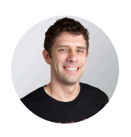
How you should think about DynamoDB costs

May 8, 2023 · 16 min read



Alex DeBrie
Founder, DeBrie Advisory

Last week, someone emailed me to ask about a potential cost optimization mechanism in DynamoDB. More on the specifics of that situation below, but the basic point is they were thinking about adding some additional application and architectural complexity because they were concerned about high DynamoDB costs for a particular use case.

I responded the way I always respond for these requests -- "have you done the math?"

One of my favorite things about DynamoDB is that you can easily do the math when considering how much it will cost you. I use this all the time in a few different ways, from getting a rough guess at how much DynamoDB will cost for my application to deciding between different approaches to solving a specific access pattern.

KEY POINT

You can and should think about DynamoDB costs as you're designing your model to understand feasibility and weigh tradeoffs.

In this post, I'll walk through my approach to reasoning about DynamoDB costs. I'll start with an overview about how DynamoDB pricing works -- feel free to skip this if you're already familiar. From there, I'll walk through a few examples of how I use this to make decisions about DynamoDB costs.

This post isn't comprehensive on DynamoDB costs -- it's more about how to approach this and make cost modeling part of your design process. If you want some other tips on DynamoDB costs, check out Khawaja Sham's epic thread on patterns for saving costs with DynamoDB. It has a ton of wide-ranging thoughts on DynamoDB costs.

There also was a nice Reddit thread on DynamoDB costs yesterday. Some correct stuff, some incorrect stuff, but I was happy to see a few people mention you can just do the math.

How DynamoDB pricing works

Let's start with a basic overview of how DynamoDB pricing works. You can get bogged down in the different pricing modes (on-demand vs. provisioned?) or table storage classes (Standard or Standard-IA?), but that is secondary to understanding the core dimensions on which DynamoDB charges.

Basically, DynamoDB is charging for three things:

- **Read consumption**: Measured in 4KB increments (rounded up!) for each read operation. This is the amount of data that is read from the table or index. I'll refer to each increment as an 'RCU' (read capacity unit) for this post, even though it's sometimes called a 'read request unit'. Thus, if you read a single 10KB item in DynamoDB, you will consume 3 RCUs (10 / 4 == 2.5, rounded up to 3).

- **Write consumption**: Measured in 1KB increments (also rounded up!) for each write operation. This is the size of the item you're writing / updating / deleting during a write operation. I'll refer to each write increment as a 'WCU' (write capacity unit). Thus, if you write a single 7.5KB item in DynamoDB, you will consume 8 WCUs (7.5 / 1 == 7.5, rounded up to 8).

- **Storage**: Measured in GB-months. This is the amount of data stored in the table or index, multiplied by the number of hours in the month. I almost never think about this during modeling as read and write consumption are usually the dominant cost factors.

The great thing about this is its predictability! You don't have to wonder about how many concurrent operations a `db.m6gd.large` instance can handle. You don't have to worry about how much data you can store in a 1TB `gp3` storage volume. You just have to worry about how much data you're reading, writing, and storing.

Not only this, but you can easily do a rough calculation of your costs as you're designing your data model. I'll walk through this further in the Doin' the Math sections below, but this is pretty straightforward math you can do with an Excel spreadsheet.

A few other quick notes / quirks about DynamoDB pricing:

- Notice that RCUs and WCUs are step functions -- you're charged for each increment, even if you don't use the full increment. For example, if you read 1.5KB of data, you'll be charged for 1 RCU. If you write 0.5KB of data, you'll be charged for 1 WCU.

- For reads, you have the option between a strongly consistent read request and an eventually consistent read request. An eventually consistent read request consumes half as many RCUs as a strongly consistent read request. **IMO, you should almost always use an eventually consistent read request.** If you think otherwise, let's fight :)

- A DynamoDB Query allows you to read multiple items in a single request. When doing a Query operation, the items will be added together first before dividing by 4KB to determine RCU usage. Thus, using a Query to read 100 items of 100 bytes each will cost you 3 RCUs (10KB / 4 == 2.5, rounded up to 3). If you had read each of those 100 items separately, it would have cost you 100 RCUs!

- For reads, you're charged for the data that's actually read, not the data returned to the client. Thus, DynamoDB Filter Expressions don't help the way you think they might. You should almost always use your key attributes to filter your results.

- When doing a write operation, you get charged the larger of the size of item as it was before the operation or as it is after the operation. For example, if you have an existing item that is 9.5KB and you increment a single counter integer on it, you will pay the full 10 WCUs for the operation. Likewise, if you delete that 9.5KB item, you will pay 10 WCUs.

- When you have a condition expression that fails, you'll still be charged WCUs for it. Further, it will be based on the size of the matching item (if any), with a minimum consumption of 1 WCU. (This one kind of makes me grumpy.)

- Using DynamoDB Transactions doubles your WCU usage. It's using two-phase commit under the hood, so basically think of being charged for each phase.

Choosing between provisioned and on-demand

In the examples below, we'll do some calculations to see how you can think about DynamoDB pricing. For my examples, I always calculate using DynamoDB On-Demand billing mode. On-Demand billing has a fixed price for reads and writes, so it's easy to calculate the cost of a particular access pattern. I don't have to think about things like utilization over time or providing a buffer to handle unexpected bursts of traffic. This greatly simplifies the math and gives you a directionally accurate look at your potential costs.

However, this doesn't mean you should never think about your billing mode! Once you've run the initial numbers, then you can fine-tune your calculations by looking at the difference between provisioned and on-demand.

At a first cut, if the on-demand numbers are negligible, you should probably stick with that. On-demand is going to be a hands-off solution that (almost completely) avoids throttling. If the potential bill won't break the bank, take the operational simplicity and move on.

However, when you're talking about real money, then you can consider whether provisioned capacity would be a worthwhile optimization. In general, on-demand billing is 6.94x as expensive as fully-utilized provisioned capacity.

This seems like a lot, but note the key qualifier there -- fully-utilized. You're not going to get anywhere near full utilization, so saving 85% is not a realistic goal. However, it's not unrealistic to think you could get to 50% utilization, which would save you 42% on your bill. For a high-traffic DynamoDB workload, that's a pretty significant savings!

Just remember what you're taking on. You're responsible for scaling up and down to account for traffic patterns. If traffic grows over time, you're responsible to account for the new highs. If you have a spike in traffic, you're responsible for scaling up to handle it. If you have a spike in traffic that you don't expect, you're responsible for handling it. If you don't do this, you'll get throttled and your users will be unhappy.

Doin' the math pt. 1: Tracking view counts

Alright, enough background chatter. Let's get into some examples.

The first example is what spurred this blog post. Someone wrote me an email about potential DynamoDB costs in tracking view counts. Every time a user visits one of his pages, he wants to increment a counter for that page.

He was worried about the cost of this and asked what I thought about batching page views in Redis for the short term, then occasionally writing them back to DynamoDB. This adds some cost (Redis instance!) and some complexity (multiple sources of data! syncing back to DynamoDB!), but he was thinking about doing it out of fear of a large DynamoDB bill.

The first thing I told him -- "Do the math!" Look at how big your item sizes are, then try to roughly estimate your traffic. I love spreadsheets, so I usually throw this in a spreadsheet and fiddle with the numbers to see what I'm dealing with.

In this situation, imagine his page record was pretty small -- under 1KB. If so, each write will be 1 WCU. Thus, he can estimate that his DynamoDB write costs will be $1.25 per million page views (on-demand billing is $1.25 per million WCUs).

This pricing may not be acceptable to you, but it gives you a sense of what you're dealing with. For him, it helped him realize that it wasn't something to optimize yet. He should spend his time on other things.

> PS: If you are thinking 'how do I know my item size?', generate an example record in your application and then paste it into Zac Charles' DynamoDB Item Size Calculator. I love this thing.

Note that I discussed an enhanced version of this during my 2022 talk at AWS re:Invent. I talked about 'vertical sharding', which can be a nice use of single-table design principles to split a single entity into multiple records to save on costs. For this example, imagine our underlying record was much larger -- 10KB instead of 1KB. Now, each million page views will cost us $12.50 even though we're just incrementing a counter.

Instead, we could separate our record into two items -- the page view counter and the actual item data. We can place them in the same item collection to fetch them in a single Query request when needed, but now a page view increment is only acting on the small, focused item. This would save 90% of our write costs for this example.

Doin' the math pt. 2: Do I really need that secondary index?

Alright, that's an easy example. Let's look at another example that involves some tradeoffs between different approaches to an access pattern within DynamoDB.

We saw earlier that each write to DynamoDB is charged 1 WCU per KB of data, rounded up. However, this applies *to each target* to which the write will be applied. I use the word 'target' to refer to not just the base table but any secondary indexes to which the item will be written.

It's not uncommon to have 4-5 secondary indexes on a DynamoDB table. When this happens, your write costs can quickly add up. Think of our view count example above. If we had 5 secondary indexes, we'd be paying 6 WCU per page view instead of 1 WCU!

There are lots of ways to reduce costs for GSIs that you truly need (check out Khawaja's tweet about saving money on GSIs during his awesome thread on DynamoDB savings generally), but sometimes you may be better off avoiding a GSI altogether.

Let's see an example.

In DynamoDB, I tell people that you almost always want to filter and sort the exact records you need via the key attributes. The key word here is 'almost'. There are certain times when it may be better to over-read your data.

Let's work with a simple example. Imagine you're selling a SaaS service that is purchased by teams. By nature of the industry you're in, it's natural that teams will be small. It would be extremely rare that over 100 people would belong to a single team. In the vast majority of cases, teams will be fewer than 10 users.

Each User record within the team is pretty small -- 1KB of data. However, you also have a few access patterns on the User -- maybe a point lookup to fetch a User by username, and two range queries -- one to fetch all the Users with a given role, and one to fetch all Users on the team ordered by the date added to the team.

You could set up a GSI for each of these secondary access patterns. However, you could also just handle the range queries by fetching all the Users on the team and filtering them in your application code. Even in the worst case scenario, you'll be fetching 100KB of data. That's 12.5 RCUs (100KB / 4 = 25 * 0.5 for eventually consistent reads = 12.5 RCUs). For most cases, it will be much smaller than that -- likely less than a single RCU. Further, RCUs are not only 4 times bigger than WCUs (4KB vs. 1KB), they're also 5 times cheaper ($0.25 per million as opposed to $1.25 per million for WCUs).

Based on some back-of-the-napkin math, it might be better to skip the GSI and over-read data to handle long-tail access patterns. This is particularly true when the overall result set is pretty small.

Again, you should still focus on using your key attributes as much as possible. However, there are times when you can relax that default assumption to save on write costs.

Doin' the math pt. 3: Complex filtering

One of the harder patterns in DynamoDB is what I call "complex filtering". This is when you may need to filter a dataset by a number of potential variables, all of which can be optional. DynamoDB wants to work with known access patterns, and this includes knowing the attributes you'll be filtering on. This kind of dynamism can be tricky for DynamoDB.

If this is a pattern you really need to support, you can again do some math to see if it's a feasible situation.

Another example. Imagine you have a CRM system that tracks customers and their orders. You want users to be able to filter their customers by a number of different attributes -- name, email, city, and industry segment. You also want to be able to filter by the date they were added to the system. Each of these attributes is optional in your filter.

These customer records can be pretty big -- let's say 4KB. Because of this, it's infeasible to add all the permutations of access patterns as GSIs. This could easily be 5 GSIs. This would make each update to your User cost 24 WCUs (4KB * 6 = 24 WCUs). This is a lot of write capacity for a single record!

But we also can't use the simple over-reading approach that we used in the previous example. Because each User is 4KB, we'll be paying half an RCU per customer on an eventually consistent read. A user might have a thousand customers in their system, so we'd be paying 500 RCUs per query to fetch them all. That's a lot of RCUs!

What are our options? Well, notice that while our application allows filtering on a few different attributes, it's still a very small portion of the record on which we filter. We only need about 100 bytes of the record, not the full 4KB.

We could use a secondary index and take advantage of index projections (more detail on secondary index projects from the wonderful Pete Naylor here). With an index projection, you can choose which attributes are included in the secondary index. As Pete notes, this helps us reduce writes by avoiding writes when only non-projected attributes are changing.

But it also helps in another way -- reducing our item size for complex filtering. If our item sizes are a mere 100 bytes, now we can fetch 40 customers per RCU unit. If we want to fetch 1000 customers, it only costs 40 RCUs (or 20 RCUs for eventually consistent reads). That's 4% of our original cost!

Note that if you need to fetch the full records, you'll need to do a follow-up, BatchGetItem operation to hydrate them. This is an additional cost to consider, both in terms of read costs and in response latency . However, if you're doing complex filtering, you may not be fetching the full record anyway. You're probably just showing a list of customers that match the filter, and then allowing the user to click on a customer to see the full record.

This pattern can also help you see when use cases are infeasible. If a user might have 100,000 customers in their CRM, now you're looking at 4000 RCUs to fetch and filter them all. If this is going to be a common pattern for you, you'll be burning through RCUs like crazy. At this point, I usually recommend integrating something like (e.g. Rockset or Elasticsearch) to augment your core DynamoDB patterns with some complex filtering. This adds cost and complexity, but it's often the right tradeoff for your use case.

Conclusion

In this post, we learned how to think about DynamoDB costs. We walked through the background of how DynamoDB does billing and then worked through some examples of how to think about DynamoDB costs in practice.

The strongest point I want to emphasize is that you should use the predictability and legibility of DynamoDB to your advantage. Use it to understand whether an access pattern will be cost effective. Use it to weigh the tradeoffs of different approaches to DynamoDB. And, yes, use it to understand when DynamoDB might not be the right fit for a particular use case.

Tags: serverless AWS DynamoDB