S3 vs. DynamoDB

In this lesson, you'll draw a comparison between S3 and DynamoDB and decide what to use for your application.

WE'LL COVER THE FOLLOWING

- Comparing S3 with DynamoDB
  - Still need a relational database?

Comparing S3 with DynamoDB

S3 is designed for throughput, not necessarily predictable (or very low) latency. It can easily deal with bursts in traffic requests, especially if the requests are for different items.

DynamoDB is designed for low latency and sustained usage patterns. If the average item is relatively small, especially if items are less than 4KB, DynamoDB is significantly faster than S3 for individual operations. Although DynamoDB can scale on-demand, it does not do that as quickly as S3. If there are sudden bursts of traffic, requests to DynamoDB may end up throttled for a while.

S3 operations generally work on entire items. Atomic batch operations on groups of objects are not possible, and it’s difficult to work with parts of an individual object. There are some exceptions to this, such as retrieving byte ranges from an object, but appending content to a single item from multiple sources concurrently is not easy.

DynamoDB works with structured documents, so its smallest atom of operation is a property inside an item. You can, of course, store binary unstructured information to DynamoDB, but that’s not really the key use case. For structured documents, multiple writers can concurrently modify the properties of the same item, or even append to the same array. DynamoDB can efficiently handle batch operations and conditional updates, even atomic transactions on multiple items.

S3 is more useful for extract-transform-load data warehouse scenarios than for ad-hoc or online queries. There are services that allow querying structured data within S3, for example AWS Athena, but this is slow compared to DynamoDB and relational databases. DynamoDB understands the content of its items, and you can set up indexes for efficiently querying properties of items.

Both DynamoDB and S3 are designed for parallel work and shards (blocks of storage assigned to different processors), so they need to make allowances for consistency. S3 provides eventual consistency. With DynamoDB you can optionally enforce strong read consistency. This means that DynamoDB is better if you need to ensure that two different processes always get exactly the same information while a record is being updated.

S3 can pretend to be a web server and let end-user devices access objects directly using HTTPS. Accessing data inside Dynamo requires AWS SDK with IAM authorisation.

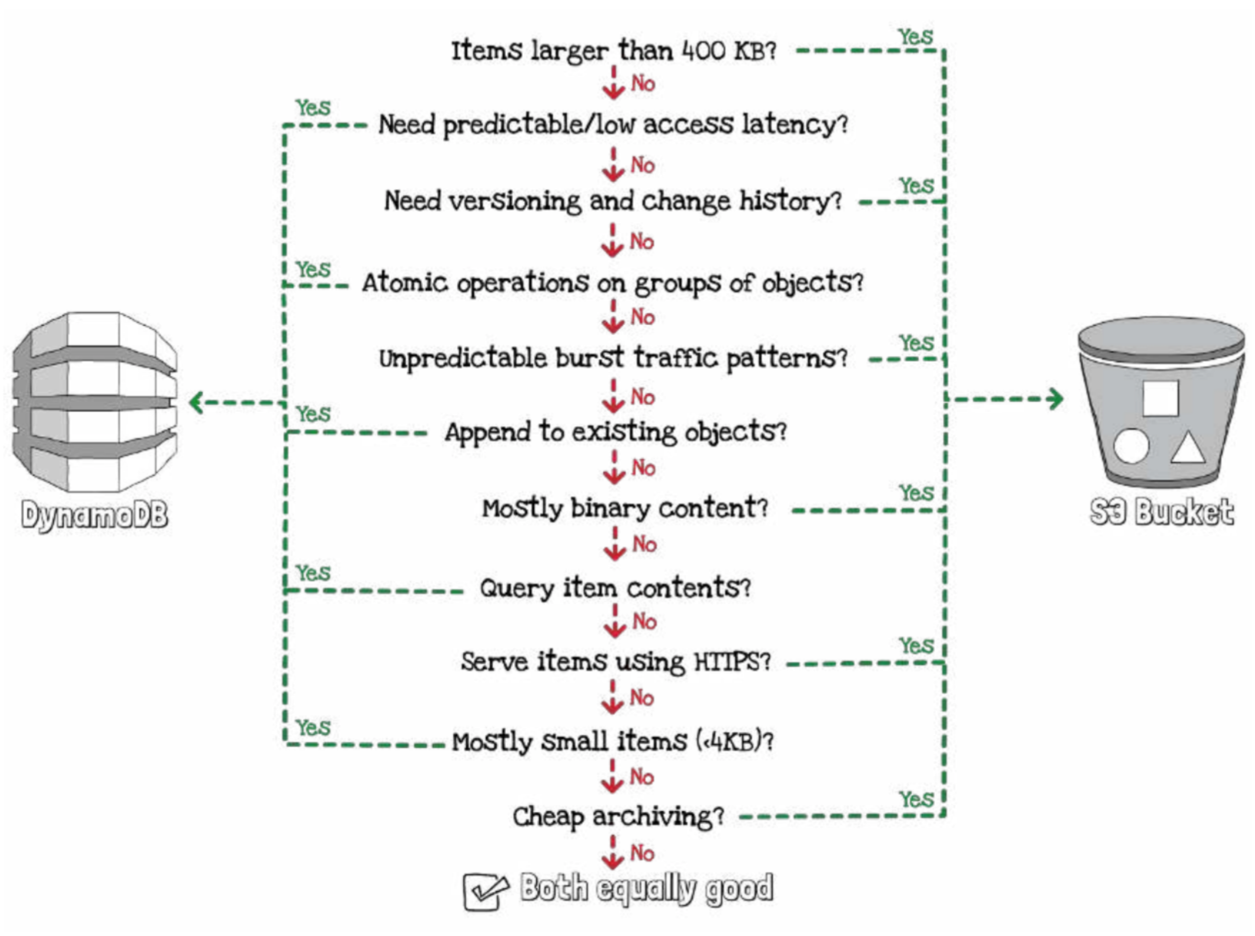
S3 supports automatic versioning, so it's trivially easy to track a history of changes or even revert an object to a previous state. Dynamo does not provide object versioning out of the box. You can implement it manually, but it's difficult to block the modification of old versions.

Although the pricing models are different enough that there is no straight comparison, with all other things equal, DynamoDB ends up being significantly cheaper for working with small items. On the other hand, S3 has several ways of cheaply archiving infrequently used objects. DynamoDB does not have multiple storage classes.

As a general rule of thumb, if you want to store potentially huge objects and only need to process individual objects at a time, choose S3. If you need to store small bits of structured data, with minimal latency, and potentially need to process groups of objects in atomic transactions, choose DynamoDB.

Both systems have workarounds for operations that are not as efficient as they would be in the other system. You can chunk large objects into DynamoDB items, and you can likewise set up a text search engine for large documents stored on S3. But some operations are significantly less hassle with documents stored on S3. But some operations are significantly less hassle with one system than with another.

The nice aspect of both DynamoDB and S3 is that you do not have to predict capacity or pay for installation fees. There is no upfront investment that you then need to justify by putting all your data into the same place, so you can mix both systems and use them for different types of information. Look at the different usage patterns for different blocks of data then choose between Dynamo or S3 for each individual data type.


Items larger than 400 KB?
Yes
No
Need predictable/low access latency?
Need versioning and change history?
Atomic operations on groups of objects?
Unpredictable burst traffic patterns?
Append to existing objects?
Mostly binary content?
Query item contents?
Serve items using HTTPS?
Mostly small items (<4KB)?
Cheap archiving?
Both equally good
DynamoDB
S3 Bucket

Typical usage patterns for S3 and DynamoDB

At MindMup, for example, we use S3 to store user files and most user requests, such as share invitations and conversion requests. We never need to run ad-hoc queries on those objects or process them in groups. We always access them by primary key, one at a time. We use DynamoDB to store account information, such as subscription data and payment references, because we often query this data based on attributes and want to sometimes process groups of related accounts together.

Still need a relational database?

> If you really need to get data coming from users into a relational database or a network file system from Lambda, it's often better to create two functions. One can be user-facing, outside a VPC, so that it can start quickly, write to a document database or to a transient external storage (such a queue), and respond to users quickly. The other function can move the data from the document database to the relational storage. You can put another service between the two functions, such as SQS or Kinesis, to buffer and constrain parallel migration work, so that you don't overload the downstream systems and that the processing does not suffer as much from cold starts.

In order to store data submitted by users with web forms, you don't necessarily need to ensure very low latency. Users will be submitting information over HTTPS connections anyway, so they won't notice a few dozen milliseconds more. On the other hand, you may want to collect large attachments at potentially unpredictable traffic spike intervals, so working with S3 will be easier.

Let's modify the application to save submitted forms to S3 before thanking the users. First, you'll need an S3 bucket so you can save information on it.

You could create an S3 bucket outside the SAM template and then pass its name to SAM as a parameter, similar to how you made the API stage name configurable in Chapter 6. This is a good option if you want the stack to connect to some pre-existing resources, for example, to integrate with a third-party service. In this case, you don't need to care too much about backwards compatibility with pre-existing resources, so it's easier to just create the bucket directly in the stack. In the resources section, for example just below the `Resources` header, you can add an entry of type `AWS::S3::Bucket` and give it some meaningful name, for example, `UploadS3Bucket`. You should put it on the same indentation level as the other resources, such as the old `WebApi`:

```
UploadS3Bucket:
  Type: AWS::S3::Bucket
```

Line 15 to Line 16 of code/ch7/template.yaml

Hopefully you can now make a good choice regarding the external storage that goes with the requirements of your application. Get ready to learn about Lambda access rights in the next lesson.