One Year of DynamoDB

Kevin Cantwell Follow
Mar 16, 2015 · 8 min read

at Timehop

2,675,812,470

That's the number of rows in our largest database table. Or it was, one year ago today. Since then, it's grown to over 60 billion rows. On average, that's roughly 160mm inserts per day.

When I think about it, that seems like quite a lot. But because the database we use is Amazon's DynamoDB, I rarely have to think about it at all. In fact, when Timehop's growth spiked from 5,000 users/day to more than 55,000 users/day in a span of eight weeks, it was DynamoDB that saved our butts.

What is DynamoDB?

I don't intend to explain the ins and outs of what DynamoDB is or what it's intended for, there's plenty of documentation online for that. But I will mention the most salient point: That DynamoDB is consistent, fast, and *scales without limit.* And when I say "scales without limit," I literally mean there is no theoretical boundary to be made aware of. That isn't to say that all of DynamoDB's features behave identically at every scale, but in terms of consistency and speed it absolutely does. More on that below. First, some history.

The Ex I Worry About Seeing in My Timehop is Mongo

Timehop currently uses DynamoDB as it's primary data store. Every historical tweet, photo, checkin, and status is persisted to DynamoDB as time-series data. It wasn't always this way. Back in ye olden days, we used a combination of Postgres and Mongo. So why the switch?

Listen, Mongo is great. I mean, I'm sure it's pretty good. I mean, there's probably people out there who are pretty good with it. I mean, I don't think my soul was prepared for using Mongo at scale.

When we were small, Mongo was a joy to use. We didn't have to think too hard about the documents we were storing and adding an index wasn't very painful. Once we hit the 2TB mark, however, we started to see severe and unpredictable spikes in query latency. Now, this might have been due to our hosting provider and had nothing to do with Mongo itself (We were using a managed service through Heroku at the time). Or, if we rolled our own we might have solved things with a sharded solution. But to be honest, none of us at Timehop are DBAs and a hosted database has always been a more attractive option (As long as it didn't break the bank!). So we started to look at alternatives. Our friends at GroupMe had some nice things to say about using DynamoDB and it seemed pretty cost-effective too, so we gave it a go.

Early Mistakes with DynamoDB

As is often the case with new technologies, DynamoDB first seemed unwieldy. The biggest hurdle for us right out of the gate was the lack of a mature client library written in Go, our primary language (Things have changed since then). But once we nailed down the subset of API queries we actually needed to understand, things started to move along. That's when we started making real mistakes.

Mistake #1: Disrespecting Throughput Errors

This one should have been obvious as the error handling docs are quite clear. DynamoDB has two main types of http errors: retryable and non-retryable. The most common type of retryable error is a throughput exception. The way dynamo's pricing model works is that you pay for query throughput, where reads and writes are configured independently. And the way Amazon enforces throughput capacity is to throttle any queries that bust it. It's up to the client to handle the errors and retry requests with an exponential backoff. Our early implementations assumed this would be a rare occurrence, when in reality, you will likely incur *some* amount of throttling at any level of usage. Make sure you handle this frequent type of error.

Mistake #2: Not Understanding How Partitions Affect Throughput

All DynamoDB tables are partitioned under the covers and each partition gets a sliver of your provisioned (purchased) capacity. The equation is simple:

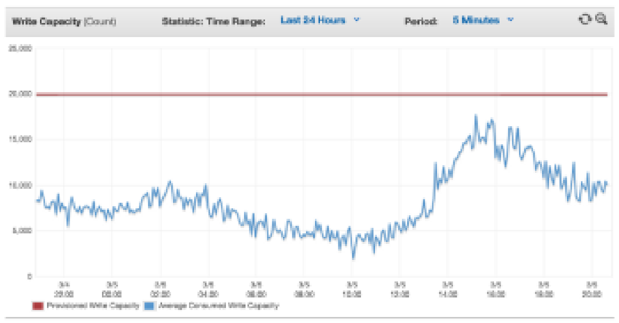
```
Total Provisioned Throughput / Partitions = Throughput Per Partition
```

What this means is that if your query patterns are not well distributed across hash keys, you may only achieve a fraction of your provisioned throughput. Not understanding this point can lead to serious headaches.

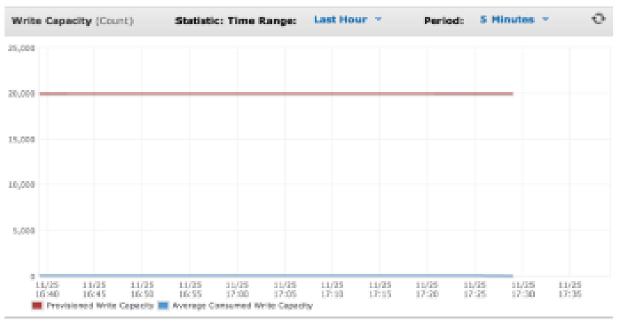
It's also important to say that the number of partitions that exist are not directly exposed to you (However, there is now a helpful guide to help you estimate the number of partitions you are accruing). A single table may have 1 partition or 1000. That means it's difficult to predict exactly how hot a key can become. And the larger your table grows the more partitions that will be allocated to it. Which means you may not even notice hot key issues until you are at scale. The best thing you can do is fully understand how to design and use a well distributed hash key.

Mistake #3: Poorly Designed Hash Keys

DynamoDB tables can be configured with just a hash key, or with a composite hash and range key. Timehop's data is in time-series, so for us a range key is a necessity. But because hash keys always map to a single virtual node in a partition, a large set of range keys per hash key can lead to hot-key problems. I dug into my old emails and found a few write capacity graphs that illustrate this point:



March, 2015



March, 2014

The red lines here indicate provisioned throughput, ie: the most you could ever use. The blue lines represent actual throughput achieved.

The top graph represents our current schema, which has a well-designed hash key based on a combination of user id and calendar date.

The bottom graph shows our first stab last March, where our hash key was just the user id. Notice the difference? Yeah, it was bad. It took us about 3 attempts at loading and destroying entire tables before we got it right.

Advanced Lessons

Once we stopped making the above mistakes we found further ways to optimize our DynamoDB usage patterns and reduce costs. These points are actually covered in the developer guidelines, but we had to make our own mistakes to reach the same conclusions.

Lesson #1: Partition Your Hash Keys

Every hash key will consistently hit a single virtual node in DynamoDB. That means that every range key associated with a hash key also hits that same node. If a Timehop user has 5000 items of content (a rough average) and most of that content is written at around the same time, that can result in a very hot hash key. Anything you can do to split the data across multiple hash keys will improve this. The pros and cons of doing so are:

- **Pro**: Hash key distribution increases linearly with each hash key partition; Throughput errors decrease.
- **Con**: The number of network queries required to write the same amount of data increases linearly with each hash key partition; the likelihood of i/o errors and poor network latency increases.

For Timehop our hash key partition strategy was a natural function of our use case: We partition all user data by date. This gives us a significant increase in hash key distribution. As an example, a user with 5000 items of content across 7 years of social media activity sees about a 2.5k times increase in distribution:

Without partitioning (ie: hash key = “<user_id>”):

```
5000 items per hash key on average
```

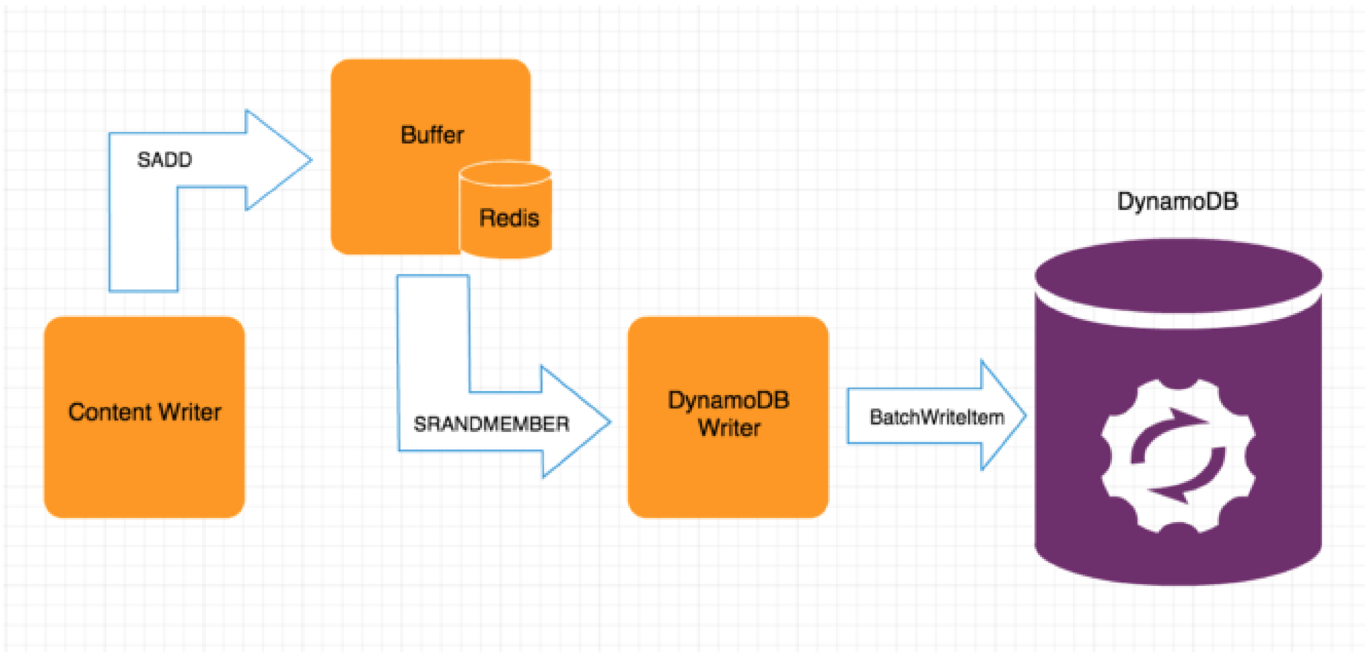
With partitioning (ie: hash key = “<date>:<user_id>”):

```
5000 / 365 days / 7 years = about 2 items per hash key on average
```

Lesson #2: Temporally Distribute Your Write Activity

As I stated earlier, a table that uses a range key can expect at least *some* throttling to occur. How many errors you experience is directly tied to how well distributed your hash keys are. However, if you really want to maximize your throughput you should also consider temporally distributing the writes you make against a single hash key.

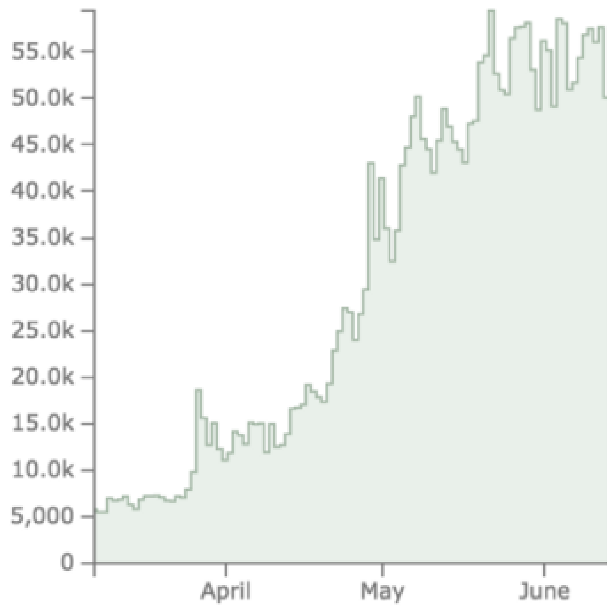
Our writes happen asynchronously when a user signs up and connects their social media accounts. At any given moment the write requests that occur are heavily skewed towards the subset of hash keys that represent the most recent new user. In order to encourage well-distributed writes, we built a simple write buffer that sits in front of DynamoDB (essentially a managed Redis set). A separate process drains random elements from the buffer and issues batch write requests to DynamoDB. The more users that sign up together, the more data that ends up in the buffer. The key benefit of this architecture pattern is the more data that's in the buffer, the more randomization there is when draining it and therefore more distributed the hash keys are in our batch write requests. *So increasing scale actually improves our write performance*. Yay!



Scaling Without Limit

So now that we've got our usage patterns stabilized, what's the real benefit? The most explicit win we saw from using DynamoDB was during our most significant period of user growth, which just happened to coincide with our final draft of a DynamoDB strategy.

]



Dat growth doe

During the 8 week period between March and May of 2014 our user growth started to spike upwards from roughly 5k signups/day to more than 55k signups/day. As you can imagine we were hyper-focused on our infrastructure during this time.

There were many many fires. Each dip in the graph probably corresponds to some outage or backend service falling over due to load. But what the dips definitely *do not* represent, is an outage or performance problem with DynamoDB. In fact, for the last year, no matter how large our table grew or how many rows we've written to it (currently in the order of hundreds of millions per day) the query latency has *always* been single digit milliseconds, usually 4–6.

I can't stress enough how nice it was not to have to think about DynamoDB during this time (aside from a few throughput increases here and there). This was a new experience for Timehop Engineering as in the past the database was always the choke point during periods of growth. The stability and performance of DynamoDB allowed us to focus our efforts on improving the rest of our infrastructure and gave us confidence that we can handle the next large spike in user signups.

AWS describes DynamoDB as a providing "single-digit millisecond latency *at any scale*" (emphasis mine). We've asked the DynamoDB team what the theoretical limits are and the answer is, barring the physical limitations of allocating enough servers, there are none. In evidence of this, our table is quite large now, about 100TB, and the performance is the same as on day one.

Thanks for reading! I hope you've gleaned something useful from this post. We will continue to optimize our DynamoDB usage patterns where we find opportunity and broadcast the lessons we learn along the way.

by Kevin Cantwell, Lead Architect @Timehop

Dynamodb Timehop Startup