All Articles

SQL, NoSQL, and Scale: How DynamoDB scales where relational databases don't

Over the past few years, DynamoDB has gotten more and more popular as a database. This is for a few reasons, such as the way it fits so well with serverless architectures using AWS Lambda or AWS AppSync or due to the growing community around how to model DynamoDB spurred by the talks from the incredible Rick Houlihan.

When I talk to people who are new to DynamoDB, I often hear the same initial grumblings:

- *"Wow, this feels weird."*
- *"Why doesn't DynamoDB have joins?"*
- *"Why can't I do any aggregations?"*
- *"Why is DynamoDB so hard?"*

To understand the answer to these complaints, you need to know one key thing about DynamoDB:

DynamoDB won't let you write a bad query

When I say "bad query", I mean "query that won't scale". DynamoDB is built for scale. When you write an application using DynamoDB, you'll get the same performance characteristics when there's 1 GB of data as when there's 100GB of data or 100TB of data.

The same isn't true of relational databases. Your application might feel snappy when it's first implemented, but you'll see performance degradation as your database grows.

In this post, we'll look at how databases do and don't scale. We'll cover the following topics:

- The biggest problem with relational databases: unpredictability

- How DynamoDB 'shifts left' on scalability
- The three reasons relational databases run into trouble at scale
- How DynamoDB avoids the three problems
- Where you might see performance problems with DynamoDB.

Note: None of this suggests that DynamoDB is the One True Database that should be used for everything. There are tradeoffs in everything, and you should consider yours carefully when choosing a database.

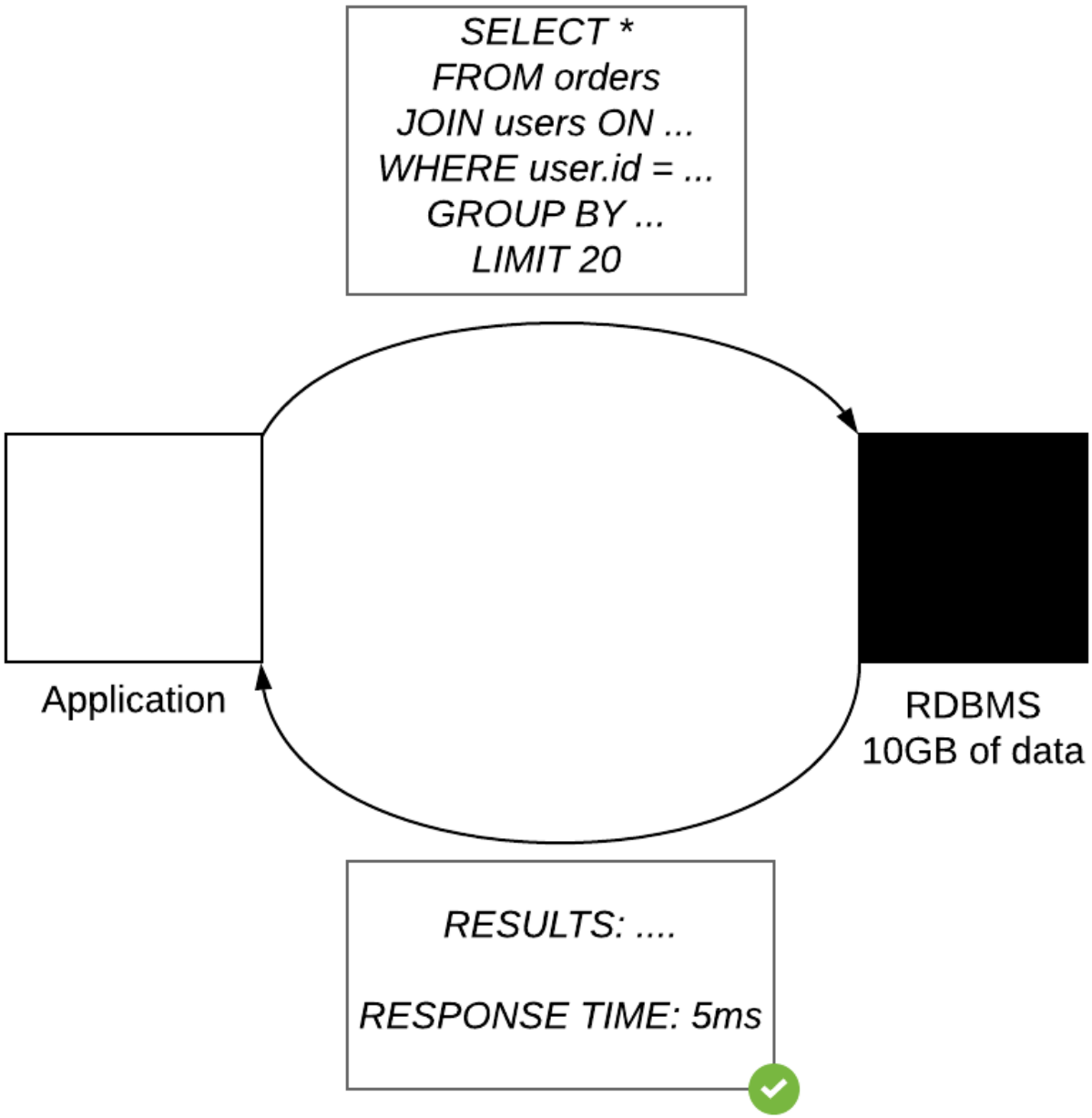
Let's get started!

The unpredicatability of relational databases

Relational databases are a wonderful piece of technology. They're relatively easy to pick up, which has made them the first database to learn for new developers. They work for a wide variety of use cases and thus are the default choice for most applications. And relational databases have SQL, a flexible query syntax that makes it easy to handle new access patterns or to handle both OLTP and OLAP patterns in a single database.

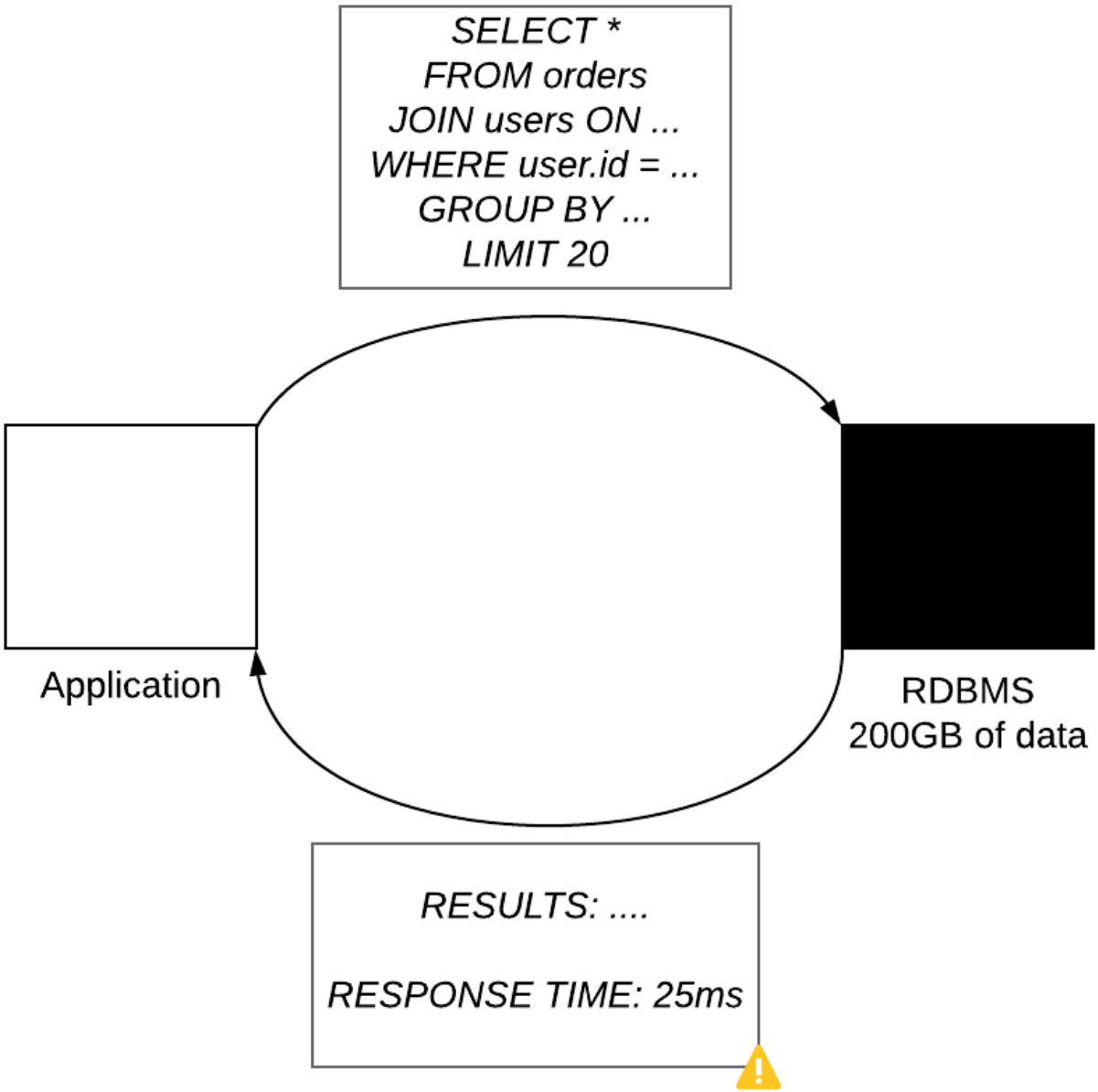
But there's one big problem with relational databases: performance is like a black box. There are a ton of factors that impact how quickly your queries will return.

During testing and early in your application's lifecycle, you might have a query in your application that returns quite quickly:

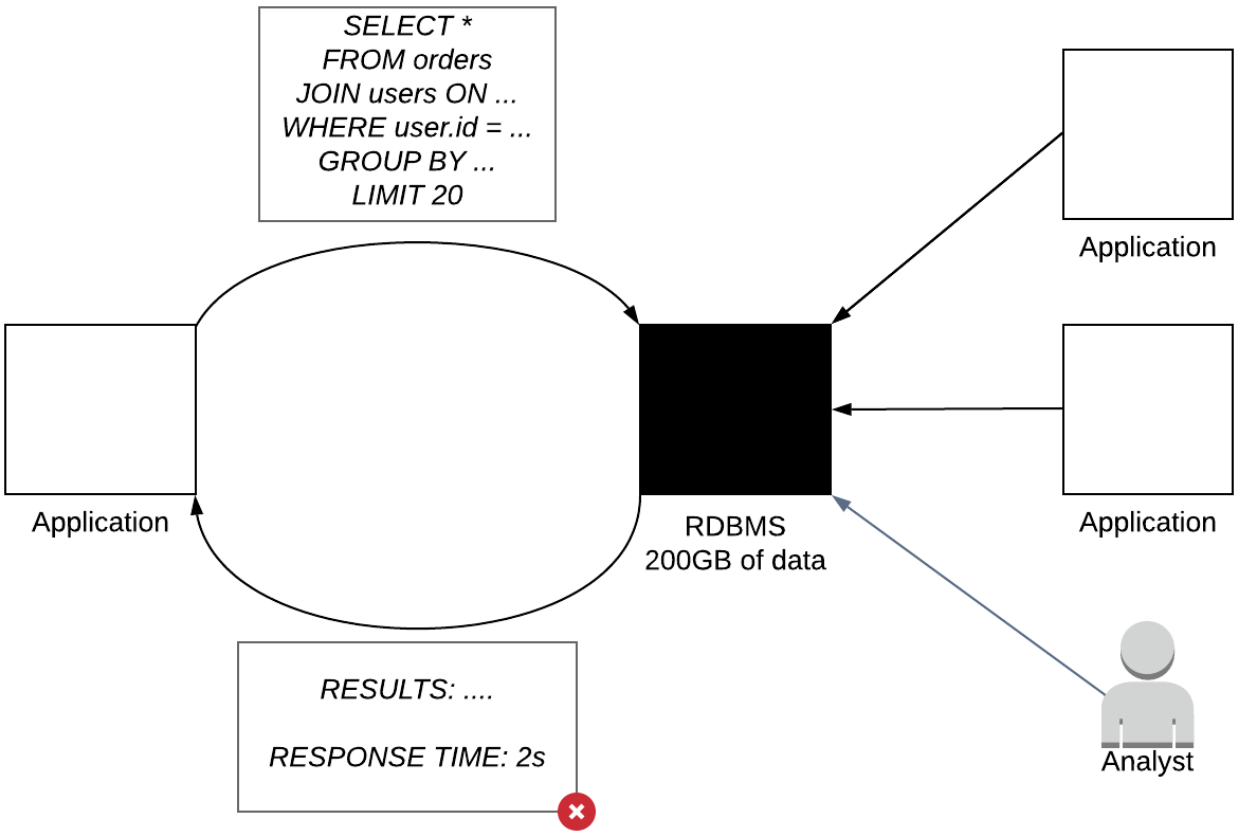


In these early stages, the query returns quite quickly and all is well.

But what happens as your dataset scales? In our example query, we have both a `JOIN` and a `GROUP BY` statement. As the size of your tables grow, these operations will get slower and slower.



And it's not just the size of your tables that you need to worry about. Relational database performance is also affected by other queries that are running at the same time. What happens to your query performance when 100 of these big queries are running at the same time? Or if your product manager is running a big analytics query against your database to find the top users from the last month?



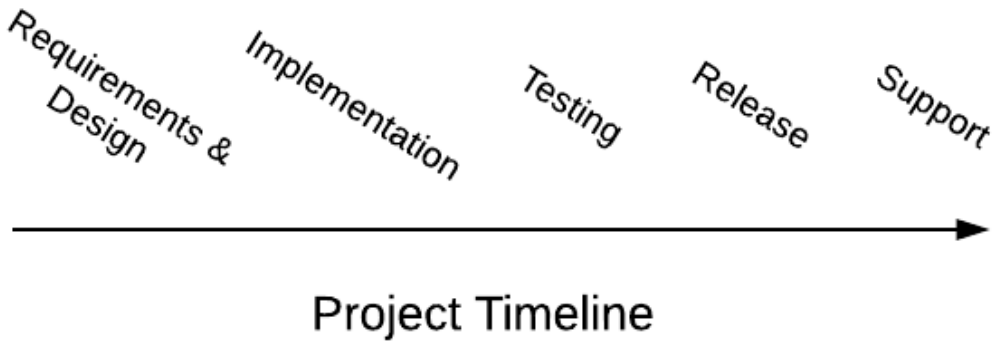
The performance of your relational database is subject to a number of factors — table size, index size, concurrent queries — that are difficult to test accurately ahead of time. Even if you have a pretty good idea of the number of requests per second you'll need to support, it's tricky to convert that into the proper instance size (# of CPUs + RAM) that can handle those requests.

It's this unpredictability that can be maddening with a relational database.

How DynamoDB shifts left on scalability

One major change in software development over the past twenty years is the idea of shift-left testing. Shift-left testing refers to thinking about testing earlier in the software development lifecycle rather than waiting until the end.

Imagine a chart with an X-axis representing the full lifecycle of building a production, from requirements & design to development to support. Typically, testing would be handled later in the lifecycle after initial design and implementation was complete.



The traditional software development model — most testing comes after implementation.

Shift-left proponents want to include testing earlier in the lifecycle, or moving left on the X-axis (hence the name, 'shift-left'). The main reason is that it's better to find defects upfront while it's easier to change and less time has been wasted on development.

The shift-left model of software development — do more testing upfront.

There's a similar model at play with databases and scalability. Many databases are difficult to performance test ahead of time. Everything goes great in testing environments at low scale.

In essence, this is the "traditional model" of scalability — you do some work related to scalability ahead of time, such as index creation and capacity planning. However, most of

the problem is backloaded to when your application hits real scale and starts to stretch the limits of your database.

DynamoDB, on the other hand, uses a shift-left model of scalability. It forces you to think about designing your data model up front in a way that will scale.

With DynamoDB, there's no black box as to how your database will scale:

- How fast will my query return when the scale is 100X as large? **The same — single-digit milliseconds**
- How will my query impact other queries in the database? **It won't — queries are bounded in their resource impact**

This shift-left approach is not costless. You'll need to put in the time to really understand your application's access patterns up front and design your data model accordingly. But the payoffs are big — you won't need to rework your data model due to scaling issues. You'll get the exact same performance at 10TB as you got at 1GB.

How does DynamoDB do this? It involves a few key design decisions in DynamoDB. But before we dive into that, let's look at the different ways that relational databases don't scale, as this informs the design decisions behind DynamoDB.

Why relational databases don't scale

Ahh, the RDBMS — the wonderful technology that underlies banking systems, hospitals, and of course the 90% of the internet that runs on Wordpress.

The RDBMS has served us remarkably well and is still a great choice for many applications. It is well-known by developers, is flexible to evolve as your needs change, and provides both OLTP and OLAP functionality.

However, we've seen a number of companies move from RDBMS to NoSQL databases over the last twenty years due to the issues with RDBMS at scale. For more on this, take a look at the Dynamo Paper, written by some Amazon.com engineers (including current AWS CTO Werner Vogels) which lays out the concepts behind the Dynamo database built for

Amazon.com. The Dynamo Paper includes many of the design decisions behind DynamoDB and inspired later NoSQL databases like Apache Cassandra.

There are three big problems with relational databases that make it difficult to scale:

- The poor time complexity characteristics of SQL joins;
- The difficulty in horizontally scaling; and
- The unbounded nature of queries

We'll explore each of these below.

SQL joins have bad time complexity

The first problem with RDBMS stems from the join operation in SQL.

In a SQL join operation, your relational database is combining results from two or more different tables. Not only are you reading quite a bit more data but you're also comparing the values in one to the other to determine which values should be returned.

Rick Houlihan, in his epic 2019 re:Invent talk on DynamoDB, walked through some of the time complexity of SQL JOINs as compared to DynamoDB operations. Your results will vary based on the type of JOIN you're using as well as the indexes on your tables, but you may be getting linear time (`O (M + N)`) or worse.

These CPU-intensive operations will work fine during testing or while your data is small. As your data grows, you'll slowly start to see CPU usage rise and query performance fall. As such, you may want to scale up your database. This leads us to the next issue with relational databases.

It's difficult to horizontally scale an RDBMS

The second problem with RDBMS is that they're difficult to horizontally scale.

There are two ways to scale a database:

- **Vertical scaling**, by increasing the CPU or RAM of your existing database machine(s), or
- **Horizontal scaling**, by adding additional machines into your database cluster, each of which handles a subset of the total data.

At lower scale, there's probably not a huge price difference between vertical and horizontal scaling. In fact, with AWS, they've done a pretty good job of price parity between one large machine vs multiple smaller machines.

However, you will eventually hit the limits of vertical scaling. Your cloud provider won't have a box big enough to handle your growing database. At this point you may want to consider horizontal scaling. Now you're running into bigger problems.

The whole point of horizontal scaling is that each machine in your overall cluster will only handle a portion of the total data. As your data continues to grow, you can add additional machines to get the amount of data on each machine relatively steady.

This is tricky with a relational database, as there's often not a clear way to split up the data. You'll often want to store all results from a given table on the same machine so that you can correctly handle uniqueness and other requirements.

If you split tables across different machines, now your SQL joins require a network request in addition to the CPU computation.

Horizontal scaling works best when you can shard the data in a way that a single request can be handled by a single machine. Jumping around multiple boxes and making network requests between them will result in slower performance.

Relational queries are unbounded

The third problem with RDBMS is that, by default, SQL queries are unbounded. There's no inherent limit to the amount of data you can scan in a single request to your database, which also means there's no inherent limit to how a single bad query can gobble up your resources and lock up your database.

A simple example would be a query like `SELECT * FROM large_table` — a full table scan to retrieve all the rows. But that's pretty obvious to see in your application code and

less likely to consume all your database resources.

A more nefarious example would be something like the following:

```
SELECT user_id, sum(amount) AS total_amount
FROM orders
GROUP BY user_id
ORDER BY total_amount DESC
LIMIT 5
```

The example above examines the `orders` table in an application to find the top 5 users by the amount spent. An application developer might think this isn't an expensive operation, as it's only returning 5 rows. But think of the steps involved in this operation:

- Scan the entire orders table
- Group orders by the user id and sum the order amount by user to find a total.
- Order the results from the previous step according to the summed total.
- Return the top 5 results

This is expensive on a number of different resources — CPU, memory, and disk. And it's completely hidden from the developer.

Aggregations are bear traps for the unwary, ready to take down your database right when you hit scale.

How NoSQL databases handle these relational problems

In the previous section, we saw how relational databases run into problems as they scale. In this section, we'll see how NoSQL databases like DynamoDB handle these problems. Unsurprisingly, they're essentially the inverse of the problems listed above:

- DynamoDB does not allow joins;

- DynamoDB forces you to segment your data, allowing for easier horizontal scaling; and
- DynamoDB puts explicit bounds on your queries.

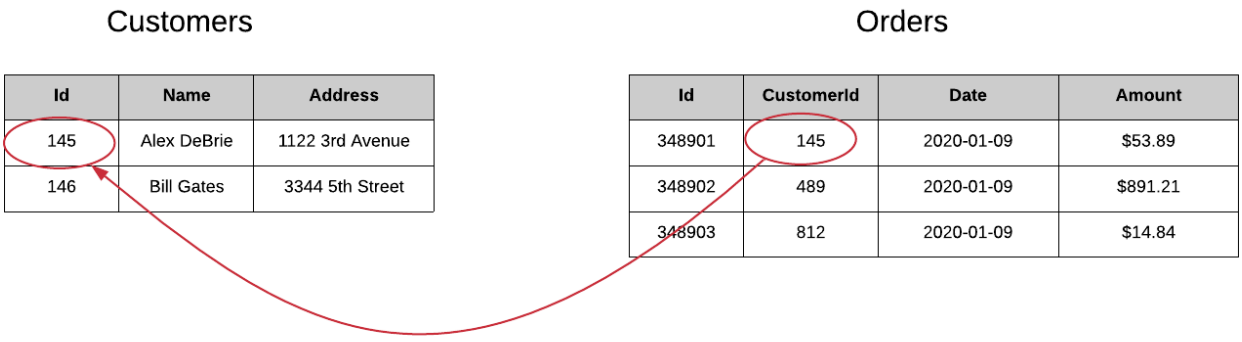
Again, let's review each of these in order.

How NoSQL databases replace joins

First, let's see how NoSQL databases replace joins. To do that, we should understand the jobs that joins are doing in a relational database.

The canonical way to model your data in a relational database is to normalize your entities. Normalization is a complex subject that we won't cover in depth here, but essentially you should avoid repeating any singular piece of data in a relational database. Rather, you should create a canonical record of the data and reference this canonical record whenever needed.

For example, a customer in an e-commerce application may make multiple orders over the course of the year. Rather than storing all customer information on the order record itself, the order record would contain a `CustomerId` property which would point to the canonical customer record.



In the example above, Order #348901 has a `CustomerId` property with a value of `145`. This points to the record in the Customers table with an `Id` of `145`. If Customer #145 changes something about him, such as his name or address, the Order doesn't have to make corresponding changes.

There are two benefits to this approach:

1. *Storage efficiency*: Because a single data record is 'write once, refer many', relational databases require less storage than options which duplicate data into multiple records.
2. *Data integrity*: It's easier to ensure data integrity across entities with a relational database. In our example above, if you duplicate the customer information into each order record, then you may need to update each order record when the customer changes some information about themselves. In a relational database, you only need to update the customer record. All records that refer to it will receive the updates.

However, normalization means that your data is scattered all over the place. How do I retrieve both the order record and the information about the customer that made it?

This is where joins come in. Joins allow you to reassemble data from multiple different records in a single operation. They give you enormous flexibility in accessing your data. With the flexiblity of joins and the rest of the SQL grammar, you can basically reassemble any of your data as needed. Because of this flexiblity, you don't really need to think about how you'll access your data ahead of time. You model your entities according to the principles of normalization, then write the queries to handle your needs.

However, as noted above, this flexibility comes at a cost — joins require a lot of CPU and memory. Thus, to get rid of SQL joins, NoSQL needs to handle the three benefits of joins:

1. Flexible data access
2. Data integrity
3. Storage efficiency

Two of these are handled by specific tradeoffs, while the third is less of a concern anymore.

First, NoSQL databases avoid the need for *flexibility* in your data access by requiring you to do planning up front. How will you read your data? How will you write your data? When working with a NoSQL database, you need to consider these questions thoroughly before designing your data model. Once you know your patterns, you can design your database to handle these questions specifically. Rather than reassembling your data at read time, you will 'pre-join' your data by laying it out in the way it will be read.

The second tradeoff of NoSQL databases is that *data integrity is now an application-level concern*. While JOINs would allow you for a 'write once, refer many' pattern for referenced items, you may need to denormalize and duplicate data in your NoSQL database. For pieces of data that are unchanging — birth dates, order dates, sensor readings — this duplication is

no problem. For data that does change, like display names or listed prices, you may find yourself updating multiple records in the event of a change.

Finally, NoSQL databases are less storage efficient than their relational counterparts, but it's mostly not a concern. When RDBMS were designed, storage was at more of a premium than compute. This is no longer the case — storage prices have dropped to the floor while Moore's Law is slowing down. Compute is the most valuable resource in your systems, so it makes sense to optimize for compute over storage.

Why NoSQL databases can scale horizontally

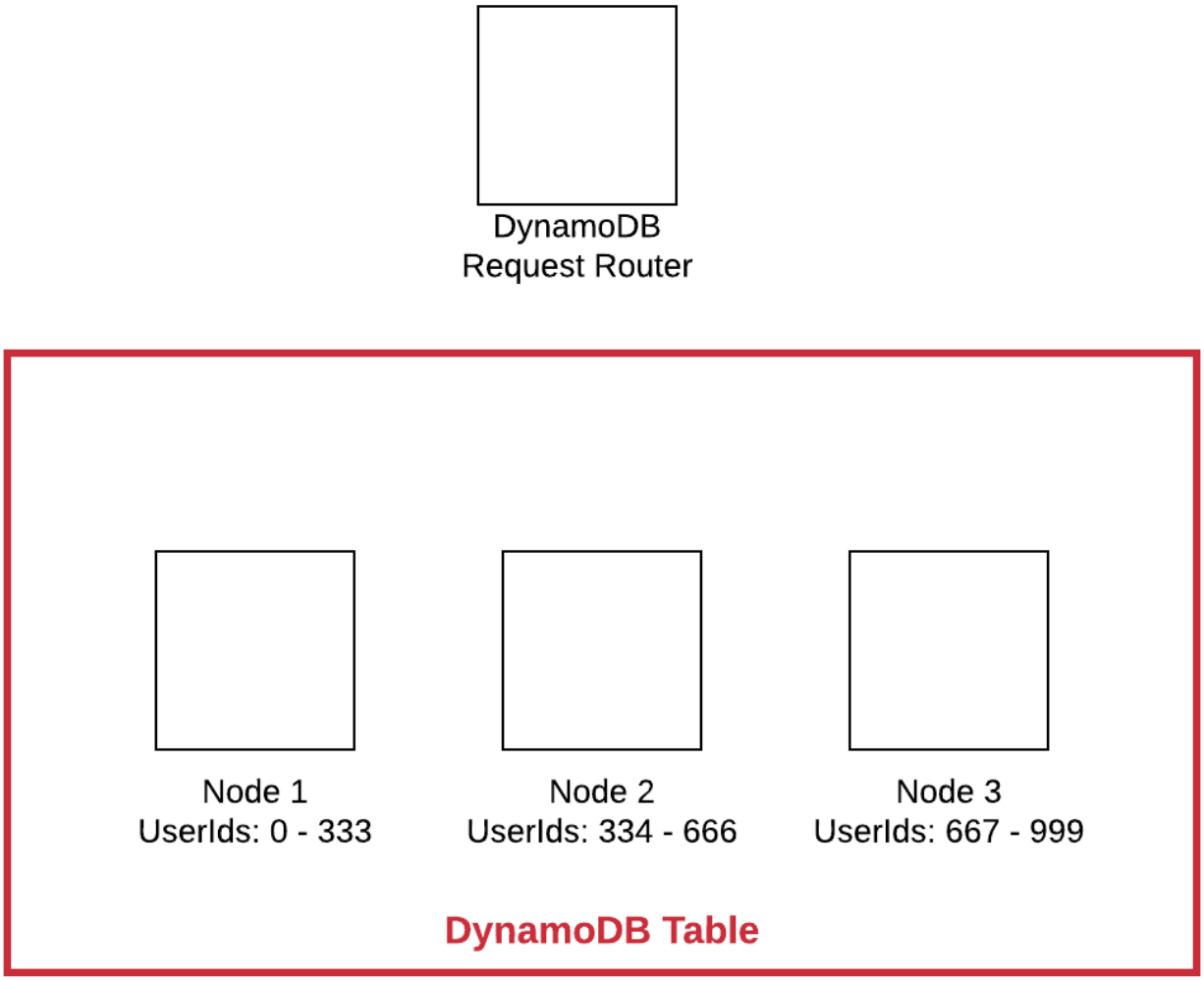
In the previous section, we saw how NoSQL databases handle the time complexity problem around SQL joins by requiring you to arrange your data such that it is pre-joined for your use case. In this section, we'll see how NoSQL solves the scaling problem by allowing for horizontal scaling.

The main reason relational databases cannot scale horizontally is due to the flexibility of the query syntax. SQL allows you to add all sorts of conditions and filters on your data such that it's impossible for the database system to know which pieces of your data will be fetched until your query is executed. As such, all data needs to be kept local, on the same node, to avoid cross-machine network calls when executing a query.

To avoid this problem, NoSQL databases require you to split up your data into smaller segments and perform all queries within one of these segments. This is common across all NoSQL databases. In DynamoDB and Cassandra, it's called a partition key. In MongoDB, it's called a shard key.

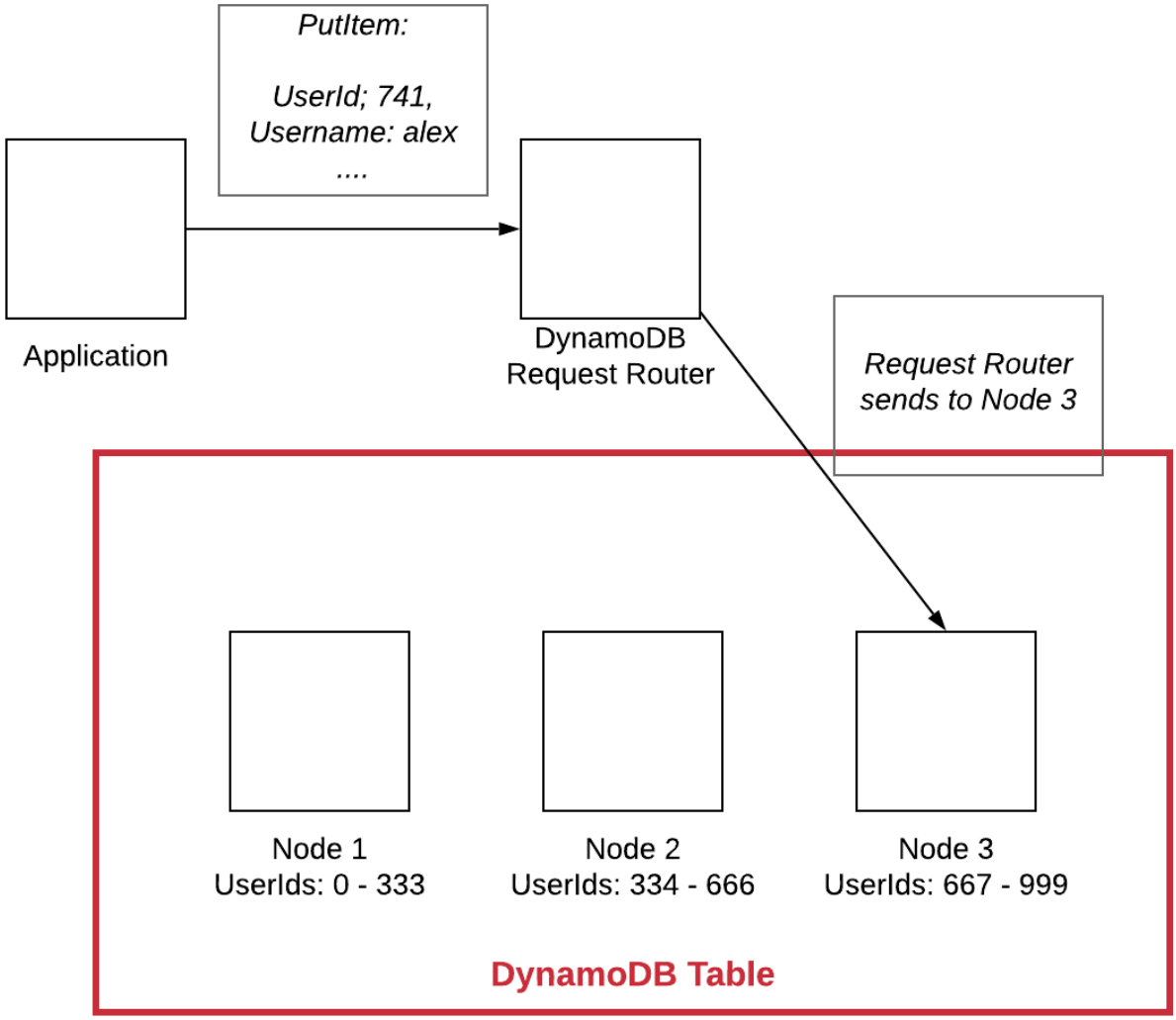
The easiest way to think of a NoSQL database is a hash table where the value of each key in the hash table is a b-tree. The partition key is the key in the hash table and allows you to spread your data across an unlimited number of nodes.

Let's see how this partition key works. Assume you have a data model that is centered around users and thus you use `UserId` as your partition key. When you write your data to the database, it will use this `UserId` value to determine which node it should use to store the primary copy of the data.



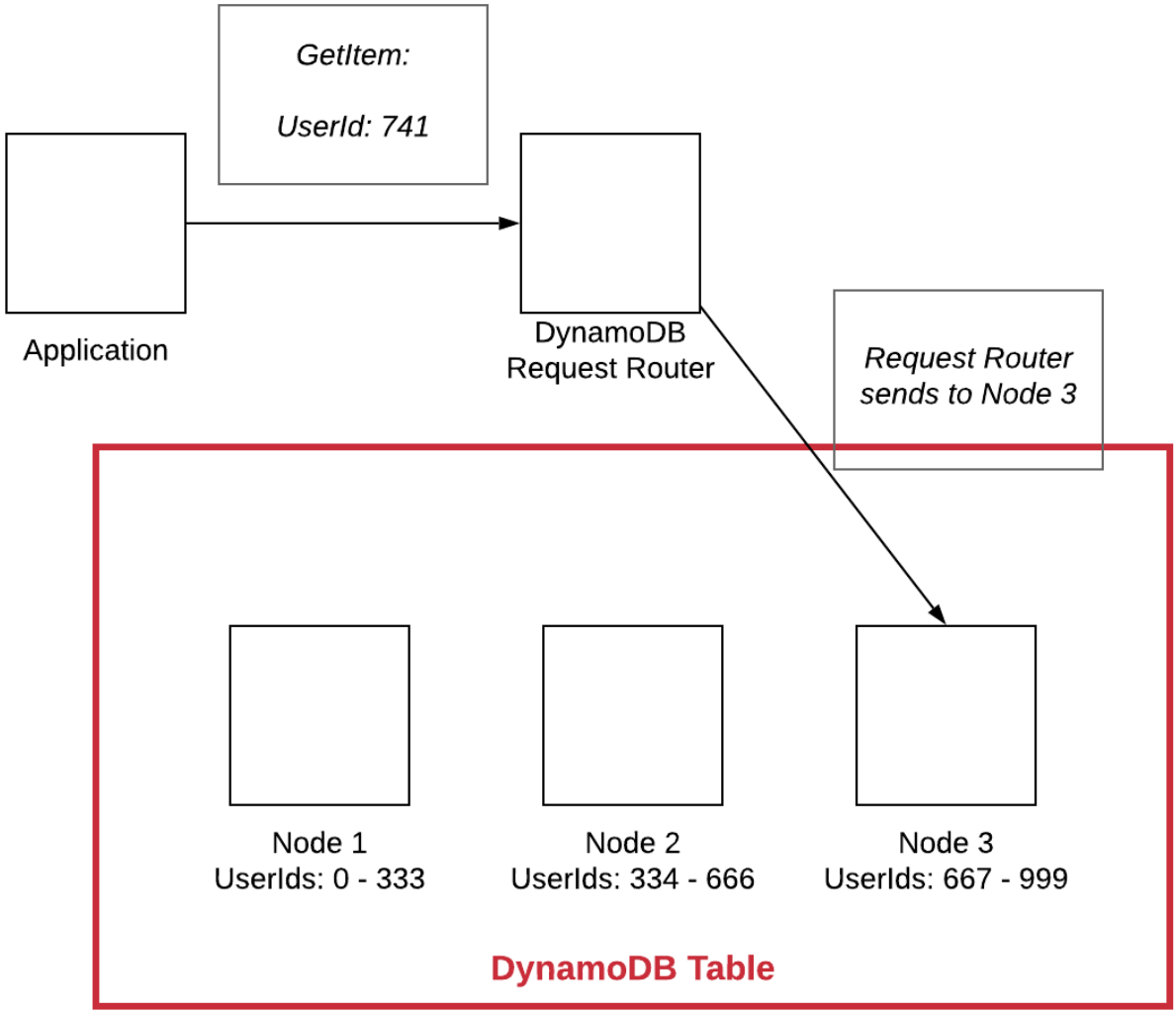
In the simplified diagram above, the red box indicates our DynamoDB table as a whole. Within the table, there are three nodes storing all of the data. Each node is the primary for a different subset of the data.

When a new write comes with a `UserId` value of `741`, the DynamoDB Request Router will determine which node should handle the data. In this case, it is routed to Node 3 since it is responsible for all `UserIds` between 667 and 999.



Note: Most NoSQL databases hash the partition key value before assigning it to a node. This helps distribute the data better, particularly when the partition key is monotonically increasing. For more information, check out the MongoDB documentation on hashed sharding.

At read time, all queries must include the partition key. Just as with the write, this operation can be sent to the node that is responsible for that chunk of the data without bothering the other nodes in the cluster.



This sharding mechanism is what allows for NoSQL's essentially infinite scale without performance degradation. As your data volume increases, you can add additional nodes as needed. Every operation hits only one node in the cluster.

How DynamoDB bounds your queries

The final way DynamoDB avoids scalability issues from RDBMS is that it bounds your queries.

We mentioned before that DynamoDB is basically a distributed hash table where the value of each key is a B-tree. Finding the value of the hash table is a quick operation, and sequentially traversing a B-tree is an efficient operation. However, if you have a large item collection, it could take a while to read and return the items. Imagine running a `Query` operation that

matched all items in an item collection that was 10GB in total. That's a lot of I/O, both on the disk and the network, to handle that much data.

Because of this, DynamoDB imposes a 1MB limit on `Query` and `Scan`, the two 'fetch many' read operations in DynamoDB. If your operation has additional results after 1MB, DynamoDB will return a `LastEvaluatedKey` property that you can use to handle pagination on the client side.

This limit is the final requirement for DynamoDB to offer guarantees of single-digit millisecond latency on any query at any scale. Let's review the operations involved in a large Query operation:

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	—	Sequential read. Limited to 1MB of data

There's just not much room for a single query's performance to degrade as your application scales. The initial operation to segment by partition key helps to narrow your table from gigantic scale (100TB) down to a single item collection (usually <10GB). Then the b-tree search is running against a manageable amount of data, and the disk risks are bounded.

One final note on the implications of this bounding before moving on. DynamoDB does not have support for any aggregation operations, such as `max`, `min`, `avg`, or `sum`. My guess is that it's due to this 1MB limit, as it could return inconsistent results. Imagine you have an

1GB item collection, and you run a `Query` against that item collection with a `max` aggregation on a particular attribute. DynamoDB only wants to read 1MB of data in that request, so the `max` would only apply to the 1MB it read, not the whole 1GB of data in the item collection.

This could confuse users as to the behavior of the aggregations. It's likely better for the application to handle aggregations itself at write time by storing aggregated information in a summary item in an item collection.

Further, if you *do* want the `max` value of that 1MB result set that was returned, it's pretty simple for you to calculate that client-side in your application on a fairly small amount of data. Thus, I think it's unlikely we'll see any explicit aggregation support on `Query` or `Scan` operations in the near future.

Where you might see performance problems with DynamoDB as you scale

In the sections above, we've seen how DynamoDB is designed to provide massive scale by making you consider scalability concerns up front. In the vast majority of applications, your DynamoDB table will perform the exact same in testing as it does with millions of requests per second.

However, there are two situations where you could see performance degradation as your application scales. Those situations are:

- Pagination
- Hot keys

Let's cover each of these below.

Pagination

Earlier, we saw how DynamoDB limits the result size from a `Query` or `Scan` operation to 1MB of data. But what happens if your operation has more than 1MB of data? DynamoDB

will return a `LastEvaluatedKey` property in your response. This property can be sent up with a follow-up request to continue paging through your query where you left off.

This pagination can bite you as you scale. When your data is small or in testing, you might not need to page through results, or it might just be a single follow-up request to fetch the second page of results. As your data grows, you may find this access pattern taking longer and longer as multiple pages are needed.

Fortunately, this problem is pretty easy to identify up front and consider. Most of the time, you can search your codebase for the string `LastEvaluatedKey`. If you don't see this property, it's likely that you aren't using pagination and can skip this section entirely.

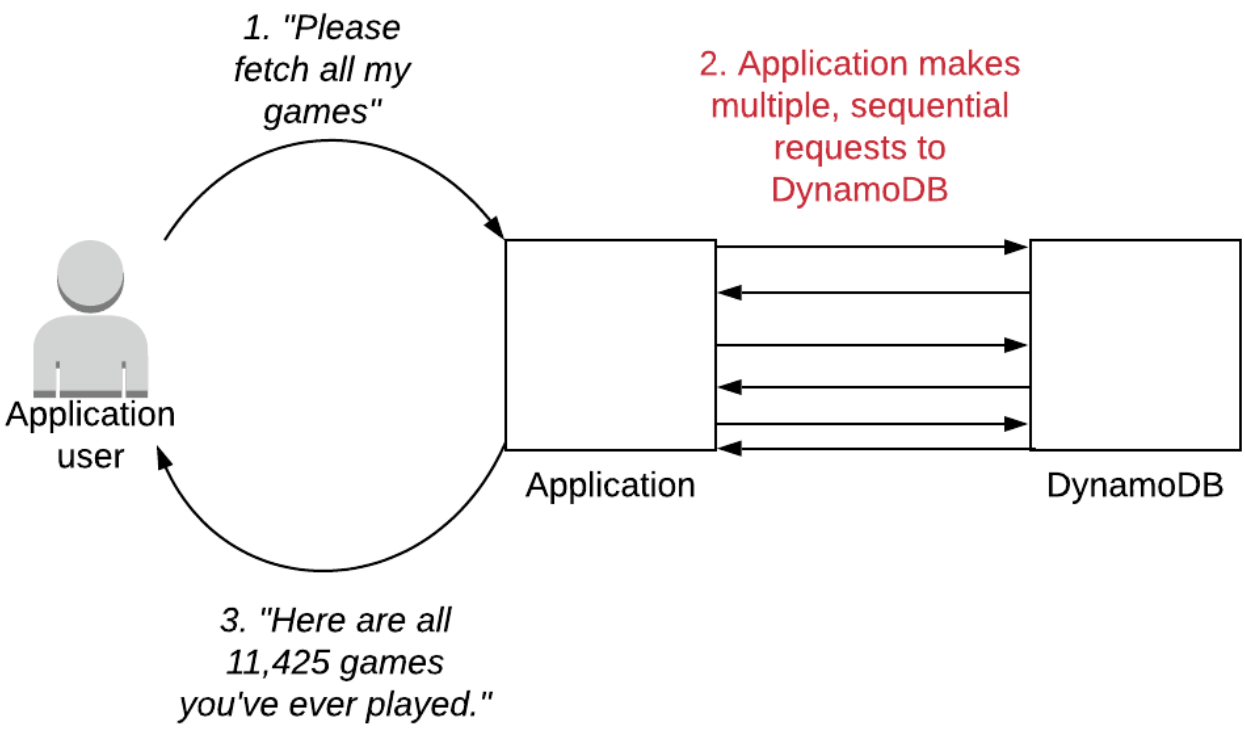
Note: There are other ways to paginate without using `LastEvaluatedKey`, such as if you're using a Query Paginator in Boto3 or if you're using a third-party DynamoDB client that supports easier pagination. If that's the case, you'll need to look closer to see if you're doing pagination.

If you do find yourself doing pagination, then you need to think harder. Consider the following questions:

- Is this pagination in a hot path where response time is important (e.g. an user-facing HTTP request)?
- Is there a chance that the result size of a query will grow to the point that it requires multiple (3+) pages to get through all the results?
- If yes to the two above, where should I handle pagination?

Let's walk through an example. Imagine you're building a multiplayer online game, like Fortnite, where players play multiple games over time. You have an access pattern — `ListGamesForPlayer` — where a player can view the past games they've played.

Over time, the number of games for a single player will increase such that you'll need to use pagination to get through the results. When you initially design it, you might think to fetch all the results in your application, then return them to the client:



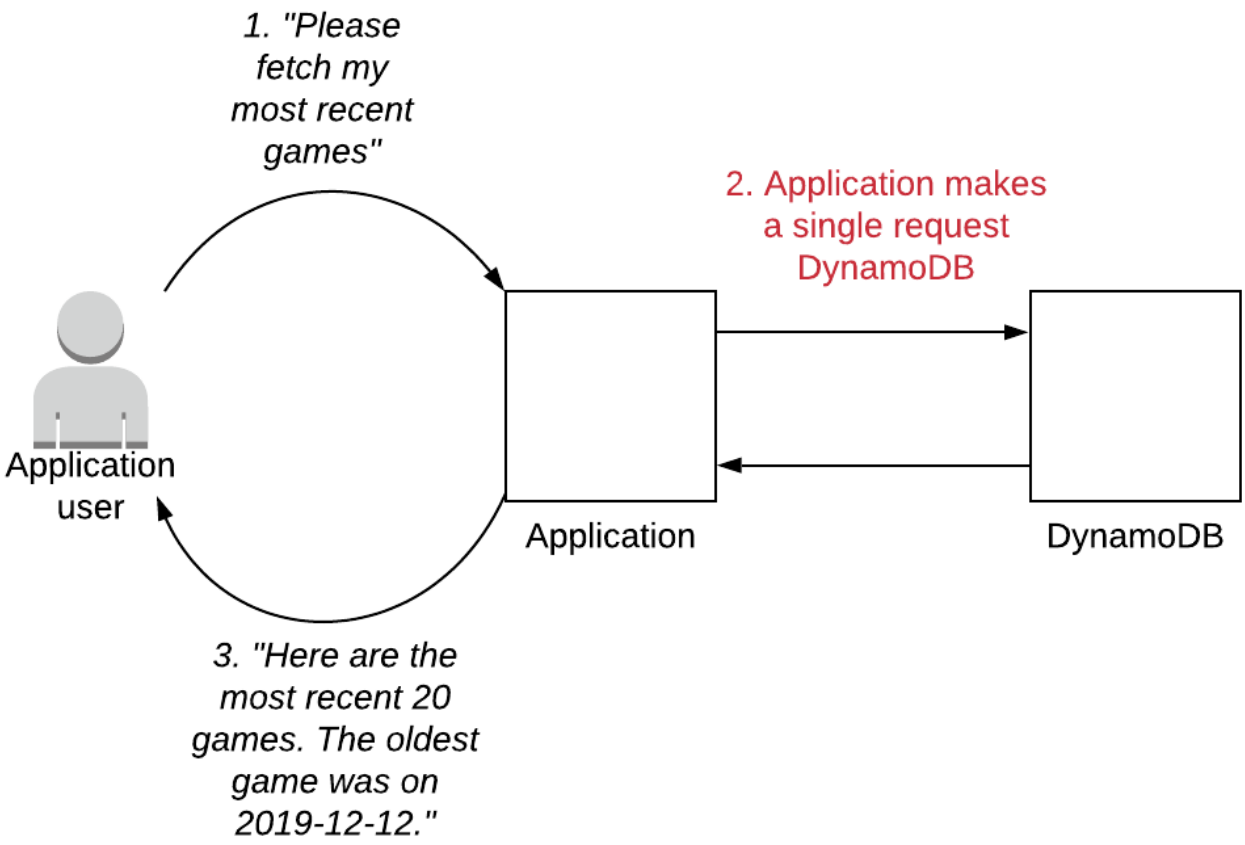
In the example above, the user makes a request to the backend for the user's games. In this case, there are a lot of games — over 11,000. The application fetches all of these games on the backend and returns all of them to the user.

This endpoint will slow down over time for players with a lot of games as the application server will be doing all the pagination in that single request from the client.

If you think about it, the player viewing these results probably can't handle viewing *all* of the results at one time anyway. Viewing 11,000 results in a one page is unmanageable. They likely want to see the 10 or 20 most recent results and only rarely want to dig deep into old results.

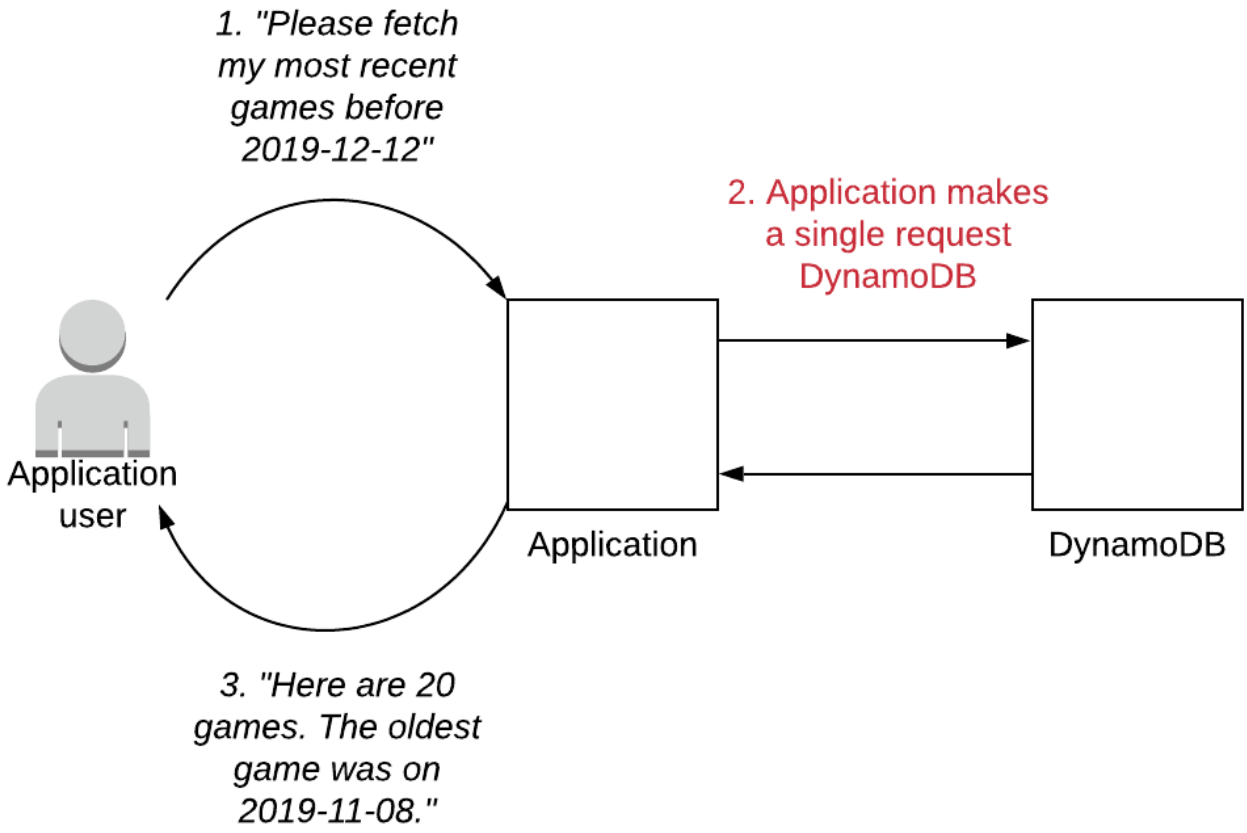
You could switch it up such that the pagination happens on the *client*. The `ListGamesForPlayer` endpoint only returns the most recent 20 results. If the user wants to see more, the client application will have a `Next` button or use infinite scroll to fetch additional games.

Now our access pattern looks like this:



The user requests a list of games. The application makes a single request to DynamoDB and returns the first twenty games as well as the time of the latest game it read. We can show twenty games in a UI pretty easily.

If the user wants to review older games, the frontend can use pages or infinite scroll to allow users to dig further. This would make a follow up request to the application to request games before the last one seen. We'd get the next twenty results to show the user.



Our endpoint stays snappy, and we only paginate through the data to fetch more results when it's actually needed by the user.

For additional information on pagination patterns in DynamoDB, be sure to read Yan Cui's post, Guys, we're doing pagination wrong …

Hot keys

The second, and more pernicious, way that your DynamoDB table will have trouble as it scales is through hot keys.

A hot key is an item collection that is accessed significantly more frequently than the other item collections in a table. For example, imagine Twitter. Justin Bieber, with his >100 million followers, is likely to be accessed significantly more frequently than me, with my slightly smaller following. If this data was read directly from DynamoDB, the Beliebers might cause a hot key on Justin's profile.

Hot keys can show up in a number of different patterns. You might have a single key that's consistently hot, such as in the Justin Bieber example above. This could also be true if you don't design your partition key properly and have an unbalanced workload. In a different vein, you could also have different keys become hot at different times. Imagine a flash deals site where different items went on sale for short periods of time. While an item is on sale, it will be accessed much more frequently than the other items.

There's some good news and bad news around hot keys.

The good news is that hot keys are mostly a problem for applications with really high scale. And this is a good problem to have! Congratulations on making something people use. For a while, you can choose to scale away from the problem. DynamoDB has a maximum of 3,000 RCUs and 1,000 WCUs against a single partition. RCUs and WCUs are on a per-second basis, so you could read a 4KB item 3,000 times per second and still be fine. That's plenty of headroom for the majority of applications.

Further, DynamoDB has done a lot of work in the past few years to help alleviate issues around hot keys. DynamoDB used to spread your provisioned throughput evenly across your partitions. This meant you needed to overprovision your throughput to handle your hottest partition. This changed in 2017 when DynamoDB announced adaptive capacity. Adaptive capacity automatically shifts your table's throughput to the partitions which need it the most. Initially, it would take a few minutes for adaptive capacity to react to a hot partition. In May 2019, AWS announced that DynamoDB adaptive capacity is now instant. Further, they announced a feature later in 2019 to automatically isolate hot items by moving them to another partition.

The bad news about hot keys is that it can be really tricky to know your hot keys when designing your table. There's no easy indicator like the `LastEvaluatedKey` property in the pagination use case. We're almost back in the SQL world where there's low visibility around how your query will perform as it scales.

One helpful tool in this area is CloudWatch Contributor Insights for DynamoDB. When you enable CloudWatch Contributor Insights on your DynamoDB table or secondary index, DynamoDB will log out the partition and sort keys accessed in your table. You can use CloudWatch charts to find your most frequently accessed keys and see how it compares to the rest of the items in your table.

This will be useful if you can generate representative traffic in your testing environment or even early on when you release your application. Monitoring these CloudWatch Contributor Insights can help you identify potential issues before you start running into the 3000 RCU limit on a single partition.

Conclusion

In this post, we got a peak behind the curtain into why DynamoDB can seem so inflexible when you first learn it. We saw how DynamoDB 'shifts left' on scalability by making you consider scalability during the table design phase. Then we saw the three reasons relational databases run into trouble at scale and the three design decisions DynamoDB has to avoid these issues. Finally, we looked at two areas where you could see performance problems with DynamoDB.

DynamoDB is a great tool, and you should consider when it's the right fit for your architecture. If you want a database that will scale seamlessly as your application grows in popularity, it's worth the time to learn DynamoDB and design your table properly.

The concepts in this post are necessarily compressed due to the format. If you're interested in more on this and related topics, be sure to check out Martin Kleppmann's excellent book, Designing Data-Intensive Applications.

If you have questions or comments on this piece, feel free to leave a note below or email me directly.

Published 6 Jan 2020

AWS DynamoDB

AWS Data Hero providing training and consulting with expertise in DynamoDB, serverless applications, and cloud-native technology.
Alex DeBrie on Twitter

LOG IN WITH
OR SIGN UP WITH DISQUS (?)

Name

Mathias Zarick • 5 months ago

Hi Alex, interesting article. Regarding the SQL example that queried "find the top 5 users by the amount spent". How can we resolve that with NoSQL? How we would need to model the data to be able to get this information efficiently without sorts and aggregates?
TIA Mathias

1 ^ | v • Reply • Share ›

Franck Pachot • 5 months ago • edited

Hi, I started to comment on "joins don't scale" and, given the length of it, I went through in a blog post of mine, with some examples and facts:
https://blog.dbi-services.com/the-myth-of-nosql-vs-rdbms-joins-dont-scale/
I like your contributions on DynamoDB, especially how you advocate for good data modeling, but I'm also fighting some myths about RDBMS and... this article explain very well a few of them ;)

1 ^ | v • Reply • Share ›

> **Alex D** Mod ➚ Franck Pachot • 5 months ago
>
> Great post, Franck! Thanks for sharing. Going to finish taking a look and then will share more broadly :)
>
> ^ | v • Reply • Share ›

Mahdir Ishmam • a year ago

Very well written article. I just don't agree with using DynamoDB if you ever plan on reaching scale. You have to partition your data in a way that keeps the read/writes below the imposed 3k/1k per second limit. I'd rather use Redis which is cheaper and handles scale way better. An r4 node from Elasticache can easily process 30k+ calls per second. I'd also look into MongoDB Atlas, which can be hosted on AWS.

AWS probably imposes those limits because their own internal infrastructure for DynamoDB would crap out if they allowed higher rate limits. Which speaks volumes of their product. They don't have this kind of rate limiting with MySQL or other managed service products.

Its a decent database for simple prototyping, but using it in production to handle workloads at scale... Seems like a nice way to have a lot of sleepless nights.

1 ^ | v • Reply • Share ›

> **Alex D** Mod ➚ Mahdir Ishmam • a year ago
>
> **@Mahdir Ishmam** Respectfully, I disagree :). There is a limit of 3k/1k per second per partition, but that gets you a heck of a long way. A partition is usually a very narrow subset of your data -- think a single user, or a single organization. It's

pretty rare that you'd be doing more than 3000 reads per second off a single user, and you can use DAX for caching if you do need more than that.

For some objective numbers, Amazon.com released details on their AWS usage on Prime Day 2019 -- their DynamoDB usage peaked at over 45 million requests per second.

Other databases don't have these published limits because they're more black box -- the performance you get depends a ton on the specifics of your workload. Regardless, I'd be surprised if you could get that kind of performance from a MySQL database.

Reply • Share ›

Peter Miller • a year ago

Fantastic post. I agree that Houlihan's talks are amazing. I'm starting to adopt DynamoDB in applications and this info is so useful coming from a relational background

1 Reply • Share ›

Alex D Mod → Peter Miller • a year ago

Thanks, **@Peter Miller** ! Glad it was helpful

Reply • Share ›

John Rodkey • 6 months ago

Scaling of relational dbs is becoming easier with tools like Vitess and Citus for example. Why not use relational and adopt one of these technologies instead of dynamodb?

Reply • Share ›

Alex D Mod → John Rodkey • 6 months ago

You're right that it is getting easier. However, at that point, you're trading off a lot of the features that most people like about a relational database. If you're sharding across multiple instances, you often lose the ability to do joins (or to do them efficiently) or do to ad-hoc analytics queries.

Additionally, there's usually more management overhead with Vitness or Citus. With DynamoDB, you get a fully-managed service that you don't really need to touch.

Different people can make different choices here, but I go with DynamoDB :)

Reply • Share ›

John Rodkey → Alex D • 6 months ago

Yes, I prefer NoSQL solutions for a few reasons with a tool like Bigquery or Redshift for analytics. I haven't had a chance to use dynamo for a project, I generally use Firestore today, but I may start a little side project just to give dynamo a try

Reply • Share ›

Basharat Wani • a year ago • edited

Very good article. I use DynamoDB a lot and we love it. As far the comparison goes between RDBMS and DynamoDB, I find lot of gaps in the writeup & humbly disagree around unbounded queries & data sharding in RDBMS (there are ways to smartly shard to avoid multi shard joins). Both DynamoDB and RDBMS have its pro and cons.

Reply • Share ›

> **Alex D** Mod → Basharat Wani • a year ago
>
> **@Basharat Wani** Thanks, and glad you liked it!
>
> Agree completely that DynamoDB and RDBMS have their pros and cons. I also agree that you *can* do a lot to scale with an RDBMS. The downsides of an RDBMS is that you can do it the wrong way too (and most generic designs will be wrong if you have multiple shards). There's nothing to limit you from shooting yourself in the foot.
>
> 1 Reply • Share ›

Zixma Liamg • a year ago

Thanks

Reply • Share ›

> **Alex D** Mod → Zixma Liamg • a year ago
>
> You're welcome!
>
> Reply • Share ›