Scaling a Startup Using DynamoDB

The Pros and Cons of DynamoDB

Leonard Austin Follow
Aug 16, 2016 · 7 min read

TL;DR

- If you are happy hosting your own solution, use Cassandra. If you want the ease of scaling and operations, Use DynamoDB.
- You MUST distribute your hash keys correctly. Hot keys kill you and increasing provisioning to try to beat them with brute force will not help
- Where possible feed DynamoDB from asynchronous queues. Requeue on throughput exceptions.
- Read the Limits document. All of it. Not just the sections you think are relevant to you.
- DynamoDB is NOT a swiss army knife. It is very good at high throughput reads and writes with ~10ms response time on simple queries. It is not good at complex queries. Use other tools for that (Elasticsearch, Google BigQuery, Postgres, etc)
- There is still some work to do with dynamic scaling. The 4 downscales per day limit is a real problem.

At Ravelin we are constantly grappling with the problems of rapid scaling. Similar to many companies, as we grow and our client base widens, we need to increase our scale to meet the demand of new business. The extent of this scaling can be challenging to

predict as it's not a linear problem. Bringing on a number of small clients will have a relatively small impact on scale, whereas having the team land a "big fish" client can dramatically increase the throughput to our system on a short time scale. With this in mind, we have chosen technologies that have proven scaling potential wherever possible.

An example of this is DynamoDB.

Why DynamoDB?

In the early days of Ravelin, we knew we wanted a NoSQL solution that allows write scaling to go beyond one machine. We were (and still are for some of our data) using Postgres since it allows for more rapid and agile development. This is particularly useful when the query pattern is still in a state of flux and we don't have a proper data warehouse to do offline analytics. Postgres gave us the flexibility to build better solutions while providing a decent working "stopgap" that will scale for a long time. That said, eventually we know we were going to outgrow Postgres since we query huge quantities of data in realtime from a large number of clients. In a sense, we aren't scaling to fulfil the needs of one company, our scale needs to meet the needs of an ever-growing number of client companies.

We first considered Cassandra since we have a number of developers with experience using and building data models for it. Generally, we have been impressed with its performance in the past. Companies like Netflix and Hailo have extolled the virtues of the technology; even Apple, in a rare, public engineering talk, endorses Cassandra. But we are a small team and try to use managed services when possible, to save ourselves the overhead of managing infrastructure within the company. With this in mind, we decided to keep Cassandra on the back burner and pick a similar technology but one with push button management.

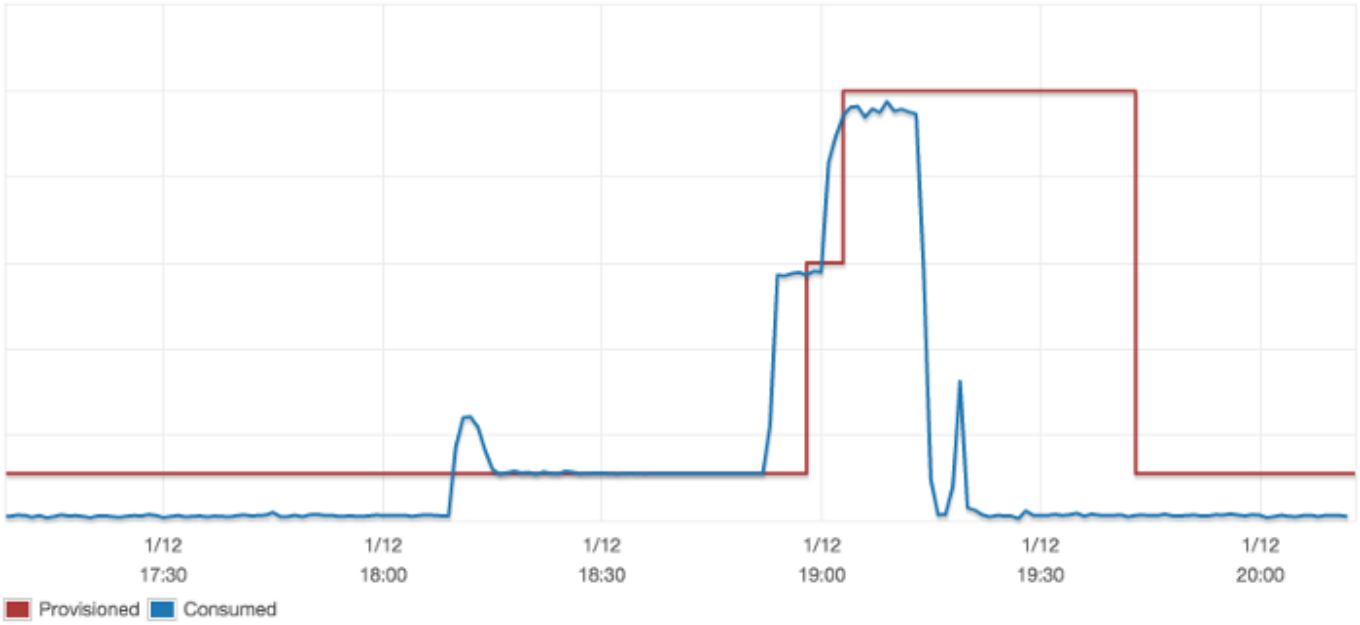
We run in AWS and given these restrictions, the only option is DynamoDB. Furthermore, we reached out to some of our colleagues with DynamoDB experience and they were largely positive about the product.

First Impressions

We wrote an initial small service that would pump data into DynamoDB, which enabled us to exercise the product and learn its basic limitation. For this simple test, we did not

consider the data model to the degree we would need to for a production system. Getting this test working took little time as Amazon takes care of installing, automating and managing a DynamoDB cluster. There was zero setup required by developers and/or devops. This is an understated key feature that almost completely erases worry over instances and deployment in DynamoDB.

The results of this initial service with DynamoDB, in terms of scaling potential, were really positive. We would hit our system with load, then up the throughput with the push of a button and we would quickly get the scale required.



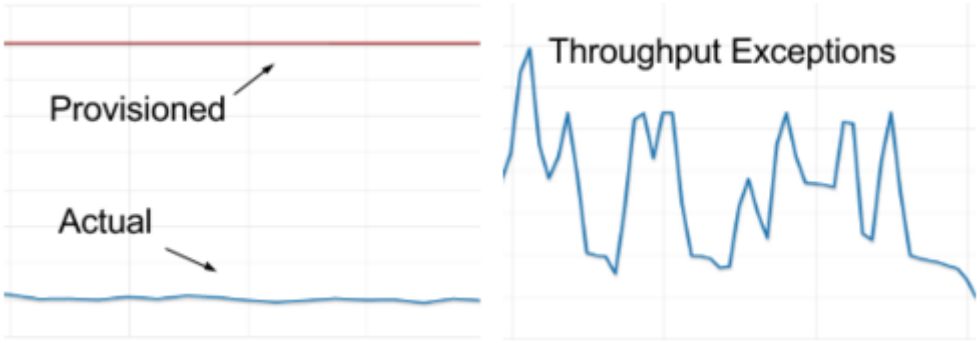
In the graph above, we overloaded the system for about 30 minutes and then increased the throughput. Almost immediately, DynamoDB responded allowing a much larger set of requests through. Our system is largely asynchronous, so we can afford to have writes retry on queues as long as it doesn't get too far behind. So not only did it keep up with the extra traffic but also powered through the backlog on the queue quickly. This dynamism is great for keeping costs down, which can be a concern on DynamoDB. If load increases, knowing that we can easily provision more throughput on a temporary basis is a great weight off our minds.

Beyond performance, we quickly ran into problems with queryability, which wasn't surprising as we hadn't spent a huge amount of effort on the data model. Our team's previous experience with Cassandra elsewhere, which is a very similar system, was

enough to tell us this was going to make getting the answers we wanted quite challenging.

First Mistake

After our initial test, we started again with the data model, moving away from simple unique keys to something that gave us a bit more query potential. This is when we really learned about DynamoDB. We decided to key our data based on an identifier for our clients. In theory, since we had a number of clients this would shard across whatever cluster Amazon had created and proper scaling would happen. In practice however, this didn't happen and instead, we got something like this:



In some cases, we actually got less throughput when we scaled. which was because we had a few clients with a huge amount of traffic and a number of smaller clients with a small amount of traffic. Our sharding key was not balanced. When Amazon scales DynamoDB in the background, it may increase the number of machines while decreasing the size of the machines. Since throughput is measured per machine we were overloading the instances to which those particular hot key were mapping. This is the wrong way to use DynamoDB.

Second Mistake

Even though we had scaling issues, we assumed that as we gained more clients this would distribute the load. In theory, having more client identifiers in the system would balance the sharding. We hoped we would eventually find a scaling point that worked for us. What we didn't know yet is that we had made another mistake which was fatal. We had used local secondary indexes to allow for more complicated query patterns but what we didn't realize was that there was a 10GB limit on data under a single key when using this index. This eventually brought our system to a screeching halt. Luckily, we were not dependant on DynamoDB yet and still used Postgres for the majority of our

data. I cannot stress enough, of all the docs to read for DyanmoDB, the Limits doc is the most important one. Read it top to bottom. You can't just read relevant sections because the doc mentions limits pertaining to all aspects of the product throughout.

Getting It Right

We went back to the drawing board and came up with a new data model that both ensured we had distributed hash keys but also allowed for the limited queries we wanted to perform. We used a concatenated key of the ID for the client and also unique ID's for the entities being stored. This limited how we could query the data but such are the restrictions of a NoSQL solution. We soon realized we were trying to solve too many problems with one tool. With that in mind, we limited the scope of Dynamo's function to only real time data for analysis. The data in DynamoDB is not structured to populate a dashboard nor is it structured to work well for more complicated analytics. For these tasks, we chose two different technologies: Elasticsearch and Google BigQuery.

Elasticsearch provides complex query capabilities (particularly listing and time series) that allow us to drive a dashboard with decent response times. Google BigQuery provides data warehousing, which gives us deeper queryability over huge sets of data. By splitting up the tasks, we stopped asking too much of DynamoDB and could focus on what it does well: high throughput read and write at ~10ms response time on simple queries.

Automated Scaling

Recently, we have been toying with automatically scaling throughput in DynamoDB. We have a few tables that have highly variable traffic on them and keeping them at a level that will always allow for enough throughput is expensive. We have been manually scaling this as required but this is a frustrating task for a human therefore we have tried out Dynamic DynamoDB, as suggested by Amazon.

Dynamic DynamoDB is a good temporary solution but there is more work to be done in this area. The config is finicky and challenging to tune correctly and it's unaware of the 4 downscales per 24 hour period that Amazon limits you to. The script will often overcompensate for incoming traffic and crank the provisioned throughput only to be unable to bring it down again for 18 hours, costing much more than if we have done it manually. We would really love if Amazon allowed for more downscaling events or if this script was more intelligent about when to scale up and down.

Should You Use DynamoDB?

That really depends on your usage needs. If we had a larger team and were more accepting of DevOps responsibilities, I think Cassandra would be the better choice. There is more flexibility with Cassandra and it is likely more cost efficient. That said, once you understand the limitations of DynamoDB and spec your use case accordingly, it is a decent product that is very simple to integrate if you are on AWS.

*Originally written by **Jono MacDougall** on https://www.ravelin.com/blog*

AWS Dynamodb Bigquery Startup