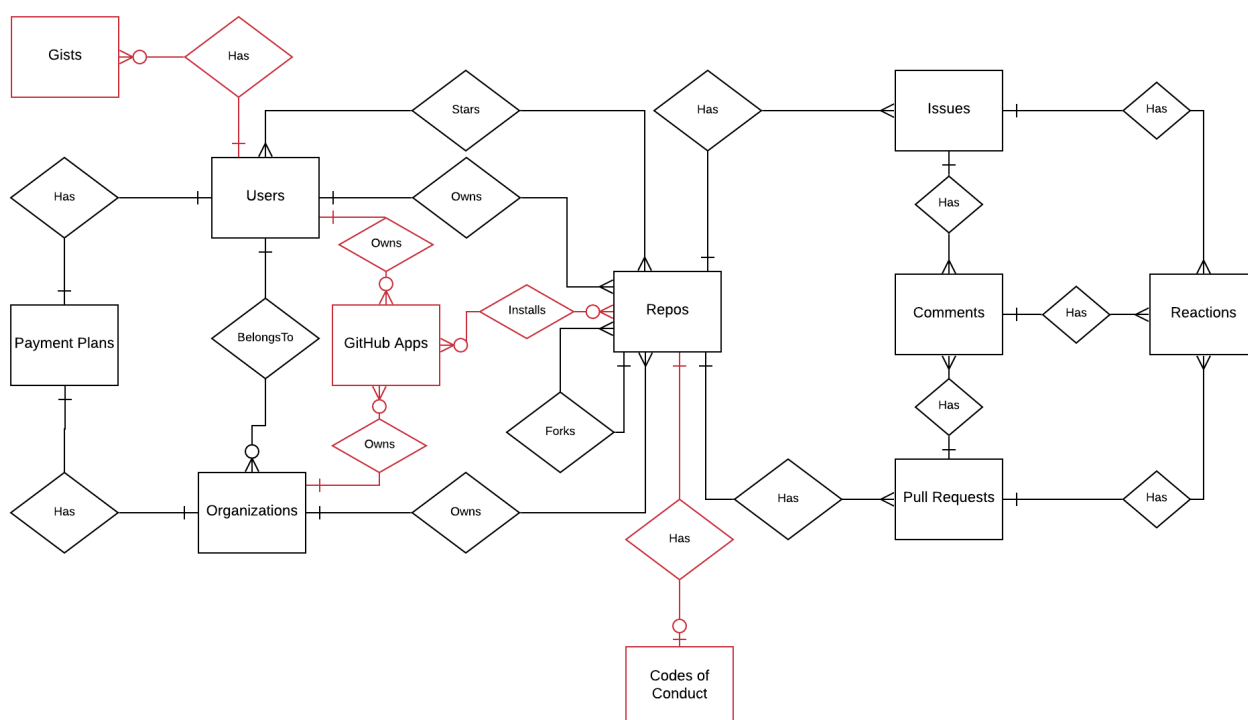
The major changes are:

1. **Codes of Conduct.** There is now a Code of Conduct entity, and there is a one-to-one relationship between Repos and Codes of Conduct.

2. **Gists.** There is a newly-added Gist entity in the top-left corner. There is a one-to-many relationship between Users and Gists, as Users can create one or more Gists. Note that while Organization and Users are interchangeable in many scenarios, an Organization may not own Gists.

3. **GitHub Apps.** There is now a GitHub App entity shown in the middle-left of the ERD. There is a one-to-many relationship between Users and GitHub Apps as well as a one-to-many relationship between Organizations and GitHub Apps, since both Users and Organizations may own Apps.

    Further, there is a many-to-many relationship between Repos and GitHub Apps, since a GitHub App can be installed in

multiple repositories and a single repository may install multiple GitHub Apps.

Note that we don't need to make any changes to the ERD for updated approach for issues and pull requests as nothing changed about the data model, just about how we want to handle it.

We also have the following new access patterns:

- Add Code of Conduct
- Create Gists
- View Gists for User
- Create GitHub App
- Install GitHub App to Repo
- View GitHub App Installations
- View GitHub App for Repo
- Fetch Open / Closed Issues
- Fetch Open / Closed Pull Requests

Note that the last two aren't *new* patterns, but they are ones we need to handle.

With that in mind, let's get started!

22.3. Data modeling walkthrough

We already have our primary key structure defined from our original data model, so we don't need to start there. We just need to decide where to start.

There's no real rhyme or reason to attacking these new access

patterns. I generally like to knock out the easiest ones first and then work on the harder ones. When modeling in DynamoDB, I rank them in the following order of difficulty:

1. Adding a new attribute
2. Adding a new entity with no relationship
3. Adding a new entity & relationship into an existig item collection
4. Adding a new entity & relationship into a new item collection
5. Migrating existing patterns

Adding a Code of Conduct to a Repo (one-to-one relationship) is basically like adding a new attribute, so we'll handle that first.

Adding Gists is adding a one-to-many relationship with Users, so we'll take that second.

Adding GitHub Apps involves both one-to-many relationships (with Users and Organizations) and a many-to-many relationship with Repos, so we'll handle that third.

Finally, our reworking of Issues and Pull Requests is the most difficult, so we'll handle it last.

22.3.1. Modeling Codes of Conduct

The first pattern we'll handle is adding a Code of Conduct. As discussed above, a Code of Conduct has a one-to-one relationship with a particular repository.

In the GitHub API response, there are two attributes included for a repository's Code of Conduct:

1. The name of the Code of Conduct (`name` property)
2. The URL of the Code of Conduct (`url` property)

A one-to-one relationship in DynamoDB is almost always modeled as part of the same item. We're big fans of denormalization with DynamoDB, and a one-to-one relationship isn't even denormalization! The only time you may want to split this out is if the related item is quite large and has different access patterns than the underlying object. You may be able to save read capacity by only fetching the related item when needed.

In this case, the `name` and `url` properties are likely to be quite small. As such, we'll just add the Code of Conduct entity as an attribute to our Repo item.

Our Repo items with the new `CodeOfConduct` attribute looks as follows:

We are using a `CodeOfConduct` attribute of type map to handle this one-to-one relationship.

The great thing about adding this new entity is that we can be lazy about it. This isn't used for indexing and querying in DynamoDB, and the `CodeOfConduct` attribute is not a required property. Accordingly, We don't need to run a big ETL job to modify all of our existing Repo items. As users take advantage of this feature by registering a Codes of Conduct in their repositories, we can add the `CodeOfConduct` attribute to the corresponding Repo item.

22.3.2. Modeling Gists

The second migration we want to handle is adding GitHub Gists.

The create and update functionality around a new item like Gists should be pretty straightforward. We just need to determine the primary key pattern for a Gist.

The tricky part with Gists is that we have an access pattern of "Fetch User and Gists for User". This means we'll need to ensure Gists are in the same item collection as a User entity.

Let's take a look at the existing User entity in our base table:

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
ACCOUNT#alexdebrie	ACCOUNT#alexdebrie	Type	Username	CreatedAt	Organizations
		User	alexdebrie	2019-10-04 09:54:04	{"megacorp":{"S":"Member"}}
ACCOUNT#danny-developer	ACCOUNT#danny-developer	Type	Username	CreatedAt	
		User	danny-developer	2019-10-06 08:24:30	
ACCOUNT#megacorp	ACCOUNT#megacorp	Type	OrganizationName	CreatedAt	
		Organization	megacorp	2019-10-08 23:14:28	
	MEMBERSHIP#alexdebrie	Username	CreatedAt	Role	
		alexdebrie	2019-10-10 00:14:34	Member	

If we look at our User entity, there are no other entities in its item collection in the main table at the moment. That means we can reuse this item collection without having to decorate our User items to create a new item collection in a secondary index.

When determining how to model this, we have two basic needs:

1. Gists are identifiable by a unique identifier
2. We want to be able to find these Gists in reverse chronological order for a given User.

To do this, we'll use our old friend, the KSUID. Remember, the KSUID is like a UUID but with a timestamp prefix to give it chronological sorting. We've used KSUIDs in a number of previous examples, and you can read more about it in Chapter 14.

When adding Gists to our table, we'll use the following pattern:

Gists:

- **PK:** `ACCOUNT#<Username>`
- **SK:** `#GIST#<KSUID>`

Here's a look at our table with a few Gists added to the existing Users:

| Primary key | | Attributes | | |
Partition key: PK	Sort key: SK			
ACCOUNT#alexdebrie	#GIST#1YK484E7V3xiMT1NMrfWlFaLPFU	KSUID	DateCreated	GistTitle
		1YK484E7V3xiMT1NMrfWlFaLPFU	2020-02-26T00:43:08.000	sls.yaml example
	#GIST#1YK4DXEfgPZrUjWxBRkPmSgFQnf	KSUID	DateCreated	GistTitle
		1YK4DXEfgPZrUjWxBRkPmSgFQnf	2020-02-26T00:43:52.000	graphql-test
	ACCOUNT#alexdebrie	Type	Username	CreatedAt
		User	alexdebrie	2019-10-04 09:54:04
ACCOUNT#danny-developer	ACCOUNT#danny-developer	Type	Username	CreatedAt
		User	danny-developer	2019-10-06 08:24:30

The URL to fetch or update a specific Gist will be `https://gist.github.com/<username>/<gistId>`. This URL contains both the username and the KSUID of the Gist, so we can easily find it in our table.

Further, we can use the Query API to retrieve a User and the User's most recent Gist items. That Query would be written as follows:

```
result = dynamodb.query(
    TableName='GitHubTable',
    KeyConditionExpression="#pk = :pk",
    ExpressionAttributeNames={
        "#pk": "PK",
    },
    ExpressionAttributeValues={
        ":pk": { "S": "ACCOUNT#alexdebrie" },
    },
    ScanIndexForward=False
)
```

Note that we're using the `ScanIndexForward=False` parameter to indicate we want to read our items in descending order, which will

return our User and the most recent Gists.

22.3.3. Modeling GitHub Apps

With the two easier ones out of the way, it's time to move to a more difficult example—GitHub Apps.

GitHub Apps have a few things going on. First, there is a one-to-many relationship between GitHub Apps and its owner (either a User or an Organization). Second, a GitHub App can be installed into multiple repositories, meaning there is a many-to-many relationship between GitHub Apps and Repo items.

Let's start with the one-to-many relationship first, then circle back to the many-to-many relationships.

One-to-many relationships between Accounts and GitHub Apps

For our one-to-many relationship between Accounts (Users & Orgs) and GitHub Apps, we have an access pattern that wants to fetch an Account plus all GitHub apps owned by the Account. To do this, we'll need to ensure that GitHub Apps are in the same item collection as their owner.

Let's first take a look at what is in the Users and Orgs item collections in our base table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
ACCOUNT#alexdebrie	#GIST#1YK484E7V3xiMT1NMrfWlFaLPFU	KSUID	DateCreated	GistTitle
		1YK484E7V3xiMT1NMrfWlFaLPFU	2020-02-26T00:43:08.000	sls.yaml example
	#GIST#1YK4DXEfgPZrUjWxBRkPmSgFQnf *(Gist items)*	KSUID	DateCreated	GistTitle
		1YK4DXEfgPZrUjWxBRkPmSgFQnf	2020-02-26T00:43:52.000	graphql-test
	ACCOUNT#alexdebrie *(User item)*	Type	Username	CreatedAt
		User	alexdebrie	2019-10-04 09:54:04
ACCOUNT#danny-developer	ACCOUNT#danny-developer	Type	Username	CreatedAt
		User	danny-developer	2019-10-06 08:24:30
ACCOUNT#megacorp	ACCOUNT#megacorp *(Org item)*	Type	OrganizationName	CreatedAt
		Organization	megacorp	2019-10-08 23:14:28
	MEMBERSHIP#alexdebrie *(Membership item)*	Username	CreatedAt	Role
		alexdebrie	2019-10-10 00:14:34	Member

Notice that our User item collections contain Gist items which are located *before* the User item, which gives us reverse-chronological ordering. Our Organization item collections, on the other hand, have Membership items *after* the Organization item because we're fetching user memberships in an organization in alphabetical order.

This means we can't easily place the GitHub App items in the relevant item collections in the base table. We could place it *after* the User item in the User item collections but *before* the Organization item in Organization item collections. Howver, our client won't know at query time whether they're dealing with a User or an Organization and thus which way to scan the index to find the most recent Gists.

Instead, we can handle this in a secondary index. Currently, our User and Org items are using GSI3 to handle Account + Repo access patterns, but they don't have any access patterns using GSI1 or GSI2. We'll add this pattern in GSI1.

To handle our patterns, let's create the GitHub App entity:

GitHub App:

- **PK:** `APP#<AccountName>#<AppName>`
- **SK:** `APP#<AccountName>#<AppName>`
- **GSI1PK:** `ACCOUNT#<AccountName>`
- **GSI1SK:** `APP#<AppName>`

Second, we'll add the following properties to our Users and Organizations:

Users and Organizations:

- **GSI1PK:** `ACCOUNT#<AccountName>`
- **GSI1SK:** `ACCOUNT#<AccountName>`

With these updates in place, our base table would look as follows:

Notice that we've added two GitHub App items into our base table (outlined in red). We've also added `GSI1PK` and `GSI1SK` attributes to our existing User and Org items (outlined in blue).

If we look at GSI1, we will see the following:

Primary key		Attributes					
Partition key: GSI1PK	**Sort key: GSI1SK**						
ACCOUNT#alexdebrie	ACCOUNT#alexdebrie	PK	SK	Type	Username	CreatedAt	
		ACCOUNT#alexdebrie	ACCOUNT#alexdebrie	User	alexdebrie	2019-10-04 09:54:04	
	APP#cicd	PK	SK	AccountName	AppName	Description	
		APP#alexdebrie#cicd	APP#alexdebrie#cicd	alexdebrie	CICD	This app gives you magic CI/CD!	
	APP#githubanalytics	PK	SK	AccountName	AppName	Description	
		APP#alexdebrie#githubanalytics	APP#alexdebrie#githubanalytics	alexdebrie	GitHub Analytics	Magic analytics for your GitHub repo	
ACCOUNT#danny-developer	ACCOUNT#danny-developer	PK	SK	Type	Username	CreatedAt	
		ACCOUNT#danny-developer	ACCOUNT#danny-developer	User	danny-developer	2019-10-06 08:24:30	
ACCOUNT#megacorp	ACCOUNT#megacorp	PK	SK	Type	OrganizationName	CreatedAt	
		ACCOUNT#megacorp	ACCOUNT#megacorp	Organization	megacorp	2019-10-08 23:14:28	

Notice the three items outlined in red. We have created an item collection in GSI1 that includes both an account (whether a User item or an Org item) as well as all the GitHub Apps that belong to that account.

Before we move on to the many-to-many relationship with GitHub Apps, I want to discuss one more thing. We mentioned above that we'll add two new attributes (GSI1PK and GSI1SK) to our existing User and Org items. I don't want to hand-wave past that, because it's important. *How* do we add these attributes?

On one hand, we could use the lazy method that we used above with Codes of Conduct. Because a GitHub App is a new type of entity, we would only need to add these attributes to Users or Orgs that create a GitHub App. Whenever adding a GitHub App, we could ensure that the adding User or Org item has the relevent attributes.

If we don't want to go the lazy approach, we can decorate our existing items with new attributes. To do this, we would use a one-time process to update our existing items.

That process would:

1. Scan our entire table;
2. Find the relevent items to be updated; and
3. Run an update operation to add the new properties.

A simplified version of this script would be as follows:

```python
last_evaluated = ''

while True:
    params = {
        "TableName": "GithubModel",
        "FilterExpression": "#type IN (:user, :org)",
        "ExpressionAttributeNames": {
            "#type": "Type"
        },
        "ExpressionAttributeValues": {
            ":user": { "S": "User" },
            ":org": { "S": "Organization" }
        }
    }
    if last_evaluated:
        params['ExclusiveStartKey'] = last_evaluated

    results = client.scan(**params)

    for item in results['Items']:
        client.update_item(
            TableName='GitHubModel',
            Key={
                'PK': item['PK'],
                'SK': item['SK']
            },
            UpdateExpression="SET #gsi1pk = :gsi1pk, #gsi1sk = :gsi1sk",
            ExpressionAttributeNames={
                '#gsi1pk': 'GSI1PK',
                '#gsi1sk': 'GSI1SK'
            }
            ExpressionAttributeValues={
                ':gsi1pk': item['PK'],
                ':gsi1sk': item['SK']
            }
        )

    if not results['LastEvaluatedKey']:
        break
    last_evaluated = results['LastEvaluatedKey']
```

This script is running a Scan API action against our DynamoDB table. It's using a filter expression to filter out any items whose Type

doesn't match `User` or `Organization` as we only care about those items.

As we receive items from our Scan result, we iterate over those items and make an UpdateItem API request to add the relevant properties to our existing items.

There's some additional work to handle the `LastEvaluatedKey` value that is received in a Scan response. This indicates whether we have additional items to scan or if we've reached the end of the table.

There are a few things you'd want to do to make this better, including using parallel scans, adding error handling, and updating multiple items in a BatchWriteItem request, but the general shape should work.

Many-to-many relationships between GitHub Apps and Repos

Now that we have our one-to-many relationships handled, let's move on to the many-to-many relationship between GitHub Apps and Repos. A GitHub App may be installed in multiple repositories, and a single repository may have multiple installed apps.

For both of these relationships, we want to fetch the root item and the related sub-item. That is, we want to fetch a GitHub App and all its installations, and we want to fetch a Repo and all installed apps. Further, there is no limit to the number of installations a single GitHub App may have.

Because of these patterns, we'll go with the adjacency list pattern for many-to-many relationships. We'll create a separate AppInstallation item that will track the installation of a particular

app into a particular repository. We will include this AppInstallation item in an item collection for both the parent app as well as the repository to which it was installed.

Because this AppInstallation item will need to be in two different item collections, we know that at least one secondary index will be involved. Let's see if we can handle the other access pattern in the base table.

Our Repo item is already pretty busy in the base table. We're storing Issues in there as well as Stars. Accordingly, we'll have to use a secondary index for that item collection.

Our GitHub App item, however, is not as busy. We just created this item in the last step, and there's nothing in its item collection yet. Let's put our AppInstallation items in there.

Further, our Repo item only has one relationship modeling in the GSI1 index. We can handle the other side of the many-to-many relationship there.

We want to ensure that an app is only installed into a particular repository one time. Thus, the primary key for our AppInstallation item should include both information about the App as well as information about the Repo.

Let's use the following pattern:

AppInstallation:

- **PK:** `APP#<AccountName>#<AppName>`
- **SK:** `REPO#<RepoOwner>#<RepoName>`
- **GSI1PK:** `REPO#<RepoOwner>#<RepoName>`
- **GSI1SK:** `REPOAPP#<AppOwner>#<AppName>`

Adding an AppInstallation item into our table will look as follows:

Partition key: PK	Sort key: SK	Attributes					
APP#alexdebrie#cicd	APP#alexdebrie#cicd	AccountName	AppName	Description	GSI1PK	GSI1SK	
		alexdebrie	CICD	This app gives you magic CI/CD!	ACCOUNT#alexdebrie	APP#cicd	
REPO#alexdebrie#graphql-demo	REPO#alexdebrie#graphql-demo	CreatedAt	RepoOwner	RepoName	GSI1PK	GSI1SK	
		2019-12-20 19:47:15	alexdebrie	graphql-demo	REPO#alexdebrie#graphql-demo	REPO#alexdebrie#graphql-demo	
REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book	CreatedAt	RepoOwner	RepoName	GSI1PK	GSI1SK	
		2019-10-07 23:38:31	alexdebrie	dynamodb-book	REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book	
APP#alexdebrie#githubanalytics	APP#alexdebrie#githubanalytics	AccountName	AppName	Description	GSI1PK	GSI1SK	
		alexdebrie	GitHub Analytics	Magic analytics for your GitHub repo	ACCOUNT#alexdebrie	APP#githubanalytics	
	REPO#alexdebrie#dynamodb-book	RepoOwner	RepoName	AccountName	AppName	GSI1PK	GSI1SK
		alexdebrie	dynamodb-book	alexdebrie	GitHub Analytics	REPO#alexdebrie#dynamodb-book	REPOAPP#alexdebrie#githubanalytics

Notice the AppInstallation item and GitHub App item outlined in red. They are in the same item collection. Because the two items are in the same item collection, we can use a Query API action to fetch a GitHub App and all of its installations.

Now if we look at the GSI1 index, we see the following:

Partition key: GSI1PK	Sort key: GSI1SK	Attributes					
ACCOUNT#alexdebrie	APP#cicd	PK	SK	AccountName	AppName	Description	
		APP#alexdebrie#cicd	APP#alexdebrie#cicd	alexdebrie	CICD	This app gives you magic CI/CD!	
	APP#githubanalytics	PK	SK	AccountName	AppName	Description	
		APP#alexdebrie#githubanalytics	APP#alexdebrie#githubanalytics	alexdebrie	GitHub Analytics	Magic analytics for your GitHub repo	
REPO#alexdebrie#graphql-demo	REPO#alexdebrie#graphql-demo	PK	SK	CreatedAt	RepoOwner	RepoName	
		REPO#alexdebrie#graphql-demo	REPO#alexdebrie#graphql-demo	2019-12-20 19:47:15	alexdebrie	graphql-demo	
REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book	PK	SK	CreatedAt	RepoOwner	RepoName	
		REPO#alexdebrie#dynamodb-book	REPO#alexdebrie#dynamodb-book	2019-10-07 23:38:31	alexdebrie	dynamodb-book	
	REPOAPP#alexdebrie#githubanalytics	PK	SK	RepoOwner	RepoName	AccountName	AppName
		APP#alexdebrie#githubanalytics	REPO#alexdebrie#dynamodb-book	alexdebrie	dynamodb-book	alexdebrie	GitHub Analytics

Now my Repo item and all the AppInstallations for the Repo are located in the same item collection (outlined in red).

The great part about handling these two many-to-many access patterns is that we didn't have to make any changes to existing items. We were able to reuse the item collection in the base table to co-locate GitHub Apps with their AppInstallations, and we were able to reuse an item collection in an existing secondary index to

co-locate Repos with their AppInstallations.

22.3.4. Refactoring Open & Closed Issues & Pull Requests

The last access pattern we want to handle is the refactoring of how we're handling the status filter on issues and pull requests. As a reminder, when fetching either issues or pull requests, the client asks for *either* open items or closed items.

Previously we were handling this using a filter expression, but this was causing extra reads on our table. Accordingly, we're going to build this filtering into the table design itself.

As I go through this example, I'm going to handle Issues first. However, the exact same pattern will apply to Pull Requests. We'll apply this at the end of this section.

For Issues, we have two access patterns:

- Fetch Repo and Most Recent *Open* Issues for Repo
- Fetch Repo and Most Recent *Closed* Issues for Repo

Notice that each access pattern has a double filter: a match on Status and an order by Issue Number.

A double filter is a great use case for the composite sort key pattern that we discussed in Chapter 13. We can have a sort key that combines the Status and the Issue Number using a pattern of `ISSUE#<Status>#<IssueNumber>`. For example, for Issue Number 15 that has a Status of "Closed", the sort key would be `ISSUE#CLOSED#00000015` (recall that we're zero-padding the Issue Number to 8 digits to allow for proper lexicographical sorting). Likewise, if Issue Number 874 has a Status of "Open", the sort key would be `ISSUE#OPEN#00000874`.

We could show those in an item collection as follows:

Primary key		Attributes	
Partition key: PK	Sort key: SK		
REPO#alexdebrie#dynamodb-book	ISSUE#CLOSED#00000015	IssueNumber	IssueStatus
		15	CLOSED
	ISSUE#OPEN#00000874	IssueNumber	IssueStatus
		874	OPEN
	REPO#alexdebrie#dynamodb-book	RepoOwner	RepoName
		alexdebrie	dynamodb-book

There's a problem here. The lexicographical ordering of our item is:

- Closed Issues
- Open Issues
- Repo

However, we have an access pattern that wants to fetch the Repo plus the most recent closed Issues. We can't do this because the open Issue items are between the Repo and the closed Issues!

One way you could solve this is to prefix the closed Issues and the Repo sort keys with a #. Your pattern would be as follows:

SK values:

- **Repo:** `#REPO#<RepoOwner>#<RepoName>`
- **Closed Issue:** `#ISSUE#CLOSED#<IssueNumber>`
- **Open Issue:** `ISSUE#OPEN#<IssueNumber>`

Primary key		Attributes	
Partition key: PK	**Sort key: SK**		
REPO#alexdebrie#dynamodb-book	#ISSUE#CLOSED#00000015	IssueNumber	IssueStatus
		15	CLOSED
	#REPO#alexdebrie#dynamodb-book	RepoOwner	RepoName
		alexdebrie	dynamodb-book
	ISSUE#OPEN#00000874	IssueNumber	IssueStatus
		874	OPEN
	ISSUE#OPEN#00000875	IssueNumber	IssueStatus
		875	OPEN

Open issues in ascending order

Now our Repo item is between the closed Issue and open Issue items. However, we have a new problem. When fetching our Repo item and open Issues, we'll be fetching the open Issues in *ascending* order, which will give us the oldest items first. We don't want that, so let's think of something else.

We could create a secondary index for each of these four access patterns (Repo + Open Issues, Repo + Closed Issues, Repo + Open Pull Requests, Repo + Closed Pull Requests). However, that's a lot of additional indexes to maintain and pay for, and we wouldn't be able to share capacity across them.

Instead, we're going to do something a little funky. Recall that we're zero-padding our issue number so that it's always 8 digits. We can get reverse ordering when scanning the index forward by subtracting our issue number from the maximum allowable issue number (99999999).

Let's see how this works in practice. For *open issues*, we will create the sort key in the following manner:

1. Find the difference between the maximum issue number (99999999) and the actual issue number. This is our *issue number difference*.

2. In the sort key, use the pattern of
   `ISSUE#OPEN#<IssueNumberDifference>`.

If your issue number was 15, the issue number difference would be 99999984 (99999999 - 15). Thus, the sort key for that issue would be `ISSUE#OPEN#99999984`.

Let's see how our table looks now:

Primary key		Attributes	
Partition key: PK	Sort key: SK		
REPO#alexdebrie#dynamodb-book	#ISSUE#CLOSED#00000015	IssueNumber	IssueStatus
		15	CLOSED
	#REPO#alexdebrie#dynamodb-book	RepoOwner	RepoName
		alexdebrie	dynamodb-book
	ISSUE#OPEN#99999124	IssueNumber	IssueStatus
		875	OPEN
	ISSUE#OPEN#99999125	IssueNumber	IssueStatus
		874	OPEN

Open issues in descending order

Great! Our open issues are now sorted in descending order even while searching forward in the item collection. This allows us to use one secondary index to handle both open and closed issue access patterns.

The pattern is basically the same for Pull Requests. We'll use the *pull request number difference* to order open pull requests as needed.

Note that all secondary indexes are already being used by the Repo item for other access patterns. Given that, we'll need to create two new secondary indexes: GSI4 and GSI5.

The attribute patterns to be added are as follows:

Repos:

- **GSI4PK:** `REPO#<RepoOwner>#<RepoName>`
- **GSI4SK:** `REPO#<RepoOwner>#<RepoName>`
- **GSI5PK:** `REPO#<RepoOwner>#<RepoName>`
- **GSI5SK:** `REPO#<RepoOwner>#<RepoName>`

Issues:

- **GSI4PK:** `#REPO#<RepoOwner>#<RepoName>`
- **GSI4SK:** `#ISSUE#CLOSED#<IssueNumber>` (Closed Issue)
- **GSI4SK:** `ISSUE#OPEN#<IssueNumberDifference>` (Open Issue)

Pull Requests:

- **GSI5PK:** `#REPO#<RepoOwner>#<RepoName>`
- **GSI5SK:** `#PR#CLOSED#<PRNumber>` (Closed Pull Request)
- **GSI5SK:** `PR#OPEN#<PRNumberDifference>` (Open Pull Request)

Note that all of these are existing items, so you'll need to do the Scan + UpdateItem pattern that was shown above in the GitHub Apps section to decorate your items with the new attributes.

Finally, note that this is one of the weirder patterns in DynamoDB. The zero-padded issue number and the issue number difference are both things that make no sense outside the context of DynamoDB. This is where your single-table DynamoDB table really is like machine code—indecipherable outside the context of your application's direct data access needs. If this pattern makes you uncomfortable, feel free to use the extra secondary indexes as discussed above. You may pay a bit more, but perhaps you'll save your sanity.

22.4. Conclusion

In this chapter, we made some pretty serious modifications to a complicated existing DynamoDB table. In doing so, we used a few different strategies.

First, we added a new one-to-one relationship when we added Codes of Conduct to Repos. Adding one-to-one relationships or individual attributes on an existing item is usually the easiest type of migration as you can lazily add it into your application code without making changes to existing items.

Second, we added a new entity, Gists, and a one-to-many relationship between Gists and Users. In adding this entity, we were able to reuse an existing item collection on the base table to co-locate Gists and Users. By doing so, we again were able to add this new access pattern without updating existing items.

Third, we added a new entity, GitHub Apps, which had two relationships: a one-to-many relationships with Accounts (Users or Organizations), and a many-to-many relationship with Repos. For this, we needed to do some ETL work to decorate our existing User and Organization items with new attributes to create a new item collection in a secondary index. Then we used the "adjacency list" pattern to handle the many-to-many relationship between GitHub Apps and Repos. We were able to reuse existing item collections in our base table and an existing secondary index to handle both sides of the many-to-many relationship.

Finally, we needed to refactor an existing access pattern based on some assumptions that proved faulty in production. To do this, we redesigned how we would access the data, then we ran an ETL process on our existing items to add new attributes. Finally, we added two new secondary indexes to handle these new patterns.

Our new entity charts looks as follows:

Entity	PK	SK
Repo	`REPO#<Owner>#<RepoName>`	`REPO#<Owner>#<RepoName>`
CodeOfConduct	N/A	N/A
Issue	`REPO#<Owner>#<RepoName>`	`ISSUE#<ZeroPaddedIssueNumber>`
Pull Request	`PR#<Owner>#<RepoName>#<ZeroPaddedPRNumber>`	`PR#<Owner>#<RepoName>#<ZeroPaddedPRNumber>`
IssueComment	`ISSUECOMMENT#<Owner>#<RepoName>#<IssueNumber>`	`ISSUECOMMENT#<CommentId>`
PRComment	`PRCOMMENT#<Owner>#<RepoName>#<PullRequestNumber>`	`PRCOMMENT#<CommentId>`
Reaction	`<TargetType>REACTION#<Owner>#<RepoName>#<TargetIdentifier>#<UserName>`	`<TargetType>REACTION#<Owner>#<RepoName>#<TargetIdentifier>#<UserName>`
User	`ACCOUNT#<Username>`	`ACCOUNT#<Username>`
Gist	`ACCOUNT#<Username>`	`GIST#<GistId>`
Organization	`ACCOUNT#<OrgName>`	`ACCOUNT#<OrgName>`
Payment Plan	N/A	N/A
Membership	`ACCOUNT#<OrgName>`	`MEMBERSHIP#<Username>`
GitHubApp	`APP#<AccountName>#<AppName>`	`APP#<AccountName>#<AppName>`
GitHubAppInstallation	`APP#<AccountName>#<AppName>`	`REPO#<RepoOwner>#<RepoName>`

Table 45. GitHub model entity chart

Additionally, we have the following entities and properties in the GSI1 index:

Entity	GSI1PK	GSI1SK
Repo	`REPO#<Owner>#<RepoName>`	`REPO#<Owner>#<RepoName>`
Pull Request	`PR#<Owner>#<RepoName>`	`PR#<ZeroPaddedPRNumber>`
User	`ACCOUNT#<AccountName>`	`ACCOUNT#<AccountName>`
Organization	`ACCOUNT#<AccountName>`	`ACCOUNT#<AccountName>`

Entity	GSI1PK	GSI1SK
GitHubApp	`ACCOUNT#<AccountName>`	`APP#<AppName>`
GitHubAppInstallation	`REPO#<RepoOwner>#<RepoName>`	`REPOAPP#<AppOwner>#<AppName>`

Table 46. GitHub model GSI1 entity chart

And GSI4:

Entity	GSI4PK	GSI4SK
Repo	`REPO#<RepoOwner>#<RepoName>`	`#REPO#<RepoOwner>#<RepoName>`
OpenIssue	`REPO#<RepoOwner>#<RepoName>`	`ISSUE#OPEN#<ZeroPaddedIssueNumberDifference>`
ClosedIssue	`REPO#<RepoOwner>#<RepoName>`	`#ISSUE#CLOSED#<ZeroPaddedIssueNumber>`

Table 47. GitHub model GSI4 entity chart

And finally GSI5:

Entity	GSI5PK	GSI5SK
Repo	`REPO#<RepoOwner>#<RepoName>`	`#REPO#<RepoOwner>#<RepoName>`
OpenPR	`REPO#<RepoOwner>#<RepoName>`	`PR#OPEN#<ZeroPaddedPRNumberDifference>`
ClosedPR	`REPO#<RepoOwner>#<RepoName>`	`#PR#CLOSED#<ZeroPaddedPRNumber>`

Table 48. GitHub model GSI5 entity chart

GSI2 and GSI3 remained unchanged in this migration.