Recent posts

How you should think about DynamoDB costs

Event-Driven Architectures vs. Event-Based Compute in Serverless Applications

Why I (Still) Like the Serverless Framework over the CDK

Key Takeaways from the DynamoDB Paper

Understanding Eventual Consistency in DynamoDB

Get the DynamoDB Book

Key Takeaways from the DynamoDB Paper

July 4, 2022 · 18 min read

Alex DeBrie
Founder, DeBrie Advisory

In 2007, a group of engineers from Amazon published The Dynamo Paper, which described an internal database used by Amazon to handle the enormous scale of its retail operation. This paper helped launch the NoSQL movement and led to the creation of NoSQL databases like Apache Cassandra, MongoDB, and, of course, AWS's own fully managed service, DynamoDB.

Fifteen years later, the folks at Amazon have released a new paper about DynamoDB. Most of the names have changed (except for AWS VP Swami Sivasubramanian, who appears on both!), but it's a fascinating look at how the core concepts from Dynamo were updated and altered to provide a fully managed, highly scalable, multi-tenant cloud database service.

In this post, I want to discuss my key takeaways from the new DynamoDB Paper.

There are two main areas I found interesting from my review:

- The product-level, "user needs" learnings of the DynamoDB service; and
- The technical improvements over the years to further develop the service.

This post will be at a higher-level than the paper, though I strongly recommend reading the paper itself. It's really quite approachable. Additionally, Marc Brooker has written a nice review post that includes some interesting systems-level thoughts on the paper.

Product-level takeaways from the DynamoDB Paper

Both the Dynamo paper and the DynamoDB paper describe some incredible technical concepts, but I'm equally impressed by the discussion of user needs. In both papers, there is a deep review of existing practices to see what is important and what should be re-thought around core user needs.

In the Dynamo paper, we saw this in the articulation that much of the higher-level querying functionality provided by an RDBMS is unused by Amazon's services. Werner Vogels expanded on this later as he mentioned that 70% of database operations were single-record lookups using a primary key, and another 20% read a set of rows but only hit a single table.

The Dynamo paper also noted that the traditional guarantee of strong consistency, while critical in some circumstances, was not necessary for all applications. In many cases, the enhanced availability and reduced write latency achieved by relaxing consistency requirements was well worth the tradeoff.

Just as the Dynamo paper re-examined some shibboleths from traditional database systems, the DynamoDB paper explores user needs around what was needed to make the Dynamo learnings applicable more broadly. In doing so, DynamoDB was able to institute a set of clear product priorities that distinguish DynamoDB from many other database offerings.

There are three important notes on user needs that I took from the paper:

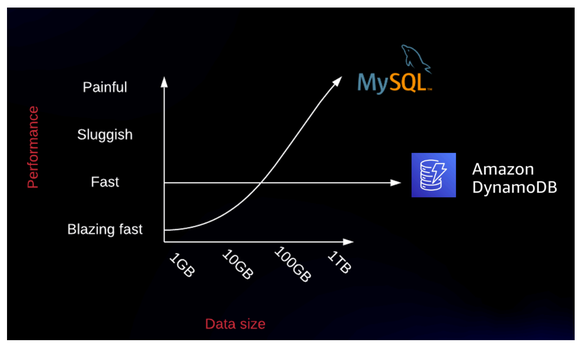
- The importance of consistent performance
- Fully managed is better than self managed
- User data isn't as evenly distributed as you want

The importance of consistent performance

One point that the DynamoDB paper hammers over and over is that, for many users, "consistent performance at any scale is often more important than median request service times." Stated differently, it's better to have a narrower range between median and tail latency than it is to reduce median (or even p90 or p95) latency.

This is a surprising point to some people who think "DynamoDB is super fast." The story is more nuanced than that.

In my DynamoDB talks, I often show the following:



In this chart, we see that RDBMS latency will get worse as the amount of data in your database increases, whereas DynamoDB will provide consistent performance as your data increases. This same relationship holds as the number of concurrent requests increases.

The chart above is a rough sketch, but the overall point stands. At certain levels of data and transaction volume, *an RDBMS will have faster response times than DynamoDB*. Conceptually, this is easy to understand. A request to DynamoDB will pass through a number of systems -- a request router, a metadata system, an authentication system -- before making it to the underlying physical storage node that holds the requested data. While these systems are highly optimized, each one of them adds latency to the overall request. Conversely, a single-node RDBMS can skip a lot of that work and operate directly on local data.

So MySQL might beat DynamoDB at the median, but that's not the full story. There are two reasons you should also consider the full spectrum of your database's performance.

First, tail latencies are important, particularly in architectures with lots of components and sub-components. If a single call to your backend results in a lot of calls to underlying services, then each request is much more likely to experience the tail latency from *some* service, resulting in a slower response.

Second, the consistency and predictability of DynamoDB's performance profile leads to less long-term maintenance burden on your service. You don't have to come back to investigate, tune, and refactor as your performance inevitably declines. You know you'll get the same performance in your test environment as you will five years after launch, allowing you to focus on more value-generating features for your users.

Fully managed over self-managed

If you're reading this blog, you're probably drinking the cloud Kool-Aid and may even be fully into the serverless world. In the serverless world, we're as focused as possible building the key differentiators of our business while offloading the undifferentiated heavy lifting to others.

But the internal experience of Amazon retail and the creators of *Dynamo* (not DynamoDB) really drives this home.

Recall that the Dynamo paper was hugely influential in the industry, and Amazon's internal Dynamo system was a big improvement in terms of availability and scalability for the enormous scale at which Amazon was operating.

Despite this improvement, many internal engineers chose to eschew running Dynamo themselves in favor of using Amazon SimpleDB, which was AWS's first foray into the hosted NoSQL database market.

If you've never heard of Amazon SimpleDB, you're not alone. DynamoDB is essentially a successor to SimpleDB and is superior in nearly every aspect. AWS rarely markets it anymore, and it's mostly a mascot for how AWS will never deprecate a service, even when better options exist.

SimpleDB has some truly surprising downsides, such as the fact that the *entire database* cannot exceed 10GB in size. This is a huge limitation for most applications but particularly for a company who has applications so large that they had to design a completely new database. Yet engineers were choosing to use batches of multiple SimpleDB tables to handle their needs, likely sharding at the application layer to keep each database under 10GB.

This had to add significant complexity to application logic. Despite this, engineers still chose to use SimpleDB over operating their own Dynamo instance.

This revealed preference by Amazon engineers helped spur the development of Amazon DynamoDB, a database that combined the scalability of Dynamo with the fully managed nature of SimpleDB.

User data isn't as evenly distributed as you want

The final user takeaway is that you have to work with the users you're given, not the users you want. In an ideal world, users would have steady, predictable traffic that spread data access evenly across a table's keyspace. The reality is much different.

The original Dynamo paper used the concept of consistent hashing to distribute your data across independent *partitions* of roughly 10GB in size. (Partitions are discussed in more depth below). It uses the partition key of your items to place data across the partitions, which allows for predictable performance and linear horizontal scaling.

Further, unlike the original Dynamo system, DynamoDB is a multi-tenant system. Your partitions are co-located with partitions from tables of other DynamoDB users.

Originally, the DynamoDB team built a system to avoid noisy neighbor issues where high traffic to one partition results in a reduced experience for unrelated partitions on the same storage node. However, as the system developed, they realized that the initial system to manage this led to a subpar experience for those with spiky, unbalanced workloads.

As the AWS team built out DynamoDB, they realized they needed to evolve the access control system that managed whether a partition was allowed to service a request. We look more into the technical aspects of this evolution below.

Technical takeaways from the DynamoDB Paper

The product-level learnings are fascinating, but this is ultimately a technical paper. The work the DynamoDB team is doing at massive scale is impressive, and many of the technical learnings apply even to those without DynamoDB's scale.

I had three technical takeaways that were most interesting from the paper:

- The use of log replicas to improve durability and availability;
- The incremental steps taken to improve servicing of unbalanced workloads; and
- The use of asynchronous cache mechanisms to improve performance, availability, and resiliency of underlying services

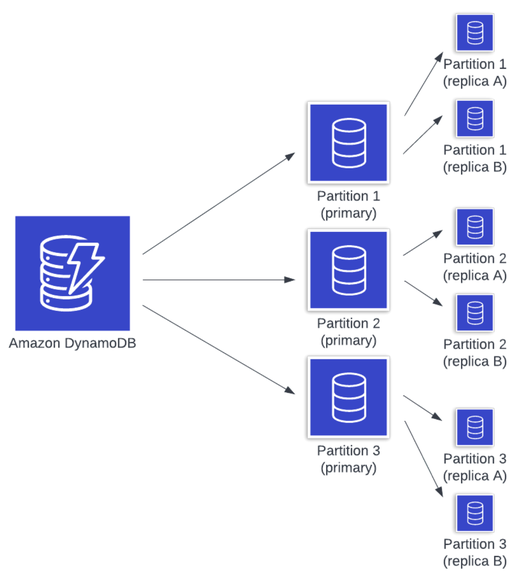
Using log replicas to improve durability and availability

One of the more interesting points was how DynamoDB uses something called log replicas to assist during periods of instance failure. To understand log replicas, we first need some background on the underlying architecture of DynamoDB's storage.

Start of DynamoDB storage background section

Under the hood, DynamoDB is splitting your data into *partitions*, which are independent storage segments of roughly 10GB in size. DynamoDB uses the partition key to assign your items to a given partition, which allows DynamoDB to scale horizontally as your database grows while still keeping related items together. DynamoDB is running a massive fleet of storage nodes which are handling partitions from many different user tables.

An individual partition is actually a set of three partition instances in different availability zones which form a *replication group*. One of the instances is the leader for a given partition and is responsible for handling all writes. When a write comes in, the leader writes it locally and ensures it is commited to at least one additional replica before returning to the client. This increases durability in the event of failure, as the loss of one node will not result in loss of data.



On each storage partition are two data structures -- the B-tree that contains the indexed data on the partition along with a write-ahead log (WAL) that contains an ordered list of updates applied to that partition. A write-ahead log is a commonly used tactic in databases to enhance the durability and latency of write operations. Updating the B-tree is slower as it involves random I/O and may include re-writing multiple pages on disk, whereas updating the write-ahead log is an append-only operation that is much faster (P.S. the write-ahead log is basically the concept behind Kafka and related systems!).

Note that in addition to the performance difference of an individual operation against these structures, there's also a vast difference in the size of these two structures. The B-tree can be 10+ GB in size (accounting for the 10GB size of a partition along with the index overhead), whereas the write-ahead log is only a few hundred megabytes (the full history of the write-ahead log is periodically synced to S3, which is used to power point-in-time restore and other features).

End of DynamoDB storage background section

When you're running a system as large as DynamoDB, instances are failing all the time. When a storage node fails, you now have potentially thousands of partition replicas that you need to relocate to a non-failing node. During this failure period, every replication group that had a replica on that node is now down to two replicas. Because two replicas are required to acknowledge a given write, you're now increasing the latency distribution of your writes as well as the probability of an availability event if another replica fails.

To reduce the period where only two replicas are live, DynamoDB uses *log replicas*. A log replica is a member of a replication group that contains only the write-ahead log. By skipping the B-tree for the partition, the DynamoDB subsystem can quickly spin up a log replica by copying over the last few hundred MB of the log. This log replica can be used to acknowledge writes but not to serve reads.

While this log replica is helping the replication group, the DynamoDB subsystem can work in the background to bring up a full replica member to replace the failed one.

It's so interesting to see the incremental tweaks the team has made to continually push the envelope on durability, availability, and latency. Most of these aren't altering the core promises that the system makes, such as making a different choice on the CAP

theorem. Rather, they're steady improvements to the reliability and performance of a database.

First, DynamoDB uses the standard combination of a write-ahead log with the indexed storage to improve durability and reduce latency on write requests.

Second, DynamoDB discards the typical single-node setup of a RDBMS and moves to a partitioned system. This reduces recovery time (and hence availability) as recovering a 10GB partition is much faster than recovering a 200GB table. Further, this replication is across three different availability zones (AZs) so that an entire AZ can go down without affecting availability of the system.

Then, DynamoDB relaxes the consistency requirements to require that only two of the three nodes in a replication group acknowledge the write. At the cost of some occasionally stale data, DynamoDB is able to enhance the availability and reduce the latency of writes.

Finally, DynamoDB uses log replicas to improve availability during periods of node failure.

Decoupling partitions from throughput

In the previous section, we discussed how partitions are used to segment data within a table and allow for horizontal scaling. Additionally, we saw how DynamoDB co-locates partitions from different customers on the same storage nodes to allow for greater efficiency of the DynamoDB service.

A second interesting technical takeaway is the slow, steady improvements to the "admission control" system for these partitions. Admission control refers to the process in which DynamoDB determines whether a request can succeed based on the amount of capacity available. In determining this, DynamoDB is checking capacity across two axes:

- Is there capacity availabile *for the customer* based on the amount of capacity provisioned for the table?
- Is there capacity available *for the partition* based on the total capacity for the storage node on which the partition lives?

The first one is a cost decision, as DynamoDB wants to make sure you're paying for the service they're delivering. The second one is a performance decision, as they want to avoid noisy neighbor issues from co-located partitions.

The first iteration of admission control was purely at a partition level. DynamoDB would divide the total provisioned throughput by the number of partitions and allocate that amount to each partition evenly. This was the easiest system, as you didn't have to coordinate across lots of partitions on a second-by-second basis. However, it led to issues with unbalanced workloads and the "throughput dilution" problem. This could lead to situations where requests to hot partitions were being throttled even though the table wasn't using anywhere near its provisioned capacity.

To fix this problem, DynamoDB wanted to decouple admission control from partitions but realized this would be a big lift. To handle this, they moved in stages.

First, they improved the partition-level admission control system. While each partition was limited to prevent over-consumption of resources on an individual node, they also realized that storage nodes were often running under full capacity. To help with temporary spikes in traffic to individual partitions, DynamoDB added short-term bursting that would let a partition use additional throughput if it was available for the given storage node. This improvement was mostly focused on the second axis of access control -- protecting against noisy neighbors.

A second initial improvement helped with the other axis of access control -- the provisioned throughput for an individual table. As mentioned, a table with skewed access patterns might consume all the throughput for one partition while still being below the total provisioned throughput for the table. To help with this, DynamoDB added adaptive capacity, where throughput from sparsely used partitions could be shifted to highly used partitions.

These two changes, while still maintaining the general partition-based access control scheme, alleviated a significant amount of pain based on uneven access patterns of data.

Later, DynamoDB moved to a global access control system which decoupled throughput from partitions entirely. This changed adaptive capacity from a slower, 'best efforts' system to a nearly instant system to spread your throughput across your partitions. This flexibility led to amazing other improvements, including the ability to separate particularly hot items onto their own partitions, to provide DynamoDB On-Demand billing, and to 'overload' storage nodes based on predicted workloads of the underlying partitions.

All of this is recounted in more detail in Section 4 of the paper, and it is well worth your own read to understand the details.

The use of asynchronous caches

The last technical takeaway was in DynamoDB's use of asynchronous caches. By "asynchronous cache", I'm meaning a system that caches data locally but then rehydrates the cache behind the scenes, asynchronously, to ensure it stays up to date.

We all know caches as a way to reduce latency by storing the results of an expensive call. In both cases mentioned in the paper, individual request router instances are storing the results of external calls locally to avoid a slow network request. But there are two more subtle points that are pretty interesting. In reviewing these, we should note how DynamoDB treats "external" systems (also called "dependencies") from "internal" systems.

DynamoDB uses other AWS services, such as IAM to authenticate requests or KMS to encrypt and decrypt data. Both of these services are external dependencies as they're not under the control of the DynamoDB team. Here, DynamoDB will cache the results of calls to these services as a way to increase availability. These results are periodically refreshed asynchronously to ensure freshness. This allows DynamoDB to keep working (somewhat) even if these external services are having issues themselves. Without this, DynamoDB's availability would necessarily be lower than those of IAM and KMS.

DynamoDB also uses asynchronous caches for 'internal' systems. DynamoDB has a metadata system that tracks table information and locations for each DynamoDB partition. When a request comes to a DynamoDB request router, it needs to find the relevant partition for the given item to forward the request to the storage node.

This metadata information doesn't change frequently, so the request routers heavily cache this data. The paper notes that the cache hit rate is *99.75%* (!!), which is quite good. However, a high cache hit rate can also lead to problems where slight decreases in traffic can result in significantly more load to the underlying service. Decreasing the metadata cache hit rate from 99.75% to a still-amazing 99.5% results in *twice as many requests* to the underlying metadata service.

The DynamoDB team found that the metadata service had to scale in line with the request router service, as new request routers had empty caches that resulted in a lot of calls to the metadata service. This led to instability in the overall system.

To increase resiliency of its internal systems, DynamoDB uses asynchronous cache refreshing to provide constant load to the underlying metadata system. While the request routers would cache locally with a high hit rate, each hit results in an associated request to the metadata service to refresh the cached data.

By pairing a local cache hit with an asynchronous request to the metadata service, it ensures a more consistent rate of traffic to the metadata service. Both a cache hit and a cache miss result in a request to the metadata service, so increasing the number of request routers with cold caches doesn't result in a burst of new traffic to the metadata service.

There's a lot of other really interesting information about the metadata caching system that I won't cover here, but I thought these two uses of asynchronous caches were interesting. Both used local, instance-based caching to reduce latency but also coupled with asynchronous refreshing to decouple availability from external dependencies and to increase the resiliency of internal services.

Conclusion

Once again, Amazon has helped to push forward our understanding of deep technical topics. Just as the Dynamo paper was revoluationary in designing new database architectures, the DynamoDB paper is a masterful lesson in running and evolving large-scale managed systems.

In this post, we looked at the core learnings from a user-needs perspective in the DynamoDB paper. Then, we looked at three technical learnings from the paper.

The paper has a number of other interesting points that we didn't cover, such as how DynamoDB monitors client-side availability by instrumenting internal Amazon services, the strategies used to deploy new versions of DynamoDB against an enormous fleet of instances, and the mechanisms used to protect against data errors, both in flight and at rest. If you write up an examination of those points, let me know and I'll link to them here!